NTNU
Norwegian University of
Science and Technology

Real-Time GPU-Based 3D Ultrasound Reconstruction and Visualization

Holger Ludvigsen

Master of Science in Computer Science

Submission date:	June 2010
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Frank Lindseth, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

3D ultrasound reconstruction can be used to generate volume data from tracked real-time 2D ultrasound frames. Compared to other imaging modalities like MRI and CT, ultrasound is a flexible low-cost solution for generating 3D image maps of the internal organs of the human body using existing 2D ultrasound scanners. This makes ultrasound the modality of choice for intraoperative use and enables image guided surgery (e.g. neuro- or laparoscopic) where surgical instruments are safely navigated inside the human body.

Current CPU-based methods for 3D ultrasound reconstruction are time consuming (typically from 1 minute to 1 hour depending on the quality). The overall goal of this thesis is to use GPU-based techniques to achieve real-time (or close to real-time) reconstruction and visualization of ultrasound.

Step 1) GPU-based real-time 3D ultrasound reconstruction of freehand 2D scans using a tracked ultrasound probe (different algorithms, command-line based).

Step 2) Simultaneous reconstruction and visualization of the ultrasound volume as it gets built. It would be desirable for the surgeon to see a visualization of the volume while the data is acquired and the volume is generated (real-time volume rendering and slicing).

Technical issues such as parallelization and memory management techniques as well as recent platforms such as OpenCL, Nvidia Fermi and recent AMD graphics cards will also be evaluated.

Assignment given: 25. January 2010
Supervisor: Anne Cathrine Elster, IDI

Abstract

Ultrasound scanning is frequently used in medical practice because it is a non-invasive, safe and low-cost solution (*vs.* CT or MR). However, conventional ultrasound probes only provide 2D scans. 3D ultrasound reconstruction builds 2D scans into 3D volumes of the patient's internals. Since these volumes can be used for acquiring out-of-angle views, 3D rendering of the anatomy, and for image guided surgery, they are rapidly expanding the possible uses of ultrasound. However, the 3D reconstruction process is computationally demanding and includes processing millions of picture and volume elements. This process can currently take minutes or even hours on conventional systems.

It is very desirable to reconstruct ultrasound images in *real-time* to guide surgeons doing surgery. In this thesis, we manage to achieve this by utilizing the parallel processing power of GPUs with hundreds of computing cores. Our novel optimized methods take advantage of this power in order to perform entire volume reconstructions in only *fractions of a second.* Several optimization techniques have been developed, including only processing the relevant parts of the input. Novel methods for real-time incremental reconstruction producing high-quality results based on advanced interpolation techniques, are also presented.

Using our novel pixel-based and voxel-based methods, we are able to generate a volume of 67 million voxels in on 0.9 and 0.6 seconds, respectively. These results are based on the new NVIDIA Fermi GPUs, OpenCL and 434 tracked ultrasound scans. For high-quality incremental reconstruction, real-time processing times are obtained for methods based on distance weighted orthogonal projections and on the probe trajectory (PT). Our GPU implementations give a performance speedup of 14 for pixel-based methods, an impressive 51 for voxel-based methods, and speedup of 6-8 for the incremental methods, compared with single-threaded CPU implementations. The cubic interpolation of the PT method is shown to be superior to the others and preserves the most details. As for possible future work, we point out techniques for handling memory constraints, complex probe movement and the device-to-host transfer bottleneck.

Acknowledgements

This thesis is the result of work done at the HPC laboratory at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. The work has been done in collaboration with the Department of Medical Technology at SINTEF in Trondheim, Norway.

I would like to thank my supervisor Dr. Anne C. Elster (IDI) and my co-supervisor Dr. Frank Lindseth (SINTEF, IDI) for their supervision and help in completing this thesis. Dr. Elster is the backbone and driver of the HPC group at IDI and NTNU, and this thesis would not have been possible without the resources she has made available for the students at the HPC laboratory. Dr. Lindseth is a world leader in medical ultrasound, and his knowledge and experience was invaluable during my work. I also want to thank Ole Vegard Solberg (SINTEF) for assistance and very helpful input. Some of the GPUs used in the work was donated to the laboratory by NVIDIA and AMD. I wish to thank NVIDIA and AMD for their support of the laboratory.

Finally, I would like to thank my fellow students at the HPC group for ideas and support, especially my good friend Ahmed A. Aqrawi for his camaraderie and humorous discussions during those countless hours at the lab.

Trondheim, June 2010

Holger Ludvigsen

Holger Ludvigsen

Contents

List of Figures

List of Tables

List of Abbreviations

B-scan A 2D ultrasound image as acquired from an ultrasound machine

C2050 Nvidia Tesla C2050, a graphics card from Nvidia's GF100 (Fermi) architecture (released spring 2010) with 448 cores (see Table 3.1 for specifications)

CPU Central Processing Unit

CUDA Compute Unified Device Architecture, a GPGPU architecture and API from Nvidia

Device The device where parallel computations are offloaded to (e.g. GPU)

DWOP Reconstruction method based on Distance Weighted Orthogonal Projections [37]

FX5800 Nvidia Quadro FX5800, a graphics card from Nvidia's GT200 architecture preceding the GF100 with 240 cores (see Table 3.1 for specifications)

GPGPU General Purpose Computations on GPU

GPU Graphical Processing Unit, an accelerator located on a graphics card and offloading graphical operations from the CPU

HD5870 AMD ATI Radeon HD5870, a graphics card from AMD's recent HD500 series with 320 cores (see Table 3.1 for specifications)

Host The system that contains *devices* like GPUs

MPR Multiplanar Reformatting, a technique for visualizing volumetric data by planar slices through the volume

NDRange N-Dimensional Range, an index space for OpenCL work-items

PNN Reconstruction method based on Pixel-Nearest-Neighbor [20]

PT Reconstruction method based on Probe Trajectory [5]

ROI Region-Of-Interest

VNN Reconstruction method based on Voxel-Nearest-Neighbor [34]

Chapter 1

Introduction

It is of great value for medical doctors to be able to peer into our bodies in a non-invasive and safe manner when diagnosing patients. Ultrasound imaging fulfills this need and also offers an important imaging technique used in IGS (image guided surgery). Performing open surgery on a patient involves great risk and is often followed by a long hospitalization time. IGS enables minimally invasive procedures where surgical instruments are safely maneuvered with the help of imaging technology into the targeted area of the human body. By reconstructing a 3D volume from ultrasound scans, the internals of a patient can be visualized in ways not possible by scans alone, enabling image guidance and diagnostics. Both the reconstruction and visualization is computationally intensive, and it is desirable to see the constructed volume while it is being built.

Reconstruction using existing methods can take minutes if not hours. Reducing this process to only seconds would enable instant feedback *during* critical surgical operations. Furthermore, incremental reconstruction while the data is acquired makes it possible to rescan areas of interest as observed on simultaneous real-time visualization. In this thesis, we evaluate how GPUs (graphical processing units) can be utilized to perform fast 3D ultrasound reconstruction and visualization, and present methods for incremental and non-incremental reconstruction with simultaneous visualization on the GPU.

1.1 Goals

The goals of this thesis are to:

- Evaluate how to utilize the GPU in the most efficient manner to speed up computation time of 3D freehand ultrasound reconstruction.
- Develop methods to incrementally reconstruct the volume in real-time on the GPU as the data is acquired.
- While the data is being reconstructed on the GPU, visualize it in real-time.

1.2 Contributions

The main contributions of this thesis are:

- A novel method of doing real-time incremental 3D reconstruction using GPU and CPU.
- Optimization techniques for both pixel-based and voxel-based reconstruction on modern GPU architectures.
- A software implementation, called *Thunder*, of GPU-based ultrasound reconstruction and visualization including:
    - Fast reconstruction by voxel-based and pixel-based algorithms.
    - High-quality reconstruction in real-time as data is acquired incrementally.
    - Simultaneous reconstruction and visualization by both slices through the volume and ray casting on the GPU.
    - Multiplatform implementation using the OpenCL standard with performance results on the newest GPU hardware architectures from Nvidia and AMD.

1.3 Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 introduces the field of 3D ultrasonography. Methods of reconstructing a volume from ultrasound scans and techniques to visualize the volume are described. Previous work in the field of ultrasound reconstruction and visualization is also presented.

Chapter 3 introduces general purpose computations on GPUs (GPGPU). A brief history and current state of GPU computing is presented, and the recent OpenCL architecture for GPGPU is described.

Chapter 4 describes the approach developed in this thesis where the GPU is utilized to perform fast reconstruction of ultrasound scans acquired in beforehand.

Chapter 5 describes the approach developed in this thesis for real-time incremental reconstruction and volume visualization performed on the GPU simultaneously as the data is acquired.

Chapter 6 presents the performance and quality of the ultrasound reconstruction and visualization as performed in this thesis. These results are also discussed together with other important issues.

Chapter 7 summarizes the findings of the work described in this thesis and presents possible avenues of future work.

Appendix A contains an annotated bibliography of selected references.

Appendix B contains large uncropped versions of selected figures.

Appendix C contains a poster summarizing the principles and main results of this thesis with focus on the incremental reconstruction.

Appendix D contains additional numerical test measurements.

Appendix E contains a paper about ray tracing on the GPU using the Nvidia OptiX library, including volume ray casting.

Appendix F contains selected code listings from implementations of the methods described in this thesis.

Chapter 2

3D Ultrasonography

Ultrasound is a non-invasive, safe, low cost and practical way to provide medical doctors with an internal view of a patient's body. By reconstructing a 3D volume from hundreds of ultrasound scans, physicians can see the internals of the body in ways otherwise impossible from scans alone. There are different ways to reconstruct the volume, and these differ in the reconstruction quality and speed. There are also different ways to visualize the resulting volume. This chapter describes how ultrasound is used in medicine and introduces different methods of reconstruction and visualization. Previous work in the field of ultrasound reconstruction and visualization is also presented.

2.1 Medical Ultrasound

In medicine, ultrasound probes such as the one shown in Figure 2.1 are used to obtain 2D ultrasound scans called *b-scans*. The probe both emits ultrasound pulses and detects their echo returned. See Figure 2.2 for an illustration. As the ultrasound pulse travels through tissue, echoes are created when it encounters materials with varying density. Some of the pulse energy is absorbed by the tissue, some is reflected back as echo and some continues to move forward.

The speed of sound depends on the transmission material, but one can assume a constant speed of $v_{sound} = 1540\ m/s$ in human tissue [3]. It should be noted that in reality the speed varies depending on the material, and can be as low as 1450 m/s in fat and over 1600 m/s in muscle, but we do not take this into consideration. This means that the distance d from the transducer to

Figure 2.1: GE M12L linear array ultrasound probe (GE Healthcare, Waukesha, Wisconsin, USA)

the density variation can be estimated by the time t it took for the echo to arrive at the transducer:

$$d = \frac{tv_{sound}}{2} \tag{2.1}$$

The strength of the echo is given by the difference between acoustic impedances of the materials next to each other. If the probe emits and receives ultrasound in a linear array, a 2D image can be formed where the height is the distance from the probe, the width is the width of the array and the intensity of a pixel is the strength of the echo. The resulting image is the ultrasound

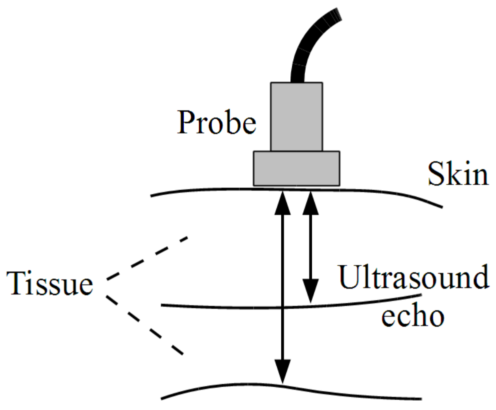


Figure 2.2: The basic principles of ultrasound in medicine

b-scan. The array can also be curved, giving a *curvilinear* b-scan. Examples of linear and curvilinear b-scans are shown in Figure 2.3.

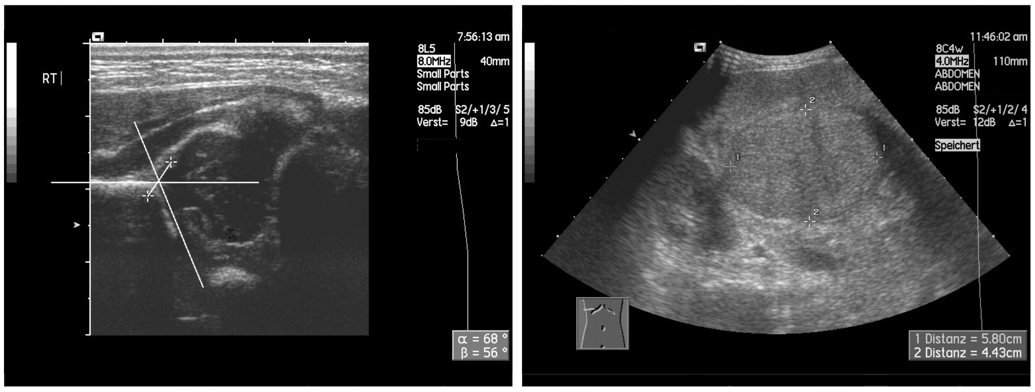


Figure 2.3: Linear and curvilinear b-scans (used with permission from A. Christaras)

To construct a 3D volume, each b-scan is tagged with a timestamp and location and rotation of the probe in 3D space. This data together with the b-scans is then processed, which can take minutes to hours depending on the desired quality of the reconstructed output. Once reconstructed, the volume can be used for many purposes. Typically, it can be visualized using direct volume rendering through ray casting or similar methods. But other techniques are also used, such as multiplanar reformatting slices (MPR slices) that are created from the volume. The MPR slices are planar slices of sampled points through the volume, like cutting it in half with a sharp flat knife. More details about visualization of 3D ultrasound data is found in Section 2.4, and description of how to reconstruct a volume from b-scans is found in Section 2.3.

2.2 Tracking in Ultrasound

Most reconstruction methods rely on an accurate tracking of the ultrasound probe, and there are several ways to obtain this. Most methods use either electromagnetic, optical, mechanical or acoustic sensors, but an alternative is to estimate the orientation from the ultrasound scans themselves. Such sensorless tracking can be done by analyzing speckle noise in the scans using decorrelation or linear regression techniques. However, sensorless systems do not offer the same accuracy as with actual sensors [21].

Mechanical tracking methods involve attaching the probe to structures with certain degrees of freedom, or letting a machine move the probe automatically. Disadvantages with this method is the reduced freedom and that additionally only one probe at the time can be tracked. Acoustic tracking uses sound emitters and receivers. The position can be tracked by measuring the time it takes for the sound to reach the receiver, or by measuring the relative phase shift when moving the probe. Disadvantages include the required line of sight for the sound waves, and varying speed of sound depending on factors such as temperature and pressure. Electromagnetic tracking systems measure current induced by moving a receiver in an electromagnetic field. This method has the advantage of being resistant against occlusion, but metallic objects as well as power sources or CRT monitors can distort the field.

An optical tracking system such as the one used for data collection for our thesis is shown in Figure 2.4. Spheres that reflect infrared light are attached to the probe, and two cameras record the reflected infrared light. The position of the spheres relative to the cameras can then be used to estimate the location and orientation of the probe. Obviously, the system requires that the probe is in line of sight from the cameras.

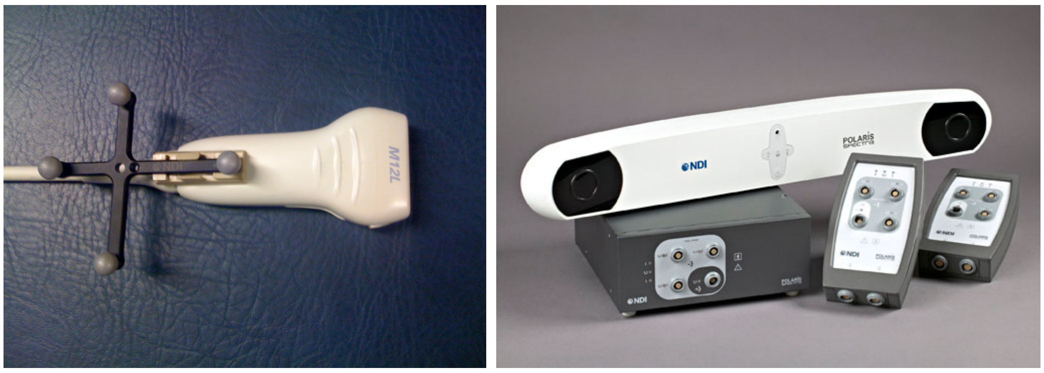


Figure 2.4: GE M12L linear array probe (GE Healthcare, Waukesha, Wisconsin, USA) with tracking frame and Polaris Spectra optical tracking system (Northern Digital, Waterloo, Canada) (Photo of Polaris Spectra used with permission from Northern Digital Inc.)

2.3 3D Ultrasound Reconstruction

There is more than one approach to reconstructing a 3D volume from a series of 2D b-scans. Two main categories of algorithms exist: *pixel-based* and *voxel-based* reconstruction. These are sometimes also called *forward* and *backward* reconstruction, and this section describes the two approaches. The section also describes the basics of geometric transformations in 3D space that are used by these methods.

2.3.1 Geometric Transformations

A geometric transformation is an operation on a set of coordinates [6]. Operations that can be applied include rotation, translation, resizing and skewing. A practical way to represent a transformation on a set of initial coordinates **P**, is by a *transformation matrix* **T**:

$$\mathbf{S} = \mathbf{T} \cdot \mathbf{P} \tag{2.2}$$

where **S** are the transformed coordinates. To make it possible to represent translation by transformation matrices, we convert **P** to *homogenized coordinates* $\mathbf{P}'$ by extending it with one dimension with a scalar value of 1. Given that **P** is a vector of N dimensions, then $\mathbf{P}'$ is of $N+1$ dimensions and **T** is a $(N+1) \times (N+1)$ matrix. The product $\mathbf{T} \cdot \mathbf{P}'$ will then give a homogenized $N+1$ vector that can be converted to normal coordinates by dividing each element by the value of the added dimension. Typically as this value is still 1, we can simply omit the last dimension.

The identity matrix represent an empty transformation with no effect. The transformations used in this thesis are rotations and translations. Equation 2.3 give a translation matrix and Equations 2.4, 2.5 and 2.6 give rotation matrices around the x, y and z axis,

$$\mathbf{T}_{translate} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$\mathbf{T}_{rotatex} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

$$\mathbf{T}_{rotatey} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

$$\mathbf{T}_{rotatez} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

where (x, y, z) are the translations in the x, y and z dimension and θ is the rotation angle. Representing combined translation and rotation is simply done by multiplying transformation matrices into one matrix. An example involving a translation and two rotations is given below:

$$\mathbf{S} = \mathbf{T}_{translate} \cdot \mathbf{T}_{rotatex} \cdot \mathbf{T}_{rotatey} \cdot \mathbf{P} = \mathbf{T}_{combined} \cdot \mathbf{P} \tag{2.7}$$

where

$$\mathbf{T}_{combined} = \mathbf{T}_{translate} \cdot \mathbf{T}_{rotatex} \cdot \mathbf{T}_{rotatey} \tag{2.8}$$

The orientation of an ultrasound b-scan can be represented by a transformation matrix that is given by a tracking system (which is the case in this thesis).

2.3.2 Pixel-Based Reconstruction

Pixel-based reconstruction, also known as forward reconstruction, iterates over the ultrasound b-scans and attempts to project their values into a volume. The term pixel-based comes from that this approach involves processing each b-scan pixel, and since the b-scans are the input, and the volume is the output this is dubbed as a forward method, as opposed to the voxel-based method described in the next section.

For each b-scan j with orientation given by $\mathbf{T}_j$, each pixel i on the b-scan at location $(x_i, y_i, 0)$ with intensity $c_{j,i}$ has a contribution $V_{i,j}$ to the volume given by:

$$V_{i,j}(x,y,z) = m(x_i,y_i)w(|(x,y,z) - \mathbf{T}_j \cdot (x_i,y_i,0)|)c_{j,i} \tag{2.9}$$

where w is a weighting function and m is a 2D mask function defining a region-of-interest (ROI) in the b-scan with the value 1 inside the region and 0 outside. The weighting function can be based on the distance between the voxel and the pixel, and examples of such weighting functions include Gaussian bell (Equation 2.10) and inverse distance (Equation 2.11) [11].

$$w_{Gaussian}(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{2.10}$$

$$w_{inverse}(x) = \frac{1}{x} \tag{2.11}$$

To perform pixel based reconstruction, the following algorithm can be used as basis (based on [35]):

```
• for each b-scan $j$:
    ○ for each pixel $i$ inside mask:
        ▷ $v$ = the coordinates of the pixel in volume space.
        ▷ for each voxel $v'$ in kernel $k$ around $v$:
            – add to the voxel's value the pixel's value weighted by the
              weighting function $w$.
```

A number of variations of this basic algorithm exist. The size of the kernel and the definition of the weighting function can obviously be changed, and the simplest case is to let the pixel contribute to only the closest voxel without any weighting. This is called pixel-nearest-neighbor (PNN).

A problem with PNN is that some voxels may never be filled from any pixels, and this is especially a problem when adjacent b-scans are far apart and thus have much space between them. To fix this, a hole-filling stage is required after the PNN volume filling. This can be done by iterating over all unfilled voxels and set them to the average of all filled neighbors in a kernel around them.

A consideration needs to be done when filling a voxel with a value, as it might already be filled from another pixel. The scheme chosen is called a *compounding method*. Common compounding methods are:

- overwrite the old value
- overwrite the old value only if the voxel has not yet been filled
- compute an average/median of the values
- keep the maximum value

If overwriting, one can lose data if the new value is erroneous due to noise in tracking data or ultrasound sensoring. Averaging reduces noise, but smudges the volume. Keeping the maximum can be useful to avoid deleting values by overwriting them with empty (i.e. dark, low-intensity) values.

2.3.3 Voxel-Based Reconstruction

Voxel-based reconstruction methods iterates over the volume to be reconstructed and for each voxel determines which pixels influence it. This is called a backwards method since it is based on the output (the volume). Some of the principles explained for pixel-based methods also apply in voxel-based methods: Pixels outside the mask should not influence the volume and a compounding method must be chosen.

Each voxel is processed to determine which pixels influence it, and how they do it. Several variants of voxel-based reconstruction exist, but the following algorithm can be used as a basis (based on [35]):

- for each voxel j:
    - ∘ for each b-scan i that is close to the voxel j:
        - ▷ for each pixel p in i that is inside the mask:
            - – add to the voxel's value the pixel's value weighted by the weighting function w.

Several variants of voxel-based methods exist. If only the closest pixel to the voxel is used without any weighting, then the method is called voxel-nearest-neighbor (VNN). Since all voxels are traversed in voxel-based reconstruction, no hole-filling is necessary.

2.4 3D Ultrasound Visualization

The goal of ultrasound reconstruction is to visualize the resulting data. Conventional ultrasound scans offer only a 2D view in the direction the ultrasound probe is pointing, but through a reconstructed volume, slices in any direction can be obtained. Furthermore, volume rendering techniques can give an overview of the area of interest in three dimensions, akin to looking into the body itself. This section explains two common visualization techniques for reconstructed ultrasound volumes. The first is multiplanar reformatting slices through the volume, and the second is volume rendering by ray casting.

2.4.1 Multiplanar Reformatting Slices

Multiplanar reformatting (MPR) is a technique for generating sagittal, coronal, and oblique views from otherwise one-axial sections [30, 9]. Parts of the human body can be difficult, if not impossible, to scan in any direction, such as down the length of the torso. Also, when scanning a brain covered in ultrasound-resistant bone skull, only a small surgically-made hole will provide a spot for the probe, not giving much freedom of movement. With MPR, one is able to generate slices in any arbitrary direction. Typically, three orthogonal directions are sufficient for clinical utility. Figure 2.5 shows an example of MPR. Notice how all three dimensions (width, height and depth) can be seen.

The MPR slices can be generated directly without constructing an actual volume, and this is demonstrated in the Stradx system [29]. But, the reconstructed volume is useful for other purposes such as volume rendering and storing offline for later use. In this thesis, we choose to obtain the MPR slices from a reconstructed volume.

2.4.2 Volume Ray Casting

Ray casting is a technique where thousands of rays, one for each rendered pixel, are cast towards the scene [30, 19]. When the rays intersect the geometry they accumulate color, similar to how light rays work in nature, only in reverse. An example of a ray casted volume is given in Figure 2.6. The technique is suitable for rendering volumes without a defined surface, but that are defined by a massive "cloud" of voxels. An alternative is to extract

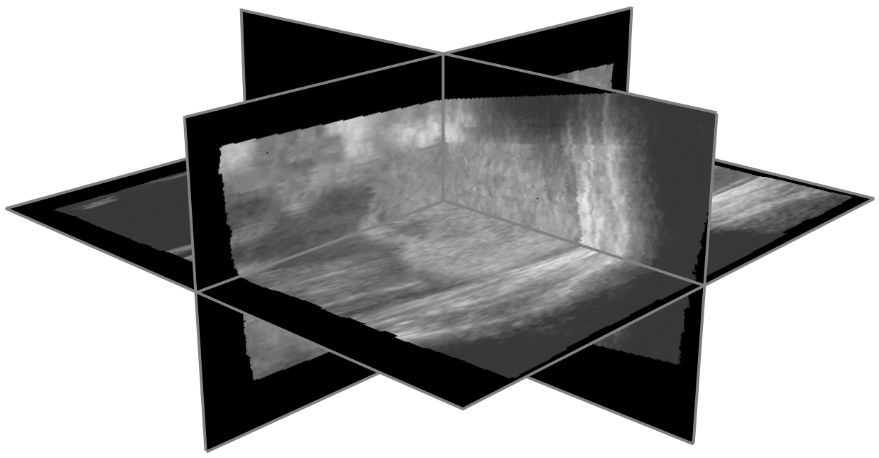


Figure 2.5: Multiplanar reformatting of ultrasound (generated by our implementation)

a surface by thresholding the voxel values and constructing facets on it by such algorithms as marching cubes [18]. However, with noisy data such as ultrasound scans, it is difficult to extract the exact surface [32]. A better approach may be to render all voxels so that trained medical personnel can use their expert knowledge and experience to interpret the visualized data.

The volume ray casting algorithm can be described as follows (based on [30]):

- for each ray:
    - step along the ray direction in small incremental steps
    - for each step:
        - if inside a voxel, sample the volume and accumulate the value

Several methods for accumulating the voxel values exist, ranging from trivial addition, to complex estimation of how light is absorbed by mass in nature. Each ray has a defined origin (the camera position) and direction (determined by field-of-view and camera direction). The step size should ideally be such that each voxel is sampled only once and no voxel in the ray's path is missed, i.e. a step size larger than the distance between two neighbor voxels along an axis, and lower than the distance between two neighboring voxels along the diagonal.

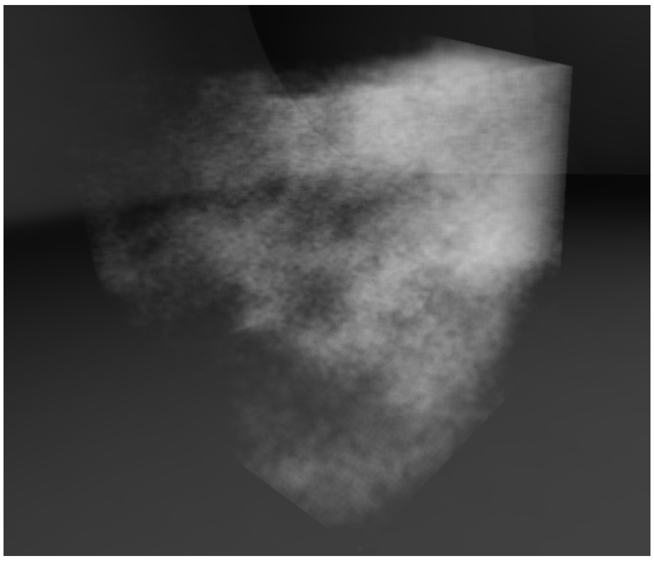


Figure 2.6: A ray casted render of a fractal volume (from our previous work [19])

2.5 Previous Work on 3D Ultrasound Reconstruction

Previous work in the field of 3D ultrasound reconstruction include novel reconstruction algorithms and methods, as well as designed and implemented production systems. The field of ultrasound and volume visualization is extensive in itself. Nelson *et al.* [22] present approaches to visualization of 3D ultrasound data including simple ray casting and MPR slice visualization, and Ludvigsen *et al.* [19] describe GPU-based volume ray casting with the Nvidia OptiX library. The rest of this section will focus on work in 3D ultrasound reconstruction. In Appendix A, an annotated bibliography of selected references can be found.

2.5.1 Categorization of Reconstruction Methods

The vast number of existing reconstruction approaches have been reviewed and categorized. Fenster *et al.* [8] present several approaches in 3D ultrasound imaging not limited to how to just reconstruct, but also the acquisition of input data and how to render the reconstructed results. An attempt at grouping existing reconstruction algorithms is done by Rohling *et* *al.* [31]. Their categorization of the methods is broadly defined as either *voxel nearest-neighbor interpolation*, *pixel nearest-neighbor interpolation* or *distance-weighted interpolation.* Solberg *et al.* [35] present a different grouping that take additional types of algorithms into account and has a clearer separation between categories. Their grouping is into *voxel-based*, *pixel-based* and *function-based* methods, and their work also includes a thorough comparison of the algorithms based on performance and quality.

2.5.2 Implemented Reconstruction Systems

Trobaugh *et al.* [37] present a formative description of a system where optical tracking is used to orient a freehand ultrasound probe, and includes volume reconstruction by both a voxel-based and a pixel-based method. A different approach is described by Prager *et al.* [29]. Their *Stradx* system does not reconstruct a volume, but instead generates MPR slices directly from the ultrasound b-scans. Welch *et al.* [39] describes a volume reconstructing system under development that allows for updates to the volume during scanning and also simultaneous visualization at *near* real-time performance. Another attempt presented by Gobbi *et al.* [10] consists of an implemented system that does simultaneous real-time 3D ultrasound reconstruction and visualization, but is limited to the simple PNN method and only orthogonal MPR slice visualization. Furthermore, the visualization is not synchronous with the reconstruction, and is updated at a lower non-real-time framerate.

2.5.3 Reconstruction Algorithms

3D ultrasound reconstruction is often a trade-off between performance and quality. Trobaugh *et al.* [37] and Gobbi *et al.* [10] use the simple PNN method to enable high performance. The alternative voxel-nearest-neighbor is used by Sherebrin *et al.* [34] in their 3D ultrasound system. Barry *et al.* [4] use a more sophisticated pixel-based method with an inverse distance weighting kernel around inserted pixels. Different approaches are described by Rohling *et al.* [31] and Sanches *et al.* [33] that fall into the function-based category according the terminology of Solberg *et al.* [35]. Rohling *et al.* use splines to construct a volume from the input b-scans, and Sanches *et al.* use statistical methods to estimate a function for the interpolation. A recent voxel-based method is described by Coupe *et al.* [5] that takes the probe trajectory into account to improve reconstruction quality, especially for sparse input where there is much space between the b-scans. A performance increasing scheme for fast slice selection is described by Wein *et al.* [38] and benefits voxel-based reconstruction methods. Karamalis *et al.* [15] describe a high performing hybrid reconstruction method partially implemented using GPU texture interpolation features. Huang *et al.* [14] describe a technique for utilizing the Fourier domain to take redundant frequency components into account, preserving the high frequencies and resulting in better resolutions.

Chapter 3

General Purpose Computations on GPUs

In the field of high performance computing, powerful computers are combined with engineering ingenuity to solve computationally hard problems such as weather forecasting, fluid simulation and seismic processing for oil exploration. Computational performance, i.e. how many floating point numbers can be calculated per second (FLOP/s), can be used to estimate how quickly problems can be solved. An intuitive way to achieve higher throughput is to perform calculations in parallel where possible [40], as exemplified by super computers consisting of thousands of computers merged into a bigger and often expensive machine. But parallelization is also desirable for desktop computers due to limits for the frequency that a computer core can operate at, and for the amount of power it can require.

A recent alternative to super computers and compute clusters is general purpose computations on graphical processing units (GPGPU) [28, 36]. This computer architecture is cheap, has a low Watt per FLOP/s and currently offers parallelity of up to hundreds of computation cores on a single commodity graphics card. With the OpenCL standard [16], it is also possible to use any parallel accelerator such as a multicore CPU, IBM and Sony's CELL BE processor or a digital signal processor (DSP), and not only the GPU.

In this chapter, a brief history and current state of GPGPU is presented, followed by an introduction to OpenCL, a platform independent standard for parallel computations. This chapter gives motivation for doing computations on GPUs, and explains how OpenCL provides a standard interface and programming model for GPGPU.

3.1 GPGPU Computing

GPU technology was developed for graphics processing in computer games for the purpose of offloading calculations involved in 2D and 3D graphics from the CPU. Early GPUs were fixed hardware accelerators specialized to perform the most common graphical operations. Graphical applications have the tendency to involve the same computations on different data, thus the GPU employs massive hardware parallelism for the computations. As an example, consider the task of rotating an object in 2D graphical applications [12]. Each point $\mathbf{P}_i$ is rotated into $\mathbf{S}_i$:

$$\mathbf{S}_i = \mathbf{R} \cdot \mathbf{P}_i \tag{3.1}$$

where $\mathbf{R}$ is a 2×2 rotation matrix and the same for all points. Each rotated point $\mathbf{S}_i$ can be evaluated individually in parallel. With e.g. thousands of points, this can reduce computation time drastically compared to evaluating them one by one after each other. Fixed function GPUs have since evolved into programmable units with the same parallel architecture, but where the parallel operations can be programmed for each stage of the graphics pipeline. Although meant for graphics processing such as 3D shading, these programmable GPUs have started to be used for general purpose computations. Applications that have a natural affinity for the GPU architecture involve executing similar calculations on thousands, if not millions, of data elements. GPU manufacturers such as Nvidia and AMD have recognized this potential in their products, and have released GPGPU frameworks such as Nvidia's CUDA (Compute Unified Device Architecture) [17, 25] and AMD's ATI Stream technology [2].

Current state of GPGPU

Many examples of utilizing the GPU for general purpose computations exist. In the field of medical ultrasound, Nielsen [23] uses the GPU for image enhancement of ultrasound through the wavelet transform. Another example is Herikstad [13], who estimates and corrects aberration in ultrasound scans on the GPU.

The third generation of Nvidia GPUs targeting GPGPU computing are based on the GF100 architecture. The GF100 is used in Nvidia products such as the GeForce GTX470 for computer games and Tesla C2050 for high performance computing. AMD's competing GPU generation is the HD5000 series, which is used in products such as HD5870. Specifications for the C2050 and HD5870 series are given in Table 3.1. The Nvidia Quadro FX5800, which is similar to the Nvidia Tesla C1060 from Nvidia's second GPU generation is included for comparison and since it is also used for test measurements in this thesis. Note that memory bandwidth and performance are highly theoretical figures stated by the manufactorers [27, 24, 1].

GPU	**C2050 [27]**	**FX5800 [24]**	**HD5870 [1]**
# of cores	448	240	320[1]
Clock frequency	1150 MHz	1296 MHz	850 MHz
Memory	3 GB	4 GB	1 GB
Theoretical memory bandwidth	144 GB/s	102 GB/s	153 GB/s
Theoretical performance	1030 GFLOP/s	933 GFLOP/s	2720 GFLOP/s

Table 3.1: Nvidia C2050, FX5800 and AMD HD5870 specifications

3.2 OpenCL

Open Computing Language Specification (OpenCL) is an open royalty-free standard for general purpose parallel programming [16] designed to be independent of platform and vendor, whether it be CPU, GPU or other class of accelerator. A detailed investigation of OpenCL and comparison with CUDA can be found in [7]. The OpenCL standard consists of an architecture, an API and a programming language. The architecture is a model of the environment that the computations are performed in, including computational devices such as GPUs and a host system such as a x86 platform that contains the devices. The API is an interface for the host system to build, launch and coordinate parallel computations on the devices. The programming language is a version of the C programming language intended for writing the programs, termed *kernels*, that are executed in parallel on the devices.

A readily available implementation of OpenCL is provided by Nvidia based on their CUDA architecture [25]. There are over one hundred million GPUs sold that can execute CUDA programs. Additionally, AMD also has support for OpenCL as part of their ATI Stream technology [2]. So there is lots of hardware out there to exploit.

[1]HD5870 has 320 *stream cores* with a total of 1600 *processing elements*

3.2.1 OpenCL Architecture

The OpenCL architecture defines models for the host system and its computation devices, how the parallel computations are enqueued and executed, and the memory hierarchy on the devices.

Platform Model

Figure 3.1 depicts the OpenCL platform model. The *host* system is connected to one or more *devices*. The devices consist of one or more *compute units* that have a number of *processing elements*. These processing elements do the actual computations. An example of a host system is a x86 desktop computer. Typical devices include GPUs, digital signal processors (DSPs), IBM and Sony's CELL BE processors and even multicore CPUs.

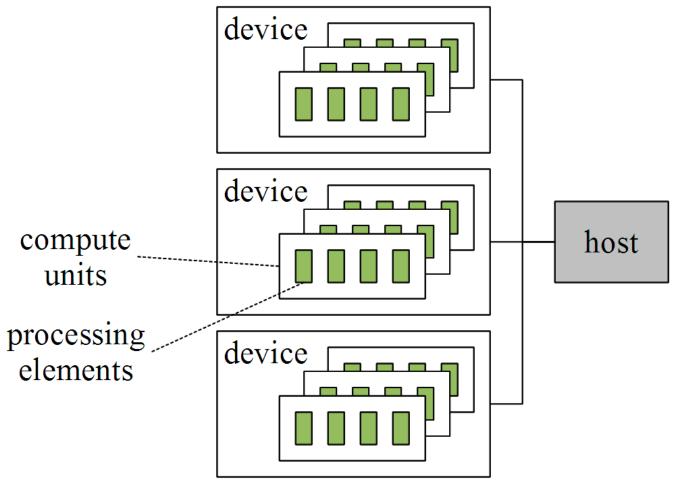


Figure 3.1: OpenCL platform model

Execution Model

Figure 3.2 shows the OpenCL execution model. The model is based on the single-instruction multiple-data model where the same operations are performed on different pieces of data. In OpenCL terminology, a *work-item* is a thread that executes a *kernel*. The kernel is written in the OpenCL C-based programming language, which will be described later. Work-items are organized in one, two or three dimensional *work-groups*, and all the work-groups make up a *NDRange* which is an index space in the same number of dimensions as the work-groups. Each work-item has a unique index in this *NDRange*.

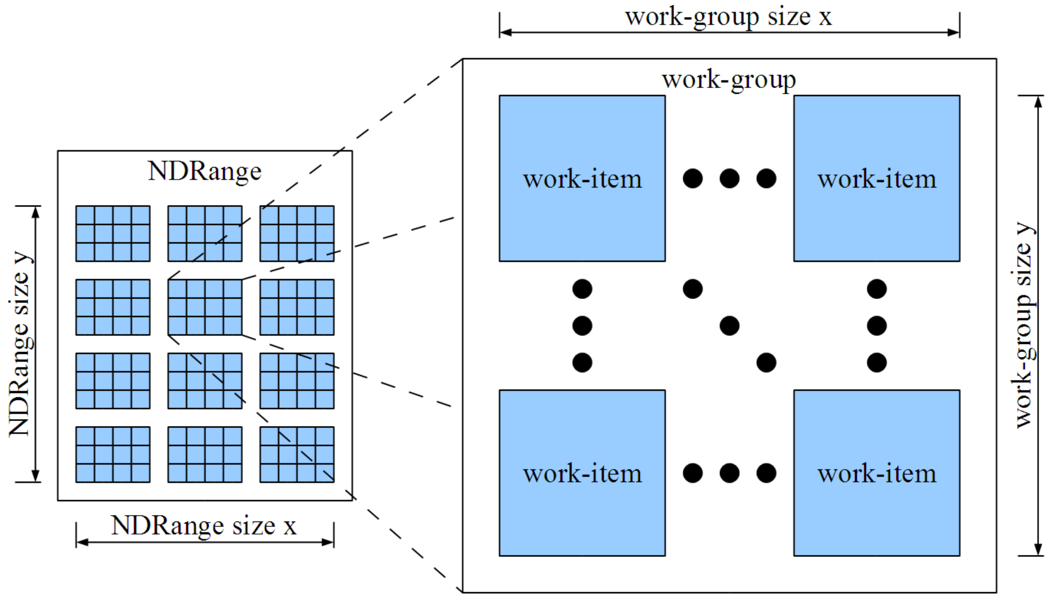


Figure 3.2: OpenCL execution model

The execution of parallel kernels, memory transfers and synchronization in OpenCL is organized through *command queues*. The tasks are called *commands*, and are inserted into queues to be performed on or in association with a device. The order of execution can be either synchronous (*in-order*) or asynchronous (*out-of-order*). When in-order, the commands are launched and completed in the order they appear in the queue. When out-of-order, the commands are launched in order, but are not guaranteed to complete in order. The execution environment is defined by a *context* which holds all the objects used during execution, such as devices, memory and command queues. Figure 3.3 shows a context containing command queues that are mapped to devices.

Memory Model

Figure 3.4 shows the OpenCL memory model. The model is closely tied to the platform model. The main memory on the device is the *global memory*, which together with the read-only *constant memory* is read- and writeable by

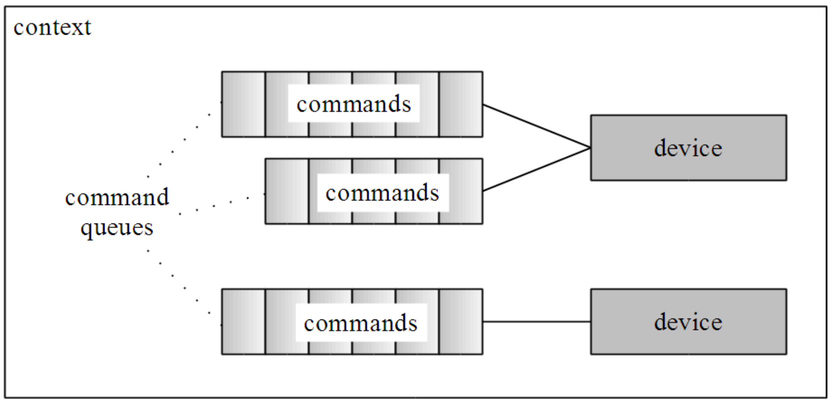


Figure 3.3: OpenCL command queues and devices

the host system. Global and constant memory may be cached on the device. Each compute unit has a *local memory* with the scope of individual work-groups, and each processing element has a *private memory* where data with local scope to individual work-items is stored. Private and local memory are not directly accessible from host.

3.2.2 OpenCL *vs.* CUDA

Historically, Nvidia's CUDA precedes OpenCL. CUDA was initially launched as a host API and programming model for Nvidia's GPUs. It's popularity made it a *de facto* standard for GPGPU development. Nvidia has taken this a step further and introduced a GPU architecture dubbed the CUDA architecture, and Nvidia's OpenCL implementation runs on the CUDA architecture. Even though CUDA is an architecture, the CUDA API and programming model still exists and is heavily used, and the OpenCL architecture clearly resembles the CUDA architecture. Thus, at the time of writing, OpenCL is an alternative to the CUDA API and programming model. However, Nvidia has implemented the OpenCL standard on top of their CUDA architecture, and in a similar fashion AMD has implemented OpenCL on their ATI Stream technology.

Some major concepts in OpenCL are analogue to concepts in CUDA. For readers experienced with CUDA, Table 3.2 shows CUDA terminology and corresponding equivalent in OpenCL.

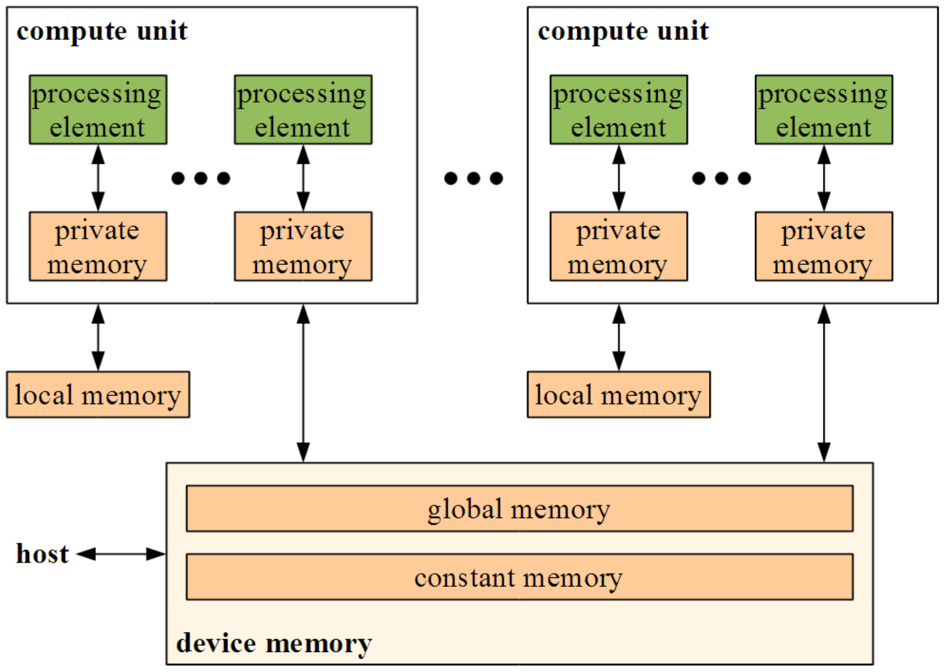


Figure 3.4: OpenCL memory model

3.2.3 Host Programming in OpenCL

OpenCL provides a host API for building, launching and coordinating parallel computations on devices. It is also possible to extract platform dependent parameters such as memory sizes, maximum work-item sizes and other platform capabilities. This section explains how to use memory, program and kernel objects from the host system, and how to perform synchronization of the parallel computations. For a complete overview, see [16].

Using Memory Objects

A *memory object* is a part of device memory together with its attributes. By creating memory objects, the host allocates memory on the device. There are two types of memory objects. *Buffers* are sequential arrays of scalar data types such as integers or floating point numbers, and are accessed as a series of bytes. *Images* are two- or three- dimensional arrays for the purpose of containing images such as textures. An important difference between images and buffers is that images are accessed through *samplers* with defined mechanisms for out-of-range coordinates, interpolation between values and filtering. This section will focus on buffer objects.

OpenCL terminology	CUDA terminology
kernel	kernel
host	host
NDRange	grid
work-item	thread
work-group	block
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory
compute unit	stream multiprocessor
processing element	core
image	texture

Table 3.2: OpenCL *vs.* CUDA terminology

Creating a buffer object is performed by the `clCreateBuffer` function:

```
cl_mem clCreateBuffer(cl_context context,
                      cl_mem_flags flags,
                      size_t size,
                      void * host_ptr,
                      cl_int * errorcode_ret)
```

where `context` is the OpenCL context that will contain the buffer; `flags` are one or more flags defining attributes of the buffer as given in Table 3.3; `size` is the buffer size in bytes and `host_ptr` is an optional host memory pointer that is used depending on the flags given. The function returns a device memory pointer to the allocated buffer. `clCreateBuffer` also returns an error code in `errorcode_ret` that is not equal to 0 if something went wrong, as most of the other OpenCL API calls also do.

Flag	Description
`CL_MEM_READ_WRITE`	Default flag. Buffer is read and written by kernels
`CL_MEM_WRITE_ONLY`	Write-only access from kernels
`CL_MEM_READ_ONLY`	Read-only access from kernels
`CL_MEM_USE_HOST_PTR`	Use previously allocated `host_ptr` as the storage area for the buffer instead of device memory
`CL_MEM_ALLOC_HOST_PTR`	Allocate *new* memory on host instead of device
`CL_MEM_COPY_HOST_PTR`	Copy data from given `host_ptr` to new buffer

Table 3.3: OpenCL buffer flags

Reading and writing buffer objects is performed by the `clEnqueue[Read|Write]Buffer` functions:

```
cl_int clEnqueue[Read|Write]Buffer(cl_command_queue cmd_queue,
                                   cl_mem buffer,
                                   cl_bool blocking,
                                   size_t offset,
                                   size_t size,
                                   void * ptr,
                                   cl_uint num_events,
                                   cl_event * event_list,
                                   cl_event * event)
```

where `cmd_queue` is the command queue that enqueues the read/write operation, `buffer` is the buffer object of `size` bytes and `blocking` is true/false depending on if the function should be blocking or not. If blocking is enabled, the function will not return until reading/writing has completed. `offset` is an offset into the buffer object to write or read from and `ptr` is a pointer to the location on the host to be written to or read from by the procedure.

All functions that enqueue commands optionally have an associated *event object* returned via the parameter `event`, that can be used to query for the status of the command or wait for its completion (if non-blocking). Furthermore, `event_list` can contain a list of `num_events` events that needs to be completed before this command is executed.

Using Program and Kernel Objects

An OpenCL *program* is a set of kernels that can be built (compiled and linked) and be executed on specified devices. The kernel is usually defined by a string of code in the OpenCL C-based programming language, but can also be a previously compiled binary. The approach described here takes kernels as code strings. The program is created with the `clCreateProgramWithSource` function and built with the `clBuildProgram` function, respectively. For clarity, their argument lists are not stated explicitly here, but can be found in [16].

When the program is built, it is possible to create, set arguments to, and execute *kernel objects* with the functions `clCreateKernel`, `clSetKernelArg` and `clEnqueueNDRangeKernel`. To create a kernel, a successfully built program and a kernel name is supplied. The total size of the arguments cannot exceed a platform dependent maximum (e.g. 4352 bytes for Quadro FX5800). Kernels are enqueued on a command queue like the buffer read/write operations. When enqueueing a kernel, the size and dimensionality of the *NDRange* that it will operate on must be specified. A *global work size* is given in n work dimensions and defines the total number of work-items in the *NDRange*. An n-dimensional *local work size* is also given such that the size in each dimension evenly divides the corresponding sizes in the global work size. The local work size defines the work-group size. Events can be used for the kernel commands just like with buffer operations mentioned above.

Host Synchronization

The previously mentioned event system handles fine-grained synchronization between specific commands in the command queue. Each command can optionally have an associated event, and other commands may depend sequentially on zero or more events before execution can take place.

For global synchronization in a command queue, the host can use the `clFLush` and `clFinish` functions. The former requires all commands in a queue be issued to their associated device, but does not have to guarantee that they complete before the function returns. The latter also blocks until they have completed, resulting in a full synchronization of commands.

3.2.4 Device Programming in OpenCL

A kernel that executes in parallel on a device is written in the OpenCL C-based programming language. This language is based on the C99 standard, but with specific extensions and restrictions. For details, we refer to [16]. The same kernel will be executed in parallel by potentially thousands of work-items in a *NDRange*. An example of a $N \times N$ matrix multiplication kernel can be found in Figure 3.5. The kernel is written so that each work-item computes one value of the output matrix product, implying a two-dimensional *NDRange* of size (N, N).

The OpenCL C-based programming language supports vector arithmetic with integer or floating point vectors of 2, 4, 8 and 16 elements. A number of built-in functions are provided for scalar and vector math, queries about *NDRange* dimensions and work-item index, and for local or global synchronization. Some restrictions for use of the C99 standard apply, and these include: no recursion, use of `stdio` routines or external variables.

```
__kernel void transform(__global float * matrix_C,
                        __global float * matrix_A,
                        __global float * matrix_B,
                        __constant int N) {
    int x = get_global_id(0);
    int y = get_global_id(1);

    float sum = 0;
    for (int i = 0; i < N; i++)
        sum += matrix_A[i][y] * matrix_B[x][i];

    matrix_C[x][y] = sum;
}
```

Figure 3.5: OpenCL matrix multiplication kernel

Chapter 4

Fast Reconstruction on the GPU

In this chapter, we present our methods for performing fast ultrasound reconstruction using the GPU. Both pixel-based and voxel-based approaches are considered, and the parallelization, implementation and optimization will be described. Here, the input data has been acquired before reconstruction; and so it is not *incremental* at this time. Some tasks are common for both pixel- and voxel-based reconstruction, and these are described first.

4.1 Preprocessing of Input Data

For both pixel-based and voxel-based reconstruction, the input data is processed before the actual reconstruction. In our case, this means calibrating the tracking data and handling the different rate of tracking data and ultrasound scans.

4.1.1 Calibration of Tracking Data

Figure 4.1 shows a tracking system with a sensor attached to an ultrasound probe. The tracking system will give the relation between the sensor and the world space, $T_{w \leftarrow s}$, in the form of a transformation matrix. This transforms coordinates in the space of the sensor to coordinates in the world reference space. However, even though the sensor is tightly attached to the probe, it is not in the origin of the space of the ultrasound images. This relation,

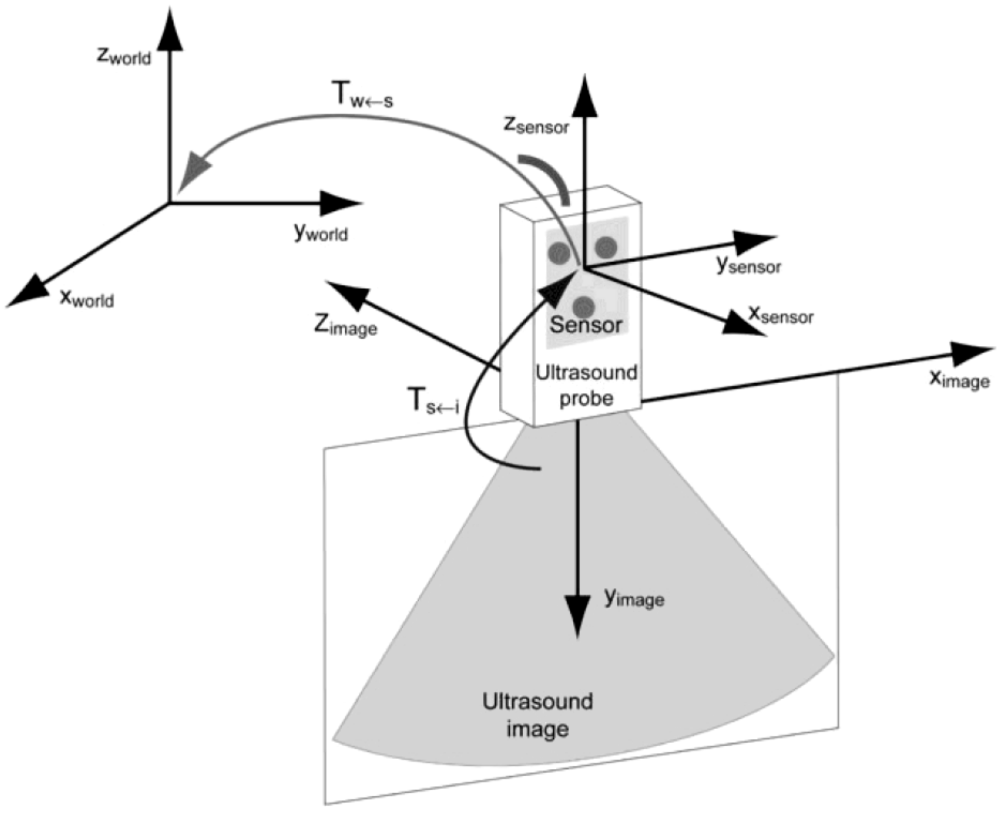


Figure 4.1: Spaces of a tracking system (used with permission from [21])

$T_{s \leftarrow i}$ varies with each tracking system's setup, so a calibration transformation is required and is part of the input. Before the reconstruction begins this calibration matrix is multiplied into the tracking transformations.

4.1.2 Handling Different Tracking and Scanning Rate

While the probe catches ultrasound images, the tracking system outputs a series of transformation matrices. Ideally, the tracking system and ultrasound probe should be synchronized such that each b-scan has an associated transformation. However, the tracking system used in this thesis has a slighter higher rate than the probe. Is is assumed that both systems start and stop at the same time. As an example, one of the input set used in this thesis has 434 b-scans and 520 tracked positions.

This is handled by interpolating the tracking data stream between the b-scan stream. Since the tracking data consists of 4×3 matrices of floating point values, it is obviously easier to interpolate than the b-scans with thousands of pixels. Each b-scan and tracked position has an associated timetag, but the probe and tracking system use clocks with different timestamp resolutions.

Given these considerations, the timetags are first normalized to the same unit by first subtracting the first timetag from all timetags in each stream respectively. In this manner, each stream begins with the time 0. Then, each tracking timetag is multiplied by the ratio between the last b-scan timetag and the last tracking timetag. This makes the tracking timetags end with the same timetag as the b-scan timetags.

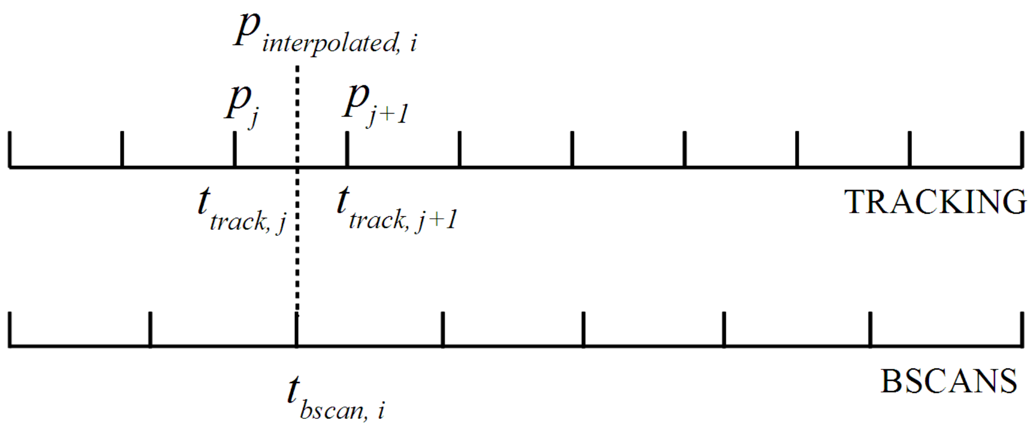


Figure 4.2: Interpolation of tracking data after timetag normalization

Figure 4.2 shows how the tracking data is interpolated using the normalized timetags. Each interpolated value is given by Equation 4.1 where $p_{interpolated,i}$ is the value interpolated between p_j and p_{j+1} given the timetags t_{bscan} and t_{track} of the b-scans and tracking data. The values in the first and last tracking matrices need not be interpolated (because of tracking and scanning starting and stopping at the same time).

$$p_{interpolated,i} = p_j + \frac{t_{bscan,i} - t_{track,j}}{t_{track,j+1} - t_{track,j}} \cdot (p_{j+1} - p_j) \tag{4.1}$$

4.2 Pixel-Based Reconstruction

Pixel-based reconstruction is an intuitive method to construct a volume from oriented ultrasound scans. In abstract sense, the scans are simply "inserted" into the volume. However, for a computer to this, a series of algorithmic steps needs to be devised. To obtain high performance, we can parallelize this algorithm and optimize the implementation. This section explains how pixel-based reconstruction is done in this thesis, including how this is parallelized and optimized for the GPU.

4.2.1 Method

The method is based on pixel-nearest-neighbor (PNN) [20]. The input is a set of n b-scans images, a mask defining a region of interest (ROI) and n tracking matrices. The b-scans are $w \times h$ arrays of grey-scale intensities, with 1 byte per pixel ($2^8 = 256$ possible values). The mask is the same format, where black is outside the ROI and white is inside, but there is only 1 mask for all n b-scans. The tracking matrices are interpolated and calibrated using the 4×3 floating point transformation matrices as described in the previous section. The bottom row in the matrices will always be $(0, 0, 0, 1)$, and is omitted.

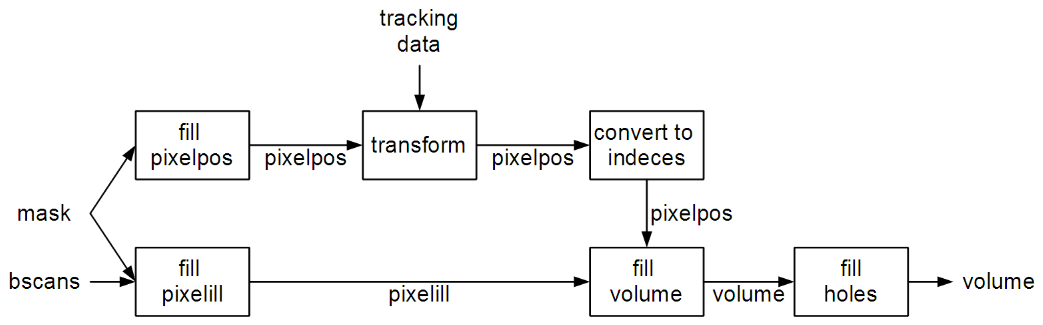


Figure 4.3: Steps performed in pixel-based reconstruction

There are 6 steps to be performed, resulting in the reconstructed volume. These steps are illustrated in Figure 4.3 and given in the list below:

1. Fill *pixelpos* from mask
2. Fill *pixelill* from b-scans and mask
3. Transform *pixelpos* by tracking matrices
4. Convert coordinates to volume indices
5. Fill volume from *pixelill* and *pixelpos*
6. Fill volume holes

In the first step, we construct an array of three-dimensional coordinates, *pixelpos*, with one position for each pixel that is in the ROI in the mask. The coordinates are in world space, and are such that all the b-scans lie flat on the YZ-plane. This is given by Equation 4.2 where i is the b-scan index (disregarded), x and y are the pixel indices, and Δx and Δy are the spacings between pixels in x- and y-direction (given as part of input). These coordinates are to be rotated by the tracking matrices to be positioned in the volume.

$$pixelpos(i, x, y) = (0, x\Delta x, y\Delta y) \tag{4.2}$$

In the second step, the grey-scale intensities of the pixels in the ROI are saved as an array of bytes. The reason for doing this apparently redundant work is that the ROI can be much smaller than the entire b-scan. Typically, the scans are captured by a frame-grabber card connected to the ultrasound machine, and these images include metadata and empty space around the actual ultrasound data. The ROI is then just a fraction of the entire image. As an example, in the test data used in this thesis the ROI is 28 % of the full scan. By extracting these into a separate array, we save memory, and there is a practical one-to-one mapping between pixel coordinates and pixel intensities.

With this data ready, the transformation can begin in step three. For each b-scan, the corresponding tracking matrix is multiplied with the pixel coordinates to yield the coordinates in the volume. The operation is shown in Equation 4.3. These coordinates are in world space, and in step four these are converted to volume indices by Equation 4.4 where Δv is a given spacing between neighbor voxels. With the transformed coordinates converted to volume indices, the volume can be populated in step five. For each of the processed pixel coordinates, their corresponding pixel intensity is inserted into the volume using one of the possible compounding methods given in Table 4.1. The effect of each specific method is given as an expression of the *old* value present in the voxel and the *new* value to be inserted.

$$pixelpos(i, x, y) := \mathbf{T}_i \cdot pixelpos(i, x, y) \tag{4.3}$$

$$pixelpos(i, x, y) := \Delta v \cdot pixelpos(i, x, y) \tag{4.4}$$

The sixth and final step is to fill the volume holes. For each empty voxel, a $k \times k \times k$ kernel around it is averaged to give the voxel a value. In this kernel, only

Method	Effect	Condition
average	$voxel \leftarrow \frac{old+new}{2}$	old is not empty
average	$voxel \leftarrow new$	old is empty
max	$voxel \leftarrow max(old, new)$	
ifempty	$voxel \leftarrow new$	old is empty
ifempty	$voxel \leftarrow old$	old is not empty
overwrite	$voxel \leftarrow new$	

Table 4.1: Compounding methods

the non-empty voxels are considered for calculating the average, otherwise the filled holes would be darker than the surroundings. To save memory, hole filling is done in place, and thus there is a need for differentiating between actual holes and the empty voxel values before the first and after the last b-scan, in addition to those outside the region-of-interest. If every empty voxel would simply be filled, then non-empty voxels at the edges would be smudged out in the empty regions. This is resolved by counting the number of empty voxels in the kernel, and then only using the average to fill the hole if this number is above a cutoff value. An example cutoff value is $(k^3/2) - k$, which is slightly lower than half the number of voxels in the kernel. If the holes are not filled in place, e.g. by using a separate copy of the volume that has its holes filled, this cutoff is not necessary.

4.2.2 Parallelization

Code listings of the implemented OpenCL kernels can be found in Appendix F.1. To parallelize the pixel-based reconstruction on the GPU, we need to divide the work into parts that can be performed simultaneously by many threads. To fully utilize the GPU, the number of work-items should in the order of hundreds of thousands or even millions [26]. An elegant split of the work-domain will also ease the design of the kernels. OpenCL supports an *NDRange* of up to three dimensions, but this multidimensional aspect does not play a role in overall performance [26], so this is not a consideration here.

Another task when parallelizing is to decide what parts to run on the highly parallel GPU and what parts to run on the relatively sequential CPU. There are additional overheads associated with doing work on the GPU, such as transferring data to and from the device memory and launching the kernel. The task of processing the input by calibration and interpolation is too small to be worth parallelizing for the GPU. However, each step in the reconstruction is parallelized.

After the input b-scans and tracking data has been transferred to the GPU memory, *pixelpos* and *pixelill* are generated and stored on the device. These arrays are only needed intermittently during reconstruction and thus never need to be transferred to or from the GPU. There are $n \times w \times h$ pixels that need to be processed into $n \times m$ coordinates in *pixelpos*, and intensities in *pixelill*, where m is the number of pixels in the ROI. Dividing this into $n \times w \times h$ work-items is problematic because each work-item does not know where in the *pixelpos* and *pixelill* buffers to write when they find a ROI pixel. The solution is to evaluate m on the CPU (negligible computation time), and give this as parameter to n work-items that process a b-scan each. Each thread should generate m elements, so a private counter is sufficient to manage where to write in memory.

The next three steps will process the $n \times m$ elements of *pixelpos* and *pixelill*. The transformation task is divided into $n \times m$ work-items that each perform one matrix multiplication. The task of converting the coordinates to voxel indices and using them to fill the volume has the same number of work-items. For clarity however, these steps are not merged into one kernel. Filling the volume holes, on the other hand, is independent of the size of the input. The massive parallelism of the GPU allows for creating one work-item per voxel, totaling possibly millions, and where each work-item averages the neighbors if the voxel is empty. After this step, the volume is transferred back from the device if so desired. If the volume is visualized while on the GPU, this step might be skipped to save the transfer time.

4.2.3 Optimization techniques

In addition to increased performance from processing elements in parallel, there are further optimizations that can be performed. Actually, a naive porting of sequential code to the GPU does not necessarily utilize the device's capabilities, and might even result in a *slowdown*. To make sure that each processing element in the device is occupied with work, we need to devise suitable work group sizes. As each work group is processed on one compute unit, we also need to ensure that its resources (such as shared memory) are not exhausted. On the CUDA architecture, groups (*warps* in CUDA terminology) of 32 work elements are processed at the same time, so work group size should[1] be a multiple of 32. In order to hide memory latency,

[1]In fact, if not multiple of 32, they will be padded to be so by CUDA.

the total number of work groups should also be such that each compute unit have multiple groups to manage. On the Nvidia Tesla C2050 GPU, there are 16 compute units.

For the task of filling *pixelpos* and *pixelill*, the *NDRange* is only n work-items, which is typically 200-500. To ensure enough work groups, we use a work group size of 32 and pad the *NDRange* to a multiple of this work size. Such padding is generally done by Equation 4.5, where p is the value of what n should be a multiple of. The next steps have a substantially higher *NDRange*, and we use work groups of 512 work-items which is the maximum possible size.

$$n_{padded} = (\lfloor \frac{n}{p} \rfloor + 1) \cdot p \tag{4.5}$$

For further optimizations, the number of variables in the code is manually reduced to lower register usage and small data buffers are put in fast constant memory. The transformation matrices are small enough to fit without modifications, and by compressing the mask it too can fit. As mentioned, the mask uses the same format as the b-scans, with one byte per pixel. As the ROI is a boolean value (either in or out), eight pixels can be encoded into one byte using bitwise operations. This compression is performed on the CPU and also makes the mask faster to transfer to the device's memory.

4.3 Voxel-Based Reconstruction

Performing voxel-based (or ”backwards”) reconstruction is not as straight forward as the pixel-based method. For each voxel, one must find the b-scan that is closest when transformed into the volume. When found, the pixel on this b-scan that will contribute to the voxel must then be located. In this section, the voxel-based reconstruction implemented in this thesis is described. As in the previous section on pixel-based reconstruction, we also cover how this is parallelized on the GPU and some of the important optimization techniques.

4.3.1 Method

The method is based on voxel-nearest-neighbor (VNN) [34]. As with pixel-based reconstruction, the input tracking data must be calibrated and interpolated. After this has been completed, the two approaches start differ. The reconstruction consists of the following steps illustrated in Figure 4.4.

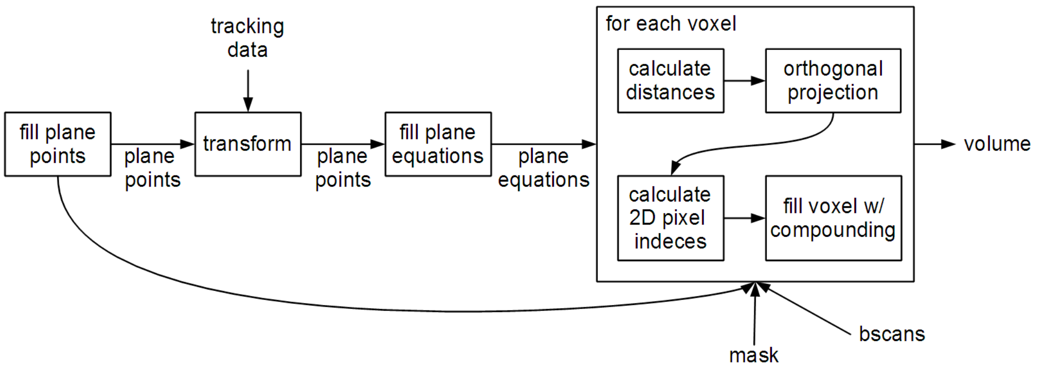


Figure 4.4: Steps performed in voxel-based reconstruction

1. Fill *planepoints*
2. Transform *planepoints*
3. Fill *planeequations* from transformed *planepoints*
4. For each voxel:
   (a) Calculate distance from voxel to planes
   (b) Calculate orthogonal projection of voxel on closest plane
   (c) Calculate 2D coordinates of projection on plane and convert to pixel indices
   (d) Fill voxel using compounding methods

In the first step an array of n triplets of 3D space locations is constructed. The points in the triplet are the top-left, bottom-left and top-right corners of the b-scans in world coordinate space such that all b-scans are parallel and lie flat on the YZ plane. See Equation 4.6. The top-left corner is dubbed $corner0$, the top-right $cornerx$ and the bottom-left $cornery$.

$$corner0 = (0, 0, 0),\ cornerx = (0, w\Delta x, 0),\ cornery = (0, 0, h\Delta y) \tag{4.6}$$

In the second step, these points are transformed according to the tracking data. The transformation matrix of each b-scan is multiplied by each of the three corners, and the resulting coordinates are now correctly placed in world space. Three points is enough to define a plane, but to avoid redundant calculations, an array of n plane equations is constructed in step three. Equations 4.7 and 4.8 show how three b-scan corners are used to calculate the parameters of a plane equation. Each plane equation has four parameters, A, B, C and D, and define a plane implicitly. The plane equation is given in Equation 4.9. The mathematically inclined reader will notice that A, B and C are vector coordinates of the plane normal, and D is the distance from origin to the closest point on the plane.

$$normal = \frac{(corner0 - cornerx) \times (cornery - corner0)}{|(corner0 - cornerx) \times (cornery - corner0)|} \tag{4.7}$$

$$(A, B, C, D) = (normal_x, normal_y, normal_z, -A \cdot normal) \tag{4.8}$$

$$Ax + By + Cz + D = 0 \tag{4.9}$$

The last step is to iterate over all voxels and fill their value from the closest b-scan. It is for this purpose that *planepoints* and *planeequations* were constructed in the previous steps. To fill a voxel, a series of *substeps* are performed (see previous list). Each voxel with indices (x, y, z) has world space coordinates given by $v = (x\Delta v, y\Delta v, z\Delta v)$. In the first substep (a), we calculate the distance of an orthogonal projection from these coordinates to each of the b-scan planes and find the minimum. The brute force approach is to test all planes, but a clever optimization technique has been devised and will be explained under "Optimization Techniques" below. For a point v, the orthogonal distance to a plane is found using Equation 4.10.

$$distance = \frac{(A, B, C) \cdot v + D}{|(A, B, C)|} \tag{4.10}$$

The next substep (b) is then to calculate the world space coordinates p of the voxel coordinates orthogonally projected onto the plane. Given the distance and the plane normal, this is given as Equation 4.11.

$$p = v - distance \cdot (A, B, C) \tag{4.11}$$

To fill the voxel, we need the pixel intensity at this location (p) on the b-scan. This requires calculating the pixel indices in the b-scan image. To do this, substep (c), the vector from the top-left corner ($corner0$) to the point

p is projected onto normalized vectors parallel with the x and y axis of the b-scan. The lengths of the projections are divided by Δx and Δy to give pixel indices (px, py). Equations 4.12 and 4.13 show the calculations.

$$px = \frac{(p - corner0) \cdot |cornerx - corner0|}{\Delta x} \tag{4.12}$$

$$py = \frac{(p - corner0) \cdot |cornery - corner0|}{\Delta y} \tag{4.13}$$

Given the indices of the contributing pixel, the last substep (d) is to fill the voxel with the calculated value.

4.3.2 Parallelization

A code listing of the implemented OpenCL kernel can be found in Appendix F.2. In the previously described method, only step 4 is computationally demanding enough to require parallelization. There are only three points and one equation per input b-scan, and as there are typically around 200-500 such b-scans, the total computation time is negligible compared to iterating over all the voxels. For example, volume size can typically be 256^3 or 512^3. Also, the sizes of the corner points and equation parameters are small enough to be quickly transferred to the device. To summarize, step 1 to 3 is performed on the CPU, and step 4 is parallelized on the GPU.

The voxels are processed concurrently in columns. A volume of $w_{volume} \times h_{volume} \times n_{volume}$ voxels will result in $w_{volume} \times n_{volume}$ threads. Each thread executes a kernel that iterates over h_{volume} voxels, performing substep (a) to (d) on each to fill them. The reason for this separation is that each voxel processed in a column is not more than than Δv from the previously processed voxel, and this property is used in an optimization technique to be described below.

4.3.3 Optimization Techniques

During voxel-based reconstruction, one of the most computationally demanding tasks is to find the b-scan that is closest to a voxel. The brute force approach involves testing all n b-scans for each of the $w_{volume} \times h_{volume} \times n_{volume}$

voxels, a computationally expensive task. However, it is possible to exploit inherent continuity in the b-scans.

Ideally, each b-scan will come "after" the previous one. In other words no b-scan will intersect another, and each b-scan is in front of the previous (in the direction of the probe's trajectory). Noise in the tracking data and an unsteady hand can cause the b-scans to be partially shuffled. Additionally, when rotating the probe, there is a high probability that b-scans will intersect each other. Even though the b-scans may come in any order, the general tendency is along the probe trajectory. In other words, they are *partially ordered*. This means that if the closest b-scan to a given voxel is found, then the index of the closest b-scan to the neighbor voxel is probably close to the index of the previously closest b-scan. To take advantage of this, the following scheme was devised:

- Let each thread process a column of voxels
- i = index of b-scan closest to first voxel in column found by brute force
- For each voxel in column:
    - Loop $j \in 0, 1, 2, ..., n$
        - Calculate distance to b-scans with index $k = i+j$ and $l = i-j$ (if they exist)
        - i = the closest b-scan of i, k and l
        - If difference between distance to i and distance to k or l is above a given cutoff, stop testing any more ks or ls, respectively

The idea is that b-scans before and after the previous b-scan that was closest are tested. It is important to process voxels next to each other and not, for instance, jump from voxel $(w-1, y, z)$ to $(0, y+1, z)$ because they are neighbors in memory. Since the b-scans are partially ordered the assumption is that when testing a b-scan with an index far enough from the last b-scan that was closest, it will not be a better (closer) choice than the b-scans tested so far. The cutoff limit will depend on the nature of the input tracking data. For example, in one of our test cases, a cutoff of $4\Delta v$ was sufficient. After this, the distances found had "peaked", and the search could end.

Chapter 5

Real-Time Incremental Reconstruction and Visualization

In our work described in the previous chapter, all input data was ready and available before the reconstruction took place. This meant that all data could be examined and taken into account while reconstructing. Another advantage was that the whole operation can be performed in "bulk" with small overhead. An alternative is incremental reconstruction where the volume is generated while the b-scans and tracking data are acquired. In this situation, one can only take previously acquired data into account, and a smaller reconstruction procedure is performed for each chunk of data as it is generated by the ultrasound and tracking system.

In this chapter, we present a novel method to incrementally reconstruct tracked ultrasound data in real-time on the GPU. Incremental reconstruction is one of the main contributions of this thesis, and while the methods in the previous chapter were only nearest-neighbor approaches, we can also obtain high-quality reconstruction using interpolation techniques from [5] and [37]. The chapter starts with how to handle different rates of b-scan and tracking data when acquired incrementally, we describe a simple pixel-nearest-neighbor scheme performed incrementally on the GPU, which is then followed by a description of the high-quality incremental reconstruction. Lastly, we describe how the volume can simultaneously be visualized while it is reconstructed.

5.1 Incremental Acquisition and Preprocessing of Input data

When reconstructing incrementally, a method is needed for acquiring the b-scans from the ultrasound device (typically from a frame-grabber card) and the tracking data from the tracking system. This thesis does not focus on what goes on behind the scenes before this data is available in the computer's memory, but we define a simple interface that is assumed can be implemented by the ultrasound and tracking systems:

Call	Effect
`get_last_b-scan(timetag, b-scan)`	returns most recent b-scan and associated timetag
`get_last_tracking(timetag, matrix)`	returns most recent tracking data in form of transformation matrix and associated timetag
`poll_b-scan` / `poll_tracking`	returns true (once) when new b-scan/tracking data is ready, and false at each call after that until another b-scan/tracking data is ready
`end_of_data`	returns true if there are no more b-scans *or* no more tracking data to be expected (i.e. system turned off), false otherwise

Table 5.1: Interface against ultrasound and tracking system

Given this interface, the following algorithm is used for acquiring and processing the input data incrementally:

```
• loop:
    1. b-scan = NULL
    2. matrix = NULL
    3. while (matrix or b-scan is NULL)
        (a) if (poll_b-scan) get_last_b-scan(timetag_b, b-scan)
        (b) if (poll_tracking) get_last_tracking(timetag_m, matrix)
    4. if (end_of_data) break loop
```

5. interpolate tracking data and calibrate
6. perform reconstruction increment

The reason for doing it in this fashion is that the rate of b-scan generation is not the same as the rate of incoming tracking data. After the while-loop in the algorithm has been completed, there has been acquired a b-scan and tracking data, but the tracking data is either from before or after the b-scan. As in the non-incremental method, we choose to interpolate the transformation matrix according to the given timetags. This, together with the calibration, make up the preprocessing of the input data. After this step, one increment of the reconstruction can be performed.

5.2 Incremental PNN Reconstruction

The pixel-nearest-neighbor (PNN) method is suitable for incremental reconstruction as each incoming b-scan can be processed and put into the volume. Since pixel-based methods work forward, only the voxels affected by a b-scan are processed. In contrast, voxel-based methods work backwards, and all voxels must be processed for each increment of the reconstruction.

In this section, we describe how to perform incremental PNN reconstruction utilizing the GPU for parallel processing. In the incremental algorithm, not all steps benefit from being offloaded to the GPU, and the distribution between CPU and GPU is covered here. Although PNN is suitable for incremental reconstruction, the hole-filling stage introduces difficulties when trying to maintain high performance. This will also be explained, and some possible solutions are presented.

5.2.1 Method and Parallelization

The method used for incremental PNN is similar to ordinary non-incremental PNN. This is why PNN was chosen for a simple incremental reconstruction method. For each increment, we assume that a b-scan and its interpolated and calibrated tracking transformation matrix has been acquired as described in the previous section. The steps are then as follows:

- For each acquired b-scan and associated tracking data:
    1. Fill *pixelill* and *pixelpos* (on CPU)

2. Transfer b-scan, tracking data, *pixelill* and *pixelpos* to GPU
3. Transform *pixelpos* (on GPU)
4. Convert *pixelpos* to voxel indices (on GPU)
5. Fill device memory volume (on GPU)
6. Transfer *pixelpos* to CPU
7. Fill CPU memory volume (on CPU)

The main steps are the same as in the non-incremental version: fill *pixelill* and *pixelpos*, transform, convert to voxel indices and fill the volume. However, in the incremental version some steps previously done on the GPU are now performed on the CPU, and in addition we maintain a duplicate volume on the CPU in addition to the GPU. The reason behind these differences will be explained below.

In step 1, the mask is used to fill *pixelill* with pixel intensities and *pixelpos* with untransformed coordinates, just like with non-incremental PNN, but this step is now performed on the CPU. The reason for this is that while it is straightforward to divide the work into one b-scan per thread, it is harder to divide the work of a single b-scan. As mentioned in Section 4.2, parallelizing the sequential code requires each thread to know where in the buffer to write the data if a pixel in the ROI is found. This could be performed with an atomic update of a global variable identifying the next position to write to, but the overhead of maintaining and waiting for this variable overshadows the benefits of parallelizing the task. This operation is fairly trivial, and takes negligible time when only processing a single b-scan. So although an atomic counter or other scheme for parallelizing this might achieve the same effect, the simplest solution is to perform this sequentially on the CPU and then transfer the results (of negligible size for single b-scan) to the device.

In contrast to step 1, transforming the coordinates in step 3 is a computationally heavy task and is best suited for the GPU, and the implementation is straight forward with m threads (one for each pixel in the ROI). A typical mask has around 100 000 pixels, which is more than enough to occupy the compute units on a modern GPU. Again the same approach is used in step 4 where we parallelize using one thread per pixel in the ROI. As the data is already on the device memory, it is logical to perform this step on the GPU.

In non-incremental reconstruction, if the volume is not solely used for visualization on the GPU, it is transferred from device to host when the reconstruction is complete. To do the same incrementally would be a performance bottleneck as tens of megabytes of data would be transferred for each increment of the reconstruction. To reduce the bandwidth used, we transfer only the processed *pixelpos* from device to host, and use the *pixelill* (from step 1) that already exists on the host to fill a volume duplicate on CPU memory. This incrementally updated volume can be used in any third-party application. If the volume is only used for visualization on the GPU, then the two last steps may be omitted. It is also possible to visualize the volume on the GPU during reconstruction, and then transfer the whole volume to the CPU only at the end of the reconstruction session.

5.2.2 Incremental Pixel-Based Hole Filling

It should be highlighted that hole filling is omitted in the method presented in the last section. In the non-incremental version, hole-filling can be performed after all b-scans have been inserted into the volume. This is a computationally heavy task if performed for each increment for the entire volume. In addition, transferring the entire volume with filled holes is unacceptable if real-time performance should be maintained. Thus, there are two problems to be solved: the first is how to find and fill holes incrementally without processing the entire volume each increment, and the second is how to incrementally maintain a reconstructed volume on the host without transferring the entire volume each time. We present possible schemes to solve this.

The first method is to use some sort of ”splatting” technique where each pixel contributes to voxels in a ”splat” around its nearest neighbor. The splats can be spherical in nature and decrease in intensity as distance to the center increases. If sphere diameter is greater or equal to the maximum separation between two b-scans, no holes will occur. To implement this in the GPU-based PNN reconstruction, the splats are filled on the CPU and GPU volumes in step 5 and 7. The disadvantages are increased computation time (on CPU and GPU) spent on filling splats and blurring of the output volume.

Another approach is to use the parallel processing power of the GPU to fill holes on device memory volume, but not to transfer any additional hole-filling information to the host. The GPU volume will then have filled holes while the CPU volume will not, and it (the GPU volume) can be used for intermediate visualization where holes are not critical. At the end of the session, the entire hole-free volume can be transferred to host. Although bandwidth is saved in this approach, full hole filling is time consuming when performed on each increment, even on the GPU.

It should not be necessary to search for holes in the entire volume for each increment. Holes may occur when two b-scans are more than Δv from each other when inserted in the volume, and no other b-scans occur between them. This can be exploited by searching for holes in an area close to each inserted b-scan. Assuming no holes wider than q voxels, we can search each inserted b-scan for holes in the $q \times m$ voxels immediately next to the voxels filled by the b-scan. In this way, a much smaller subset of the volume is searched, and we always know the upper limit of how many holes are filled on each increment. The $q \times m$ voxels intensities can be transferred to host in addition to *pixelpos*. The locations of the voxels where holes are filled can be set to the opposite direction of the current probe trajectory, and can thus be calculated from *pixelpos* and the tracking data without transferring more coordinates to the host.

5.3 High-Quality Incremental Reconstruction

While incremental reconstruction with PNN is fast, it has its shortcomings. Due to the simplicity of the algorithm, the quality of the reconstructed volume is not as good as can be obtained with voxel-based methods using interpolation. Furthermore, hole filling is problematic when aiming for real-time performance. What is desired is a method that have high hole-free reconstruction quality, that can preferably be adjusted for a quality-time tradeoff, and one that is suitable for incremental updates of the volume without processing all voxels.

Pixel-based methods have the advantage of processing only the voxels that is actually updated by each b-scan, but the simplest variants can have the problem of holes. More advanced voxel-based methods can offer high reconstruction quality without holes, but are unsuitable for incremental updates. The method presented in this section combines the advantages from both approaches. The steps taken will be explained, together with issues from implementation and parallelization of the method are also covered.

5.3.1 Method

The essence of the method is to transform incoming b-scan planes into the volume, but also to only process voxels located *between* b-scans. For the voxels processed, voxel-based interpolation methods can be used to fill their values. This is what is meant by combining the pixel- and voxel-based approaches (or rather, the forward and backward approaches). B-scans are processed forwardly into the volume, but instead of filling the volume directly from the pixels, voxels between inserted b-scans have their values interpolated from the surrounding b-scans.

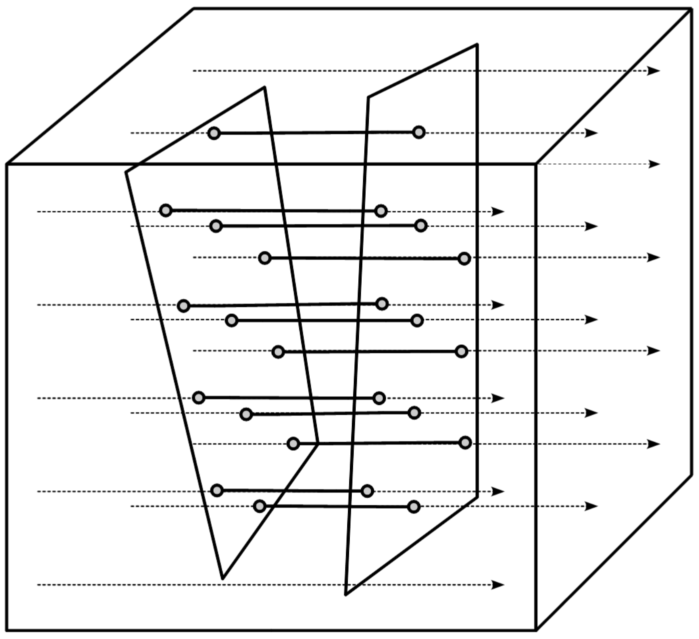


Figure 5.1: Finding voxels between b-scans

For each incoming b-scan, we transform its corner points using the tracking data and construct its plane equation as previously described in Section 4.3. Figure 5.1 show how the voxels between the current and previous b-scans are calculated. First, rays are constructed along columns of voxels, starting at the edge of the volume with one ray per column. These rays are used for ray-plane intersection calculations with the current and previous b-scan planes. The distance t along a ray starting at r_0 with direction r_d to a plane with equation parameters (a, b, c, d) is given by Equation 5.1, and the voxel indices (x, y, z) are given by Equation 5.2. Each ray gives one intersection point with the current b-scan, and another with the previous b-scan. The voxels between these will lie next to each other.

$$t = -\frac{(a, b, c) \cdot r_0 + d}{(a, b, c) \cdot r_d} \tag{5.1}$$

$$(x, y, z) = \frac{r_0 + tr_d}{\Delta v} \tag{5.2}$$

After the voxels between the b-scans have been found, they are each filled using a voxel based method. In our system, one can choose between two methods: one based on distance weighted orthogonal projections, and one based on cubic interpolation of the probe trajectory.

Distance Weighted Orthogonal Projections

This method builds on the approach described by Trobaugh *et al.* [37], where each voxel filled is orthogonally projected onto nearby b-scans and interpolated using the distance to the b-scan planes. But while Trobaugh *et al.* project onto only two surrounding b-scans, we project onto the $n_w \leq n$ b-scans surrounding the voxel. The number of b-scans (n_w) taken into account for each voxel can be adjusted for a tradeoff between quality and performance. The 2D coordinates of the projected point on the b-scan plane can be found using equations previously described in Section 4.3. What differs, however, is that we project not only to the closest b-scan, but to all nearby b-scans. The number of b-scans used for each voxel increases the quality. For each projection, the pixel intensity is given by a bilinear interpolation of the 4 pixels closest to the projected coordinates, as shown in Figure 5.2 and given by Equation 5.3 where x_f and y_f is a fraction between 0.0 and 1.0 that says how far the projected point is from the closest pixel coordinates to the top-left. E.g. 0.5 means halfway between (x, y) and $(x+1, y+1)$ (see figure). If one or more of the pixels are outside the ROI, the bilinear interpolated intensity is not used. Each intensity used is weighted by the inverse of the distance to the b-scan plane given by Equation 4.10. To normalize the result, it is divided by the sum of the weights as given by Equation 5.4.

$$\begin{aligned} bilinear &= bscan(x,y)(1-x_f)(1-y_f) + \\ &\quad bscan(x+1,y)x_f(1-y_f) + \\ &\quad bscan(x,y+1)(1-x_f)y_f + \\ &\quad bscan(x+1,y+1)x_f y_f \end{aligned} \tag{5.3}$$

$$value_{dwop} = \frac{\sum_i (bilinear_i \cdot weight_i)}{\sum_i weight_i} \tag{5.4}$$

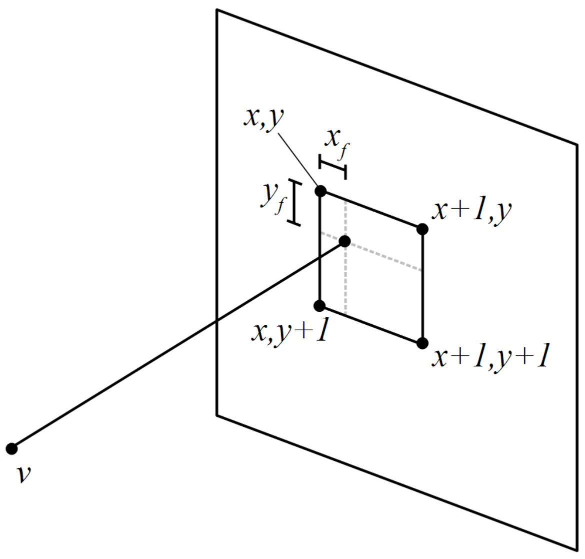


Figure 5.2: Bilinear interpolation

Cubic Interpolation of Probe Trajectory

This method is based on reconstruction as described by Coupe *et al.* [5], but performed only on the voxels between two b-scans. The principle is to perform cubic interpolation of the tracking data as an estimation of the trajectory of the ultrasound probe. First, the timetags of two adjacent b-scans are linearly interpolated based on the orthogonal distance from the voxel coordinates in space to the b-scan planes, as illustrated in Figure 5.3 and given in Equation 5.5. This new timetag is called a *virtual timetag* and belongs to a virtual plane going through the voxel.

$$t = \frac{d_2 t_1}{d_1 + d_2} + \frac{d_1 t_2}{d_1 + d_2} \tag{5.5}$$

To find the plane equation, top-left corner and x and y-vectors of the b-scan in the virtual plane, we use cubic interpolation based on the timetags through the *key function* given in Equation 5.6 with $a = -\frac{1}{2}$. We interpolate plane equation parameters, corner and vector coordinates from four adjacent b-scans, where the two internal b-scans are those previously used to interpolate the virtual timetag. The cubic interpolation is shown in Figure 5.4. For each parameter or coordinate α to interpolate, the value of the key function is used in Equation 5.7.

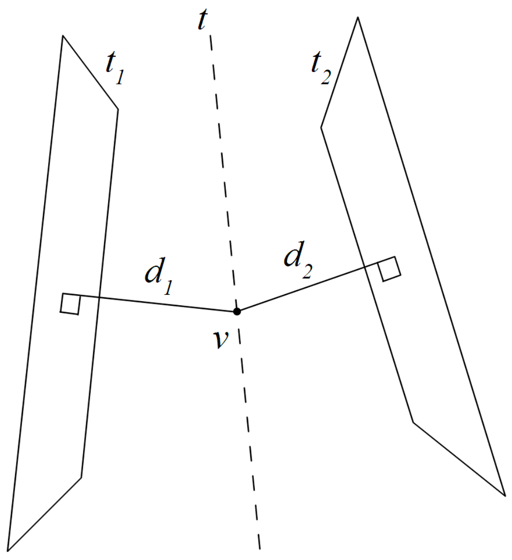


Figure 5.3: Timetag of virtual plane

$$\phi(\beta) = \begin{cases} (a+2)\beta^3 - (a+3)t^2 + 1 & \text{if } 0 \leq \beta < 1 \\ a\beta^3 - 5a\beta^2 + 8a\beta - 4a & \text{if } 1 \leq \beta < 2 \\ 0 & \text{if } 2 \leq \beta \end{cases} \tag{5.6}$$

$$\alpha = \sum_{i=0}^{4} \alpha_i \phi(\left|\frac{t - t_i}{t_1 - t_0}\right|) \tag{5.7}$$

When the virtual plane has been obtained, the 2D coordinates of the voxel (lying on the plane) can be found using the equations described in Section 4.3. These coordinates are then used on each of the four adjacent b-scans, and the pixels at those locations are bilinearly interpolated using Equation 5.3. The four bilinearly interpolated values are then weighted by the inverse orthogonal distance to each b-scan from the voxel coordinates according to Equation 5.4.

5.3.2 Implementation and Parallelization

Code listings of the implemented OpenCL kernels can be found in Appendix F.4. To be able to interpolate between several b-scans, one solution is to queue up a buffer of incoming b-scans and tracking data. As incoming data is pushed into the queue, data is popped from the other end of the queue.

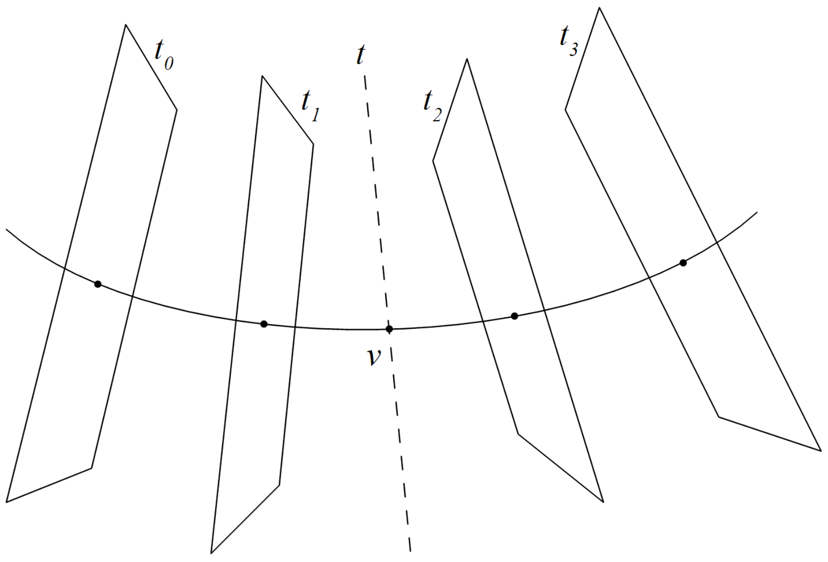


Figure 5.4: Cubic interpolation of four b-scans

When the session begins, the queue is simply filled without processing any of the input. During the session, this queue will be a "sliding window" across the stream of incoming data. The number of b-scans (each with associated tracking data) in the window is at least four if the probe trajectory (PT) method is used to fill the voxels, and a number $n_w \leq n$ if distance weighted orthogonal projections (DWOP) is used. There are several options for which two b-scan planes to use in the window when finding voxels to fill between them. To avoid repeated fillings of the same voxels, we use the planes of the two b-scans in the center of the window.

When constructing the rays for the ray-plane intersection calculations, any axis can be used as a direction for the rays. As we parallelize with one thread per ray, we want to distribute work as evenly as possible. This means choosing an axis that is the most orthogonal to the normal for each b-scan. And by this we mean the axis that has the lowest angle between itself and the normalized b-scan plane normal **n** as given in Equation 5.8. The axis used is re-evaluated for each incoming b-scan.

$$S = \{(0,0,1),(0,1,0),(1,0,0)\}, \quad ARGMIN_{\mathbf{a}\in S}(\mathbf{a}\cdot\mathbf{n}) \tag{5.8}$$

To handle discontinuities in the input stream (e.g. from pauses in tracking during one session), a check is made for each incoming b-scan. If the increase in time according to the timetags is above a cutoff value, the reconstruction is restarted from that b-scan instead of attempting to interpolate across the discontinuity. Ultrasound systems usually have a regular acquisition rate f, and so a cutoff at $\frac{2}{f}$ is sufficient.

As mentioned, the computations are parallelized with one thread per ray, and this *NDRange* is kept as the voxels found by one ray will also be processed by the same thread. Assuming that the z-axis is always used as the best axis, the *NDRange* will be $w \times h$ threads, each processing a number of voxels between 1 and the highest separation between two adjacent b-scans (typically ≤ 5).

5.4 Simultaneous Reconstruction and Visualization

One of the purposes of doing incremental reconstruction is to be able to see the volume as it is constructed. This allows for immediate feedback during scanning. There are many ways to visualize a volume, and the system described in this thesis offers orthogonal multiplanar reformatting (MPR) slices or volume ray casting. In addition, as the volume is accessible on both device and host memory, it can be used as input for third-party visualization packages. In this section, the visualization techniques used are described.

5.4.1 Orthogonal MPR slices

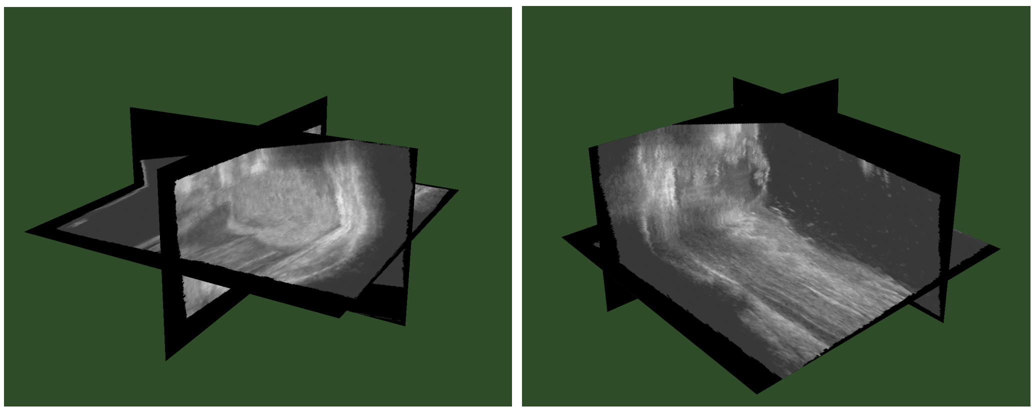


Figure 5.5: Screenshots of orthogonal MPR slices generated by our implementation

Figure 5.5 shows some screenshots of the orthogonal MPR slices, and larger figures can be found in Appendix B. The slices are along each of the three volume axis, and the voxel values in each slice are rendered on the slice plane. This is performed by extracting voxel values on indices (x, y, i), (x, i, z) and (i, y, z) for the z, y and x axis, respectively. The parameter i determines where on each axis the slices are located, while x, y and z are all the possible voxel indices in the volume. The slices are extracted from the host memory volume and used as textures for three orthogonal polygons rendered with OpenGL. It should be noted that extracting the voxels and using them as textures takes a negligible amount of time. Using the mouse, the user can rotate the volume to view the slices at different angles, and pressing keyboard keys will increase or decrease the i parameter. In this way, all parts of the volume can be visualized.

5.4.2 Volume Ray Casting

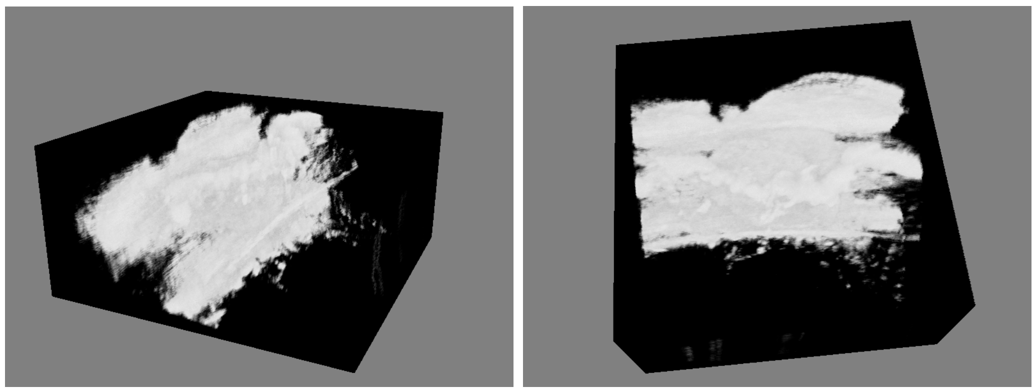


Figure 5.6: Screenshots of volume ray casting generated by our implementation

Figure 5.6 shows some screenshots of the ray casted volume, and larger figures can be found in Appendix B. Ray casting is a computationally demanding task, but the GPU's processing power is utilized while the volume data is still on the device memory. The task is parallelized with one thread per ray, and consists of two steps: first construct the rays, then cast them into the volume.

Code listings of the implemented OpenCL kernels can be found in Appendix F.3. The resulting rendered image consists of $w_s \times h_s$ pixels, with one ray (and thread) per pixel. The camera is defined by a camera location in space, a *lookat* location that the camera is looking at, a vector defining "up" in the rendered image, and a vector defining "flat" (the horizontal direction) in the rendered image. Each ray's origin is the given camera location, but their direction needs to be calculated from their pixel position and a desired field-of-view.

When the ray directions have been found, they can be cast into the volume. Instead of stepping from the ray origin, we save computations by first calculating the intersection between the box-shaped volume and each ray. This saves sampling for steps that are outside the volume.

There are many ways to accumulate voxel intensities while stepping along a ray. The method used in this work models the voxels as transparent cubes with a transparency level between 0.0 (fully oblique) and 1.0 (invisible). The entire procedure is as follows:

1. $strength = 255$
2. for each step along ray:
    (a) if $strength <$ cutoff, then break stepping
    (b) $transparency = 1 - \frac{intensity_{voxel}}{255}$
    (c) accumulate $strength \cdot (1 - transparency)$ into ray's pixel intensity
    (d) $strength \leftarrow strength \cdot transparency$

Each ray starts with a *strength* parameter representing how much light has been absorbed. The initial value is the maximum value of a pixel (255), and it is reduced for each voxel encountered. For each voxel sampled, its intensity is weighted by the current strength and accumulated into the final pixel intensity. To save computations, the stepping is ended if the strength is below a given cutoff where the voxels simply do not contribute noticeable intensities.

Chapter 6

Analysis and Discussion of our Results

In this chapter, our results obtained using the previously described methods are presented. The methods and their results are also analysed and discussed. The results include not only the quantitative performance, but also the visual quality of the reconstructed volume. The chapter ends with a general discussion of the pixel-based and voxel-based approaches, and of incremental versus non-incremental reconstruction.

6.1 Performance

The goal of this thesis is to obtain real-time performance for 3D ultrasound reconstruction. Here, we present the measured performance of the methods described in the two previous chapters, and also identify how much of the computation time is spent on the different algorithmic steps on the device. We will first start by describing the test setup used in the measurements, then we present and discuss the performance obtained on different types of computing hardware (CPU and GPU). Last, the distribution of computation time between the different steps of the reconstruction procedure as well as the amount of memory required is presented and discussed.

6.1.1 Test Setup

Table 6.1 shows the hardware specifications of the computer used to measure the results in this chapter, with the different GPUs being used one at the time. All hardware is commercially available as commodity products.

GPU$_1$	Nvidia Tesla C2050
GPU$_2$	AMD ATI Radeon HD5870
GPU$_3$	Nvidia Quadro FX5800
CPU	Intel Core 2 Quad Q9550
Memory	4 x 2 GB DDR3
OS	Microsoft Windows XP 64 bit
Compiler	Microsoft Visual Studio 9.0

Table 6.1: Test computer specifications. See Table 3.1 for GPU details

Table 6.2 gives a specification of the test input and volume size that was used for the performance tests. In parentheses are the symbols that from now on will be used to refer to these values. The tracking data were obtained from a Polaris optical tracking system (Northern Digital, Waterloo, Canada), and the b-scans from a Vivid 7 ultrasound scanner (GE Vingmed Ultrasound, Horten, Norway), equipped with a GE M12L linear array probe (GE Healthcare, Waukesha, WI).

Number of b-scans (n)	434
Number of tracking data	520
B-scan size ($w \times h$)	768 x 576 pixels
Region-of-interest size (m)	342 x 356 pixels
Volume size ($w_{volume} \times h_{volume} \times n_{volume}$)	512 x 256 x 512 voxels
Pixel spacing x-dimension (Δx)	0.107724 mm
Pixel spacing y-dimension (Δy)	0.111732 mm
Voxel spacing all dimensions (Δv)	0.08 mm
Bits per pixel/voxel	8

Table 6.2: Test input data

Most reconstruction algorithms have one or more parameters that increase the reconstruction quality at the cost of increased computational complexity. In Table 6.3 a list of important parameter settings is provided for the algorithms that were considered.

PNN	5^3 neighbor voxels used for hole filling
VNN	$5\Delta v$ distance cutoff to b-scans
Incremental PNN	no hole filling
Incremental DWOP$_4$	4 b-scans interpolated per voxel
Incremental DWOP$_8$	8 b-scans interpolated per voxel
Incremental PT	cubic interpolation of 4 b-scans per voxel
Ray casting	800 x 600 pixels (i.e. rays)

Table 6.3: Test algorithm parameters

6.1.2 GPU Performance

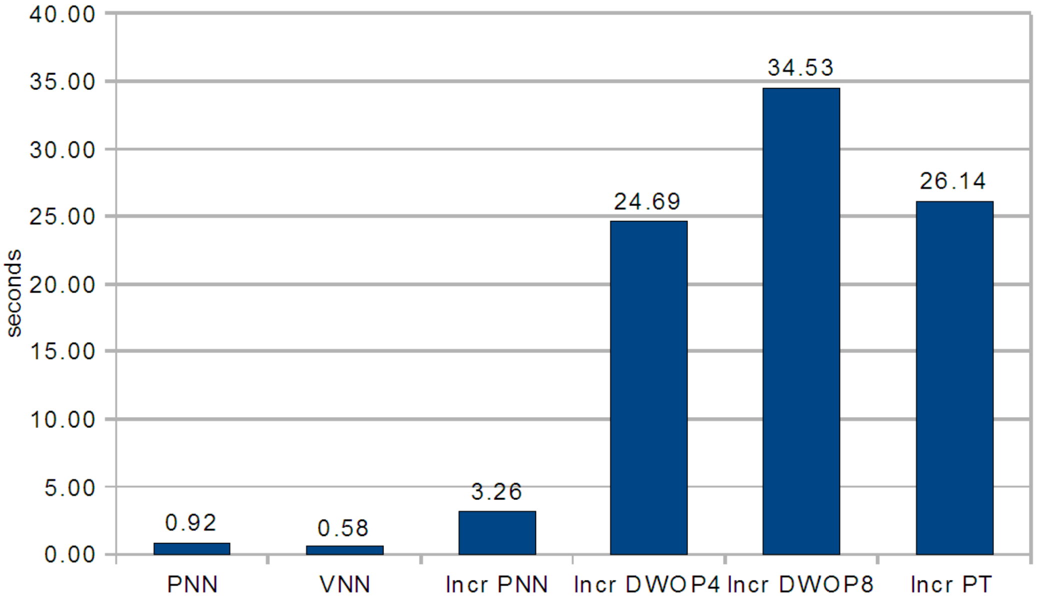


Figure 6.1: Reconstruction times using Nvidia Tesla C2050

Figure 6.1 shows the time measured when reconstructing the test input using the Nvidia Tesla C2050 for various algorithms. PNN and VNN are the non-incremental methods, and as one can see, they complete in under one second. The incremental PNN is slower, even without the hole filling, with the reason for this being the overhead associated with incremental reconstruction as previously described. The other incremental methods are substantially slower, and this is due to increased computations resulting in much higher reconstruction quality than the simple PNN. The incremental performance is, however, *real-time* as about 20-30 seconds are typically used for freehand scanning.

The volume size is purposely set at the upper end of what one requires for normal use. Typically, the voxel spacing is set to the same value as the pixel spacing, so given a pixel spacing at around 0.1 mm the volume size would be $450 \times 200 \times 450$ voxels, which is 40 % less than the one used for testing. The point to note is that these test sizes are *worst case*.

There are two variants of the distance weighted orthogonal projections (DWOP) method examined, which take into account 4 and 8 of the closest b-scans respectively. As can be seen, the increased quality obtained increases the reconstruction time by approximately 40 %. The $DWOP_4$ and the method based on the probe trajectory (PT) take roughly the same time which can be explained by the fact that they both take 4 b-scans into account for each voxel evaluation. The PT method, however, is somewhat slower, which is explained by the higher degree of interpolation used. The complexity of the PT method is reflected by more complex code, which results in generally higher register use compared to the DWOP (49 *vs.* 33 registers).

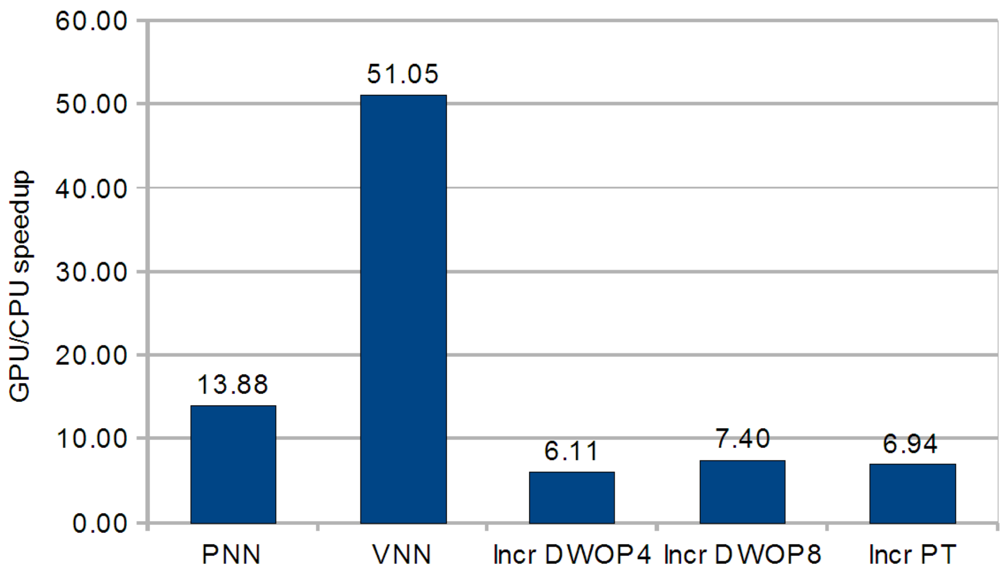


Figure 6.2: Speedup of Intel Q9550 vs Nvidia Tesla C2050

The reconstruction methods were also implemented using only a single-threaded sequential CPU for the computations. In all cases the CPU was slower than the GPU version. Figure 6.2 shows the speedup of utilizing the GPU compared to only the CPU, with the performance of the GPU versions between 6 and 14 times faster, and 51 times faster in the case of VNN. In all cases, this is a substantial speedup. The main limiting factor in the speedup of the incremental cases is in the transfer of the volume at each increment (the specifics will be discussed in Section 6.1.4). One can see that the speedups of DWOP and PT are in the same order as their respective computation times. The explanation behind this is that increased computational load means that the processing power of the GPU can be utilized even more, and that the host-device transfers are smaller in comparison to the computations.

While the non-incremental PNN gains a 14 times speedup, the non-incremental VNN gains a significant 51 times increase in performance. Note that these two algorithms are inherently different, and that they therefore cannot be expected to behave similarly to each other when utilizing the massively parallel GPU. One reason is the determinism of the implementations. Each of the threads in VNN reconstruction does exactly the same work, with a small difference at the end of the computations (based on the voxel being in the ROI or not). For PNN this difference occurs early, as the pixels are processed in the beginning. Such branching results in low occupancy on the GPU, and makes it difficult to achieve good performance. As explained in Section 4.2, one of the major steps of the PNN algorithm is partitioned among "only" 434 threads, one per b-scan. While the VNN can use $w_{volume} \times h_{volume} \times n_{volume} = 512 \times 256 \times 512$ threads, i.e. one per voxel, which is a scalability more suitable to the massively parallel GPU. Furthermore, the hole-filling needed in PNN is a task similar in complexity to the VNN, and thus PNN in total should be slower than VNN. For more details on the difference between VNN and PNN performance, see Section 6.1.4.

When using the Nvidia Tesla C2050, **100 renders** of the $512 \times 256 \times 512$ volume were ray casted in **2.81 seconds**, which is equivalent to **35.6 frames per second**. There are many definitions of how many frames per seconds are required to be considered *real-time*, but a lower limit of 25 (PAL television format) or 30 (NTSC television format) should be sufficient. Given a lower limit of 1 update per reconstruction increment, 434 b-scans over 20 seconds would require a frame rate of 21.7. Thus, the performance is more than sufficient for real-time display.

6.1.3 Performance of GPU Platforms

It should now be established that reconstruction performance is substantially greater using the GPU. However, several GPU platforms are available for comparison. The C2050 is at the time of writing the latest and most powerful generation on the market. By using the multiplatform OpenCL standard [16], it is possible to execute the code on both Nvidia and AMD GPUs. Figure 6.3 shows the performance obtained on the C2050, the older FX5800 also

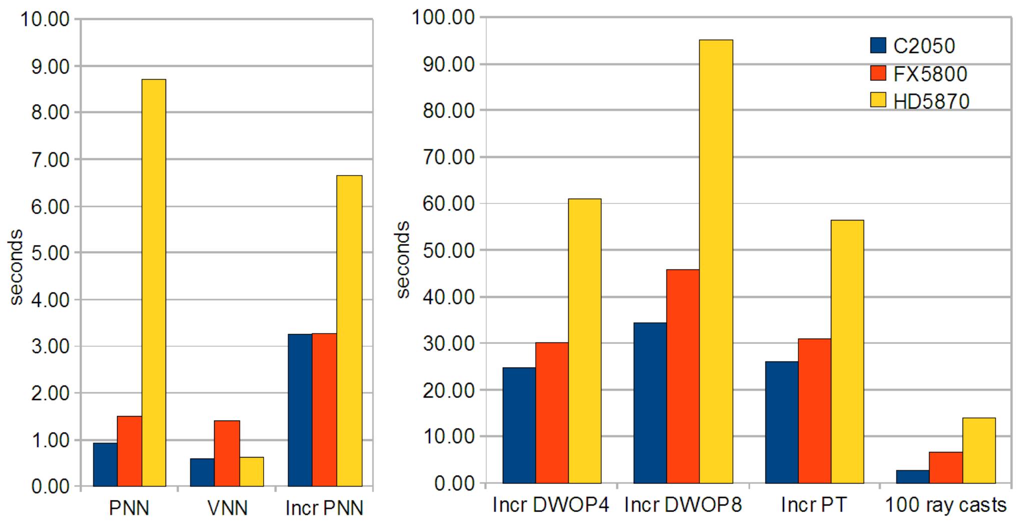


Figure 6.3: Performance of reconstruction and ray casting on Nvidia Tesla C2050, AMD HD5870 and Nvidia Quadro FX5800

from Nvidia, and the HD5870 from AMD. In terms of prior expectations, the FX5800 is one of the most powerful of the *previous* generation of Nvidia GPUs, and the HD5870 is expected to place itself roughly halfway between these two cards.

With these presumptions, the performance results both confirm and surprise. The C2050 performs better than the FX5800 in all cases, with up to 2.5 times the performance. While in some cases (like incremental PNN) the performance is almost equal or only slightly better, but this will be explained below. The HD5870 however, disappoints in all except in the one case of VNN, with performance typically around 50 % of the FX5800 and as low as 17 % in the case of PNN. Just as VNN proved to be very suitable for the GPU with a speedup of 51, it also seems to be the case on the HD5870 and compares favorably with a performance close to the much newer C2050. In the other cases, it can be pointed out that the OpenCL standard was made with CUDA in mind because of its status as a *de facto* standard at the time. AMD's OpenCL implementation obviously has room for improvement, and the potential is clearly there as seen with the performance of VNN. It should be noted however, that in all the cases the HD5870 is still between 1.5 and 3.2 times faster than the pure CPU implementation (and 47 times faster for VNN).

As mentioned, the C2050 beats the older FX5800 in all the cases, but one notices that the gap is substantially narrower for the incremental methods.

The explanation in these cases is due to the time spent on transferring the volume. Even though C2050 represents the next generation of Nvidia GPUs, the data bus between host and device (PCI Express) is identical for both. This is evident in the cases of incremental reconstruction, where the computations are lighter for each increment. Again, this is confirmed by a greater relative gap (1.23 *vs.* 1.33) between $DWOP_4$ and $DWOP_8$, where the only difference is computational complexity.

In the case of VNN and ray casting, the C2050 offers 2.5 and 2.4 times the performance of the FX5800, but the C2050 has only about twice the number of computational cores of the FX5800 and 40 % increased memory bandwidth [27, 24]. The explanation is one key feature to the Fermi GPU architecture which C2050 belongs to, the addition of L1 and L2 cache. With a L1 cache per streaming multiprocessor and a global L2 cache for all computation cores, the effective memory bandwidth can be drastically increased depending on the application. If the same data is read repeatedly by several threads, the values will be available in this fast cache. This is especially the case for VNN, where the b-scan plane equations and corner points are read repeatedly by neighboring threads, and also for ray casting where neighboring rays read the same voxel data. In both cases it is hard to estimate exactly which groups of threads will require what groups of common data, but the cache handles this temporal locality automatically.

6.1.4 Distribution of Reconstruction Time

To gain a deeper understanding of the reconstruction performance for each method, one has to look at the time is takes to do the various reconstruction steps and the data transfers between device and host. In this section we present this distribution, and explain why it occurs like it does. In all cases, the C2050 has been used for the measurements.

Figure 6.4 shows the distribution of the time for PNN on the GPU. Note that transfers between host and device (memcpyHtoDasync and memcpyD-toHasync) are negligible compared to the computations. In this case, around 200 MB (mainly from b-scans) is transferred to the device and 67 MB (the volume) is transferred back. The GPU is connected to the host via a PCI Express 2.x 16X bus, which has an upper transfer limit of 8 GB/s. These small data sizes are thus negligible. The largest portion is the task of filling *pixelill* and *pixelpos*, and as previously discussed, this is hard to parallelize, and is indicated by a GPU core occupancy of only 17 % during this step. The rest of the steps all have a 100 % occupancy. However, there are two

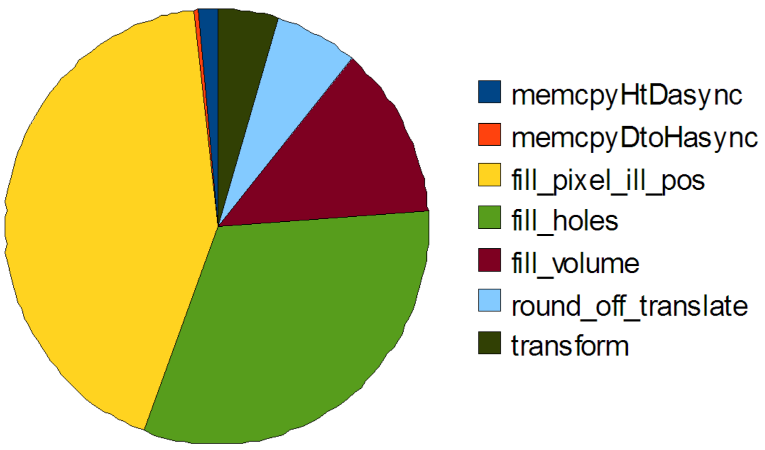


Figure 6.4: Distribution of computation time on Nvidia Tesla C2050 for PNN

other tasks that also take up a large part of the time: filling the volume and (especially) filling the holes. Actually, processing all the ROI-pixels and inserting them into the volume actually takes less time than filling those holes. This demonstrates how the PNN performance is penalized by the hole filling required due to of the nature of the algorithm.

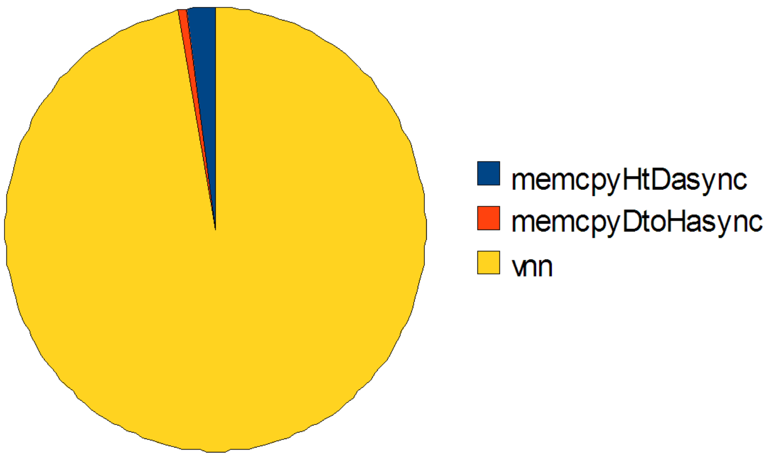


Figure 6.5: Distribution of computation time on Nvidia Tesla C2050 for VNN

Figure 6.5 shows the distribution of time for VNN on the GPU. As in the case with PNN, the transfers between host and device are negligible (in fact, they are of the same absolute sizes as in PNN). Unfortunately, the VNN reconstruction was implemented as a single GPU kernel, and thus the distribution between steps cannot be identified.

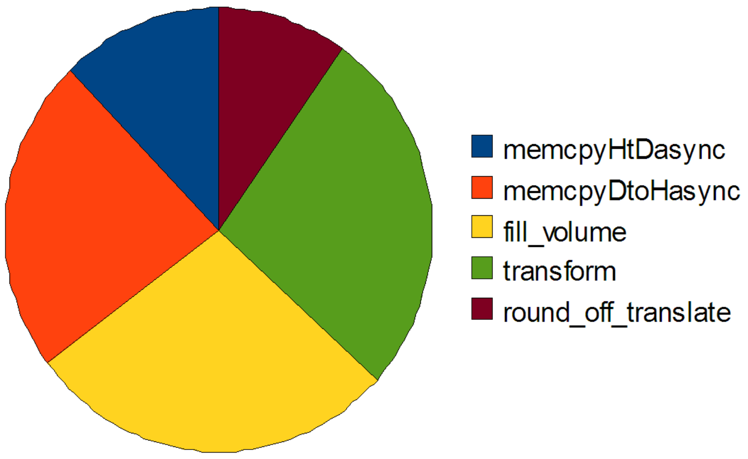


Figure 6.6: Distribution of computation time on Nvidia Tesla C2050 for incremental PNN

Figure 6.6 shows the distribution of time for incremental PNN on the GPU. As opposed to the non-incremental cases, transfers between host and device take a noticeable share as more than a third of the time is now consumed on this. For each increment, a tracking matrix and the *pixelpos* and *pixelill* for one b-scan must be transferred, approximately 1.6 MB. The transfer back of the transformed *pixelpos* is approximately 1.4 MB. However, the overhead associated with each transfer adds up, as demonstrated by the large share of the total time. This overhead is especially noticeable for incremental PNN, due to the total time spent per increment being relatively smaller.

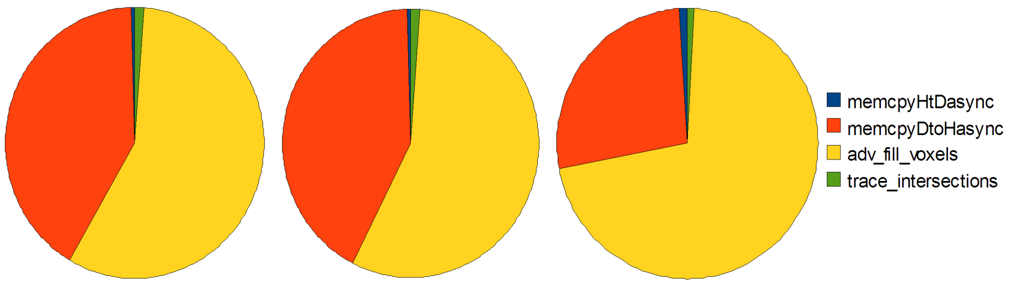


Figure 6.7: Distribution of computation time on Nvidia Tesla C2050 for PT, $DWOP_4$ and $DWOP_8$

The other incremental methods share the same characteristics in the time distribution, as shown in Figure 6.7. The transfers between device and host take up a big share, but this is only *from* the device *to* the host, as host to device is practically negligible. The reason for this is that as each increment takes a substantially longer time in these methods than in incremental PNN, and the overhead of the small host-to-device transfer is hidden, while transferring the 67 MB volume for each increment still takes its share of the total time. The same reasoning explains why $DWOP_8$ has a smaller share of transfers than $DWOP_4$; i.e. each increment is more computationally complex. In fact, incremental PT and $DWOP_4$ has almost identical time distributions, which is not surprising as their computational complexity is very similar. As can be seen in the charts, the task of finding the voxels between two b-scans (trace_intersections) is negligible, rather is the filling of those voxels by PT or DWOP that take up all the computation time.

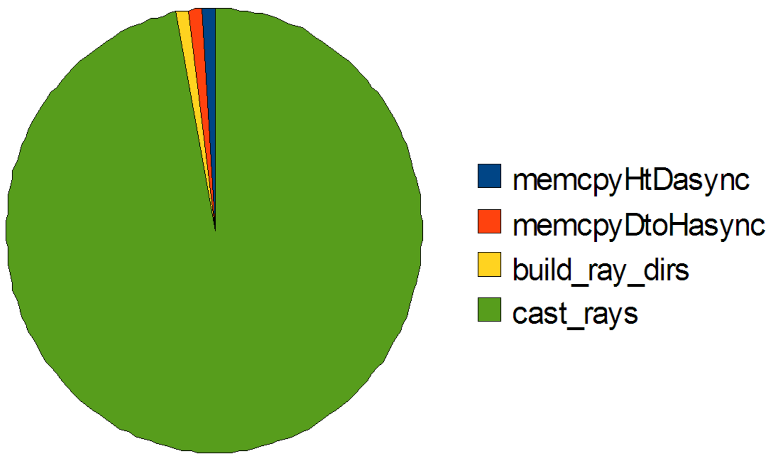


Figure 6.8: Distribution of computation time on Nvidia Tesla C2050 for ray casting

Figure 6.8 shows the distribution in the time for ray casting on the GPU. One quickly sees that transfers between host and device are negligible, as is expected given only a few camera parameters are sent to the device and a 480 KB rendered frame is sent back. This demonstrates the advantage of already having the reconstructing volume in the device memory, making additional transfers unnecessary. Building the ray directions is also a simple task, as demonstrated in the chart, and this is easily explained when one takes into account that the number of rays to be built is only $800 \times 600 = 480,000$, while the number of voxels to be sampled is $512 \times 256 \times 512 = 67,108,864$.

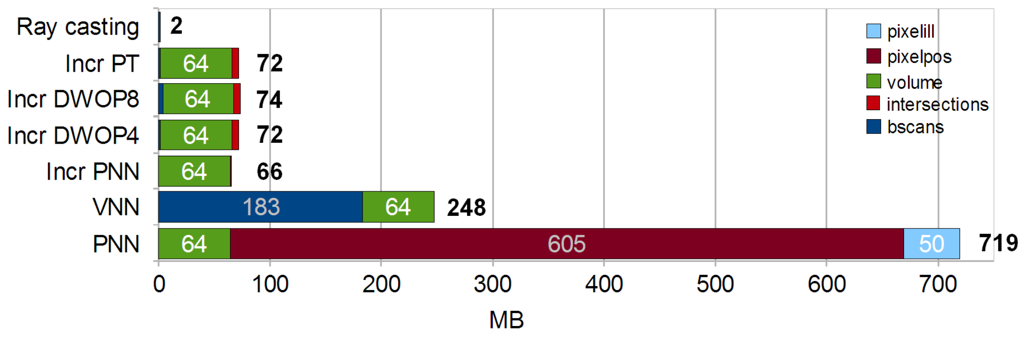


Figure 6.9: GPU memory use

6.1.5 Memory Use

The chart in Figure 6.9 shows how much GPU memory is used by each reconstruction method and the ray casting when using the previously described test data as input. It is easy to see that PNN uses the most memory, a total of 719 MB. Actually, it also requires 183 additional megabytes for the input b-scans, but as these are processed into the 50 MB *pixelill* at the beginning of the procedure they can subsequently be freed from the GPU memory, hence they are not shown in the chart. The biggest contribution to PNN's memory usage is the *pixelpos*, and this is not surprising with three 4-byte coordinates required per pixel. The VNN on the other hand, requires only the input b-scans and output volume. In all cases, there are negligible amounts of memory used for tracking data, timetags, etc. These are so insignificant that they are not noticeable in the chart, and are so not of interest in this discussion.

The incremental methods are quite similar in that they do not use much memory except for the volume, because the input is processed in small chunks at the time. While incremental PNN only keeps a single b-scan in memory at the time, incremental DWOP and PT utilize a "window" of b-scans and other data as previously described in Section 5.3.2. $DWOP_4$ and PT have a window size of 4, resulting in approximately 1.7 MB for b-scans and other data, while $DWOP_8$ uses 8, requiring approximately 3.5 MB, all relatively small amounts of data. The ray casting of course needs a volume to render, but it is assumed that this is already present in the device memory, and so only an additional 2 MB is required (and most of it is the rendered frame).

6.2 Reconstruction Quality

The last section discussed the quantitative performance of the presented methods of this thesis. Obviously, there may be a reason for not simply choosing the fastest method above all else, and that reason is reconstruction quality. Thus, for a full evaluation both performance and reconstruction quality need to be taken into account. In this section, we discuss reconstruction quality and present the quality of the various methods. The effect of noise in the tracking data and choice of compounding method is also discussed. In Appendix B, large uncropped figures can be found.

The test input is the same as given in the performance results section. As a reference point, Figure 6.10 shows some b-scans from this set. As seen in the b-scans, the probe used is of the linear type. The reconstructed volumes from each method will be presented shortly, but since they are in general very similar, we will extract an interesting part of the volume for each method. To get a view of what the volumes reconstructed by the methods described look like, Figure 6.11 shows a typical reconstruction result (obtained using the PT method).

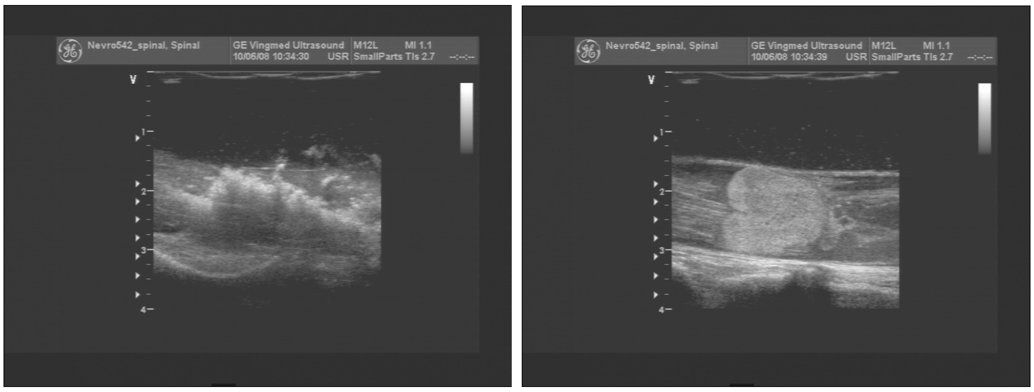

Nevro542_spinal, Spinal
GE Vingmed Ultrasound M12L MI 1.1
10/06/08 10:34:30 USR SmallParts Tls 2.7
Nevro542_spinal, Spinal
GE Vingmed Ultrasound M12L MI 1.1
10/06/08 10:34:39 USR SmallParts Tls 2.7

Figure 6.10: B-scan number 60 and 225 of the input set given in Table 6.2

6.2.1 What is Quality

It is difficult to define a scale for a qualitative measure such as reconstruction quality. The goal of reconstruction is to be as close to the ground truth as possible, but the reality is that given a ROI of 342×356 bytes over 434 b-scans and an output volume of $512 \times 256 \times 512$ bytes, the input is only 79 % of the output. This means that in the case of our test data, even with

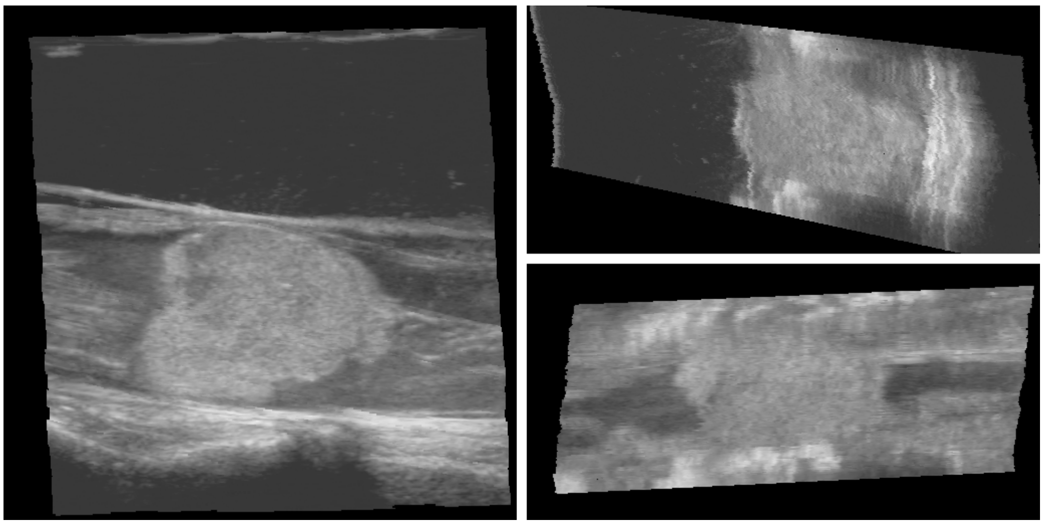


Figure 6.11: Example of three orthogonal slices of a reconstructed volume (left: X-axis, top-right: Z-axis, bottom-right: Y-axis) generated by our implementation

a perfect reconstruction algorithm, the true volume cannot be calculated as some interpolation is required.

So given that the volume will be an estimate, the discussion is what the *best* estimate is, and this also depends on how the volume will be used. Ultrasound scans are analyzed by trained medical personnel, and one of the main uses of the reconstructed volume is to be examined by such people. There is an important tradeoff between avoiding false positives and hiding true positives. A false positive is an artifact of the data set that the analyzer can misinterpret as some symptom that in fact do not exist, while a hidden true positive is an actual existing artifact that is not noticeable by the analyzer. Both cases are unwanted. However, one can argue that the latter case is more important to avoid.

6.2.2 Effect of Noise in Tracking Data

Any reconstruction method can only be as good as the input it is given. For the input data used in this thesis, the quality of the ultrasound b-scan images was more than adequate compared to the quality of the tracking data. For the test data used in this chapter, the ROI is 3.7×4.0 cm in world space, and the freehand movement covers a stretch of about 2 cm. When dealing with such small sizes, inaccuracies in the tracking can be expected. Figure 6.12

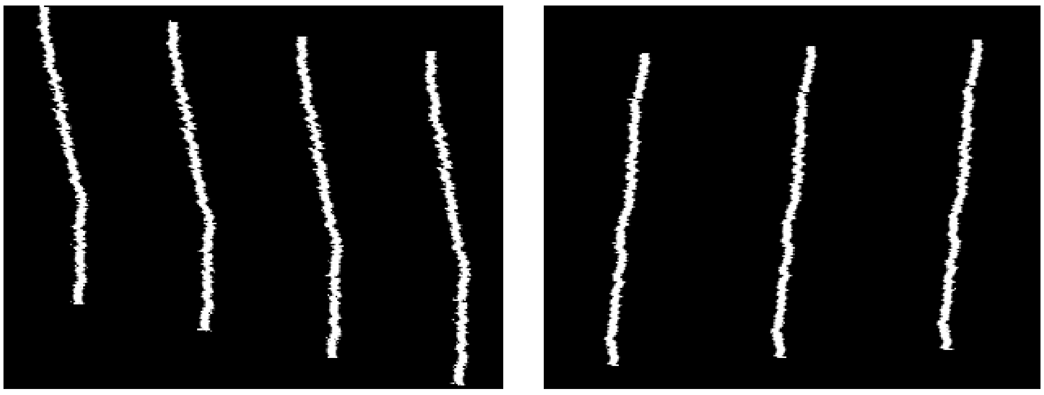


Figure 6.12: Noise in the tracking data seen in reconstructed volume (left: seen along Z-axis, right: seen along X-axis) generated by our implementation

shows the result of reconstructing images of straight lines using the tracking data. One can clearly see the oscillating disruption from the general probe trajectory. The effect is especially noticeable because of the small distances between adjacent scans.

A plausible explanation could be that the freehand movement of the probe actually was jittery (e.g. an unsteady hand), but if one would look at the input b-scans before reconstruction, one would see that details are preserved on the same location over several neighbor images without any jitters. Thus, we conclude that the accuracy of the tracking data is poor, and this affects the results of all reconstruction methods.

6.2.3 Reconstruction Quality Results

Figure 6.13 shows the results of various reconstruction methods. An interesting region of the volume is cropped from the rest to make it easier to spot key differences, and uncropped large figures can be found in Appendix B. Each crop is a slice of the same area looking in the direction of the X-axis. The compounding method used for all methods is *overwrite*.

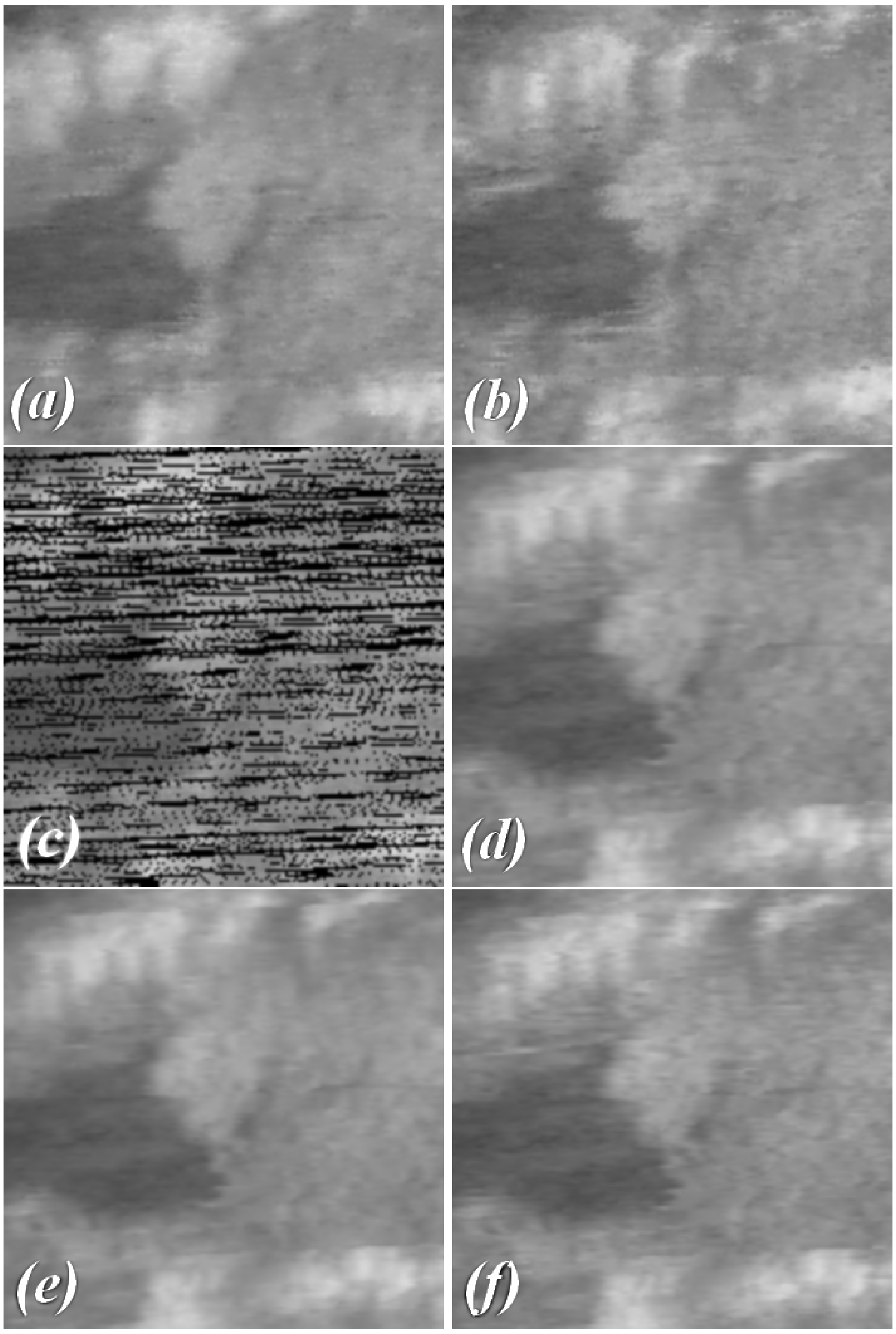


Figure 6.13: Cropped section of reconstructed volume generated by our implementation. (*a*) PNN. (*b*) VNN. (*c*) PNN w/ no hole filling. (*d*) incremental $DWOP_4$. (*e*) incremental $DWOP_8$. (*f*) incremental PT.

PNN Quality

As seen in Figure 6.13a, the PNN is somewhat grainy, and this is expected when using only the nearest neighbor for filling instead of some method of weighting. Some of the grains are also due to the fact that there are not enough pixels to fill the entire volume. This means that along a row of pixels, at some point a voxel is skipped. E.g. if there are 20 pixels along a row of a b-scan, and they are reconstructed onto a voxel that has 25 voxels along a row of the same size, then 5 voxels are periodically left empty. This is clearly seen in Figure 6.13c where hole filling is turned off. In addition to the black lines of holes between b-scans, there are periodical holes every 4-5 voxels in a grid because the pixel spacing is $\simeq 0.1$, while the voxel spacing is 0.08, i.e. $\frac{0.1}{0.08} = 1\frac{1}{4}$. Additionally, one can also see the tendency for horizontal lines in the image, but this is mainly due to the tracking noise discussed earlier, and is present in all cases.

VNN Quality

One can see in Figure 6.13b that the VNN is sharper than the PNN. More details are preserved in the white spots at the top, and the texture of the gray area at the right is more clearly visible. This can be caused by the smoothing effect of averaging voxels when filling the PNN holes. The VNN method is still based on nearest-neighbor however, and also suffers from graining as the PNN method. This is unavoidable as two neighboring voxels often have different closest b-scans when the scans are packed together.

Incremental DWOP Quality

$DWOP_4$ and $DWOP_8$, shown in Figure 6.13d and 6.13e, do not have the problem of grains that occurs with PNN and VNN. This is of course because several b-scans influence each voxel, and those b-scans are weighted differently. The drawback, however, is a slightly noticeable blurring of the volume when compared to VNN. However, one can argue that the sharper features of VNN might be largely due to the effect of the grains, thereby effectively allowing a cluster of grains to be mistaken as a feature.

The difference between $DWOP_4$ and $DWOP_8$ is not substantial, but the $DWOP_8$ is slightly smoother than the $DWOP_4$. This is explained by the fact that $DWOP_8$ weights *eight* neighbor b-scans instead of *four*, and thus each voxel is influenced by b-scans further away than with $DWOP_4$, causing the blurring. Since the weighting is distance based, this blur is minor, but the similarity between $DWOP_4$ and $DWOP_8$ does not justify the increase in computation time associated with $DWOP_8$.

Incremental PT Quality

The result of the PT method is shown in Figure 6.13f, and as the other weighting based methods, it does not contain grains. However, in comparison to DWOP, it is clearer and sharper, with the details of the white area at the top and the texture of the gray area to the right being superior to those from other methods. With PT having only a slightly greater computation time than $DWOP_4$ it is evident from this thesis that PT is a superior method.

To illustrate another advantage of the PT method, Figure 6.14 and shows the results when the input is sparse. With more space between the b-scans, the approach to filling this space becomes more important. Figure 6.15 highlights the differences even more by using straight lines instead of ultrasound b-scans (but using the real tracking data).

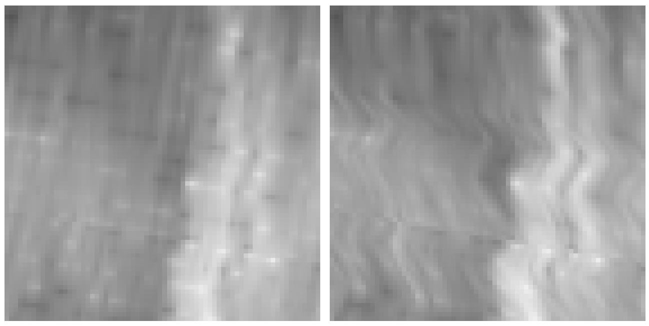


Figure 6.14: Reconstruction of sparse input ($DWOP_8$ and PT) generated by our implementation

As demonstrated in Figure 6.14, because DWOP is based on orthogonal projections, one can clearly see tendencies for straight lines at approximately an 80 degrees angle. These lines are orthogonal to the b-scan plane normals. PT uses cubic interpolation along the probe trajectory, and one can see how the features curve along the tracked path (with tracking noise).

With the straight lines of Figure 6.15, one can also see how the PT is superior in filling in values between sparse b-scans. The reconstructed line from DWOP appears discontinuous across the adjacent planes. PT in contrast, preserves the intensity of the line along the probe trajectory.

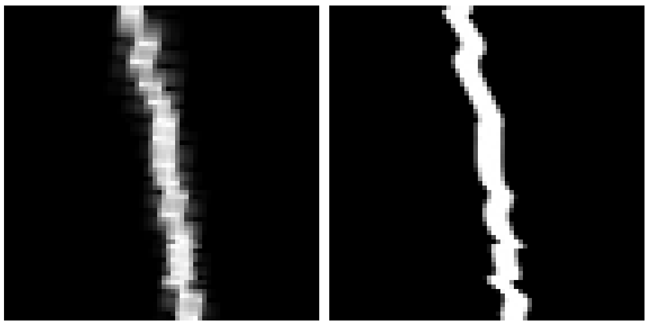


Figure 6.15: Reconstruction of sparse input (straight lines) ($DWOP_8$ and PT) generated by our implementation

6.2.4 Effect of Compounding Methods

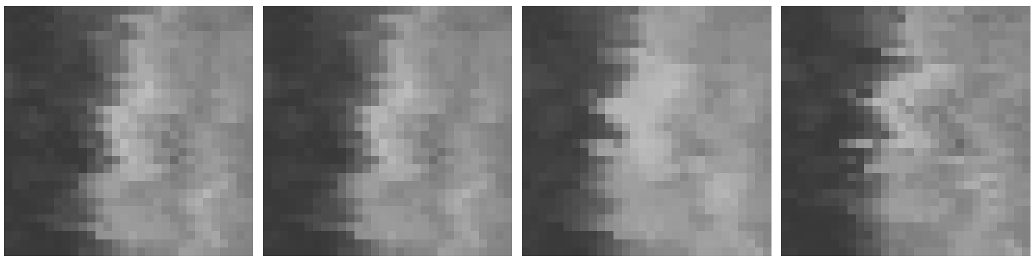


Figure 6.16: Result of compounding methods generated by our implementation: *overwrite*, *avg*, *max* and $ifempty$ (PT method)

Figure 6.16 shows the effect of each of the four compounding methods as presented in Table 4.1: *overwrite*, *avg*, *max* and $ifempty$. Larger uncropped figures can be found in Appendix B. As expected, the *avg* method slightly blurs the output compared to *overwrite*, but since the averaging occurs internally in each voxel, the details and features persist.

The *max* method is smoother and brighter than the others. This is the case since only the brightest voxel values are kept, resulting in an overall brighter output. Some details are lost, however, when comparing to the two previous methods, and this can be explained by the fact that a single bright voxel value will dominate any set of darker values for that voxel. With some noise in the tracking, a bright pixel will thus "spread out", resulting in smoother transitions.

The result of $ifempty$ is somewhat different than *overwrite* and *avg*, and one can see some features appear in this method that do not appear in the results of *overwrite* and *avg*. The main difference between these methods is that *overwrite* keeps the *latest* value of a voxel, *ifempty* keeps the *first* value, and *avg* averages them all. Given that *overwrite* and *avg* so similar, it can be speculated that the last voxel values are more "correct" than the first values. Noise in the tracking data should be just as likely to cause jitter forward as well as backward, and so it is believed to be a property of the incremental reconstruction method.

6.3 General Discussion

As the result of the work done in this thesis, some major topics can be discussed in general. Here, we discuss the differences between pixel-based and voxel-based reconstruction, and compare incremental and non-incremental reconstruction. These approaches have their own challenges and advantages, and from the experience gained in this thesis we address these.

6.3.1 Pixel-Based *vs.* Voxel-Based Reconstruction

Although both pixel-based and voxel-based reconstruction have the same goal, to reconstruct the volume according to given input, they are fundamentally different. Pixel-based methods tend to be easier to understand and implement, as there is a natural path from input to output. This also makes them intuitively superior for incremental reconstruction, where the input needs to be processed one slice at a time into the volume, and additionally we want to avoid processing the entire volume for each new increment.

But simple pixel-based methods such as PNN have a main disadvantage of leaving holes in the volume. Although these can be filled using averaging or interpolation methods, doing so is inherently a voxel-based task: for each voxel with a hole, it must be filled with a suitable value. If such a step is to be performed anyway, one can argue that the whole process should be streamlined as voxel- based in the first place.

Another disadvantage of pixel-based methods is that read-write conflicts can occur in the volume when processing in parallel. When a pixel is used to update one or more of the volume voxels, they may already have a value that should be accumulated with the chosen compounding method. Race conditions can occur in such cases where pixels are updating the same voxel(s). If complex pixel-based methods are used, such as splatting a sphere around each pixel, then even more conflicts will occur. Although atomic operations on the voxels would solve this, it would introduce a potentially heavy performance penalty if there were many conflicts (as neighboring pixels are prone to have).

In general, voxel-based methods can offer high quality in the reconstructed volume. One can reason for this by claiming that given accurate tracking, reconstruction is fundamentally a resampling problem: There exists a set of data points in space defined by the oriented b-scans, and the task is to resample them onto a rectangular grid of voxels.

Another issue where pixel-based and voxel-based methods differ is how they scale with increased problem sizes. In the case of reconstruction, the problem size is determined by many parameters. The most salient are:

- n = number of b-scans
- B = size of b-scans
- V = size of volume
- K = size of hole fill kernel

In addition, there are algorithm specific parameters such as number of b-scans taken into account for each voxel in DWOP and the values of performance-linked cutoff limits. For simplicity, we choose to ignore such parameters. Using big-O notation, PNN has the complexity

$$O(nB + VK) \tag{6.1}$$

where the first part is due to each of B pixels of the n b-scans being processed. The last part is due to the hole filling. VNN on the other hand has the complexity

$$O(Vn) \tag{6.2}$$

which is due to taking the n b-scan into account for each of the V voxels. From the functions, it is clear that both PNN and VNN are dependent on the volume size and the number of b-scans, which are the two parameters most likely to vary significantly. However, the PNN has a complexity based on a sum of these two parameters, while with VNN it is a product.

6.3.2 Incremental *vs.* Non-Incremental Reconstruction

The motivation for doing reconstruction incrementally is the ability to perform *real-time* reconstruction, thus allowing the user to get instant feedback on the freehand ultrasound scanning, and if desirable, to rescan interesting areas or adjust settings early if the original result was not satisfactory. However, independent of the specific reconstruction method used, there are some challenges introduced when doing the reconstruction incrementally as opposed to a bulk-operation with all the data ready.

Unknown Volume Extents

One problem is that the extents of the volume is not known in beforehand. Even though the location of the first incoming b-scan can be used to assume the *location* of the volume in space, its final *size* in the three dimensions is unknown. It might also be the case that a specific *orientation* of the volume would allow better utilization of the space, e.g. if the scans fit better into a volume rotated 45 degrees. The quick fix is to assume a certain size and orientation from the start, and allocate memory for the volume given these assumptions. Data outside these extents will then be ignored. It would be possible to dynamically allocate memory as the volume grows, but such a scheme would be complex and may reduce performance. In the non-incremental case, however, the input data can be first analyzed to figure out a suitable volume size that covers all the data.

Scope of Available Data

Another advantage with non-incremental approaches is that they can take *all* the b-scans into account for the reconstruction. By using all available information, the reconstruction should be closer to the actual ground truth. Incremental methods, however, only know the *past* data, and not the future. The means to overcome this, as described in the previous chapter, is to build up a buffer of incoming data before the reconstruction begins, and then update the volume with this queue of b-scan data. In this way, both the past and some of the future b-scans can be taken into account. The delay introduced by this approach to the real-time reconstruction is negligible given a high rate of incoming data.

Data Transfer Bottleneck

As the volume is reconstructed incrementally on the GPU, it is also a challenge to keep an updated volume in host memory. In the non-incremental case, such a transfer need only be performed once at the end of the reconstruction. Doing this at each increment introduces much overhead, especially if the reconstruction procedure is computationally easy and the transfer time dominates the processing time. However, while reconstruction must be performed once per incoming b-scan to be real-time, the host memory volume need not be updated at the same rate. If the volume in device memory is used for the visualization, a difference in update rates will not be immediately noticed by the user.

Chapter 7

Conclusions and Future Work

Ultrasound is a non-invasive, safe, low cost and practical way to provide medical doctors with an internal view of a patient's body. By performing 3D ultrasound reconstruction, 3D volumes can be constructed from 2D ultrasound scans, and be used for acquiring out-of-angle views, 3D rendering of the anatomy, and for image guided surgery. The purpose of this thesis was to investigate the parallel processing power of the GPU for fast ultrasound reconstruction. Having the ability to reconstruct a volume in a fraction of a second enables instant feedback and real-time incremental reconstruction while scanning. This thesis has presented our techniques to perform fast, non-incremental and real-time incremental reconstruction using the GPU. Optimization techniques for both pixel-based and voxel-based approaches have been described, and a novel method of doing real-time incremental reconstruction was presented. The performance obtained by these methods have been measured on some of the latest hardware architectures at the time of writing from both Nvidia and AMD. Issues for future work to look into are suggested in Section 7.2, and some final thoughts and closing statements are also given in Section 7.3.

7.1 Conclusions

Our work resulted in *Thunder*, a software implementation of the developed techniques. This system included fast reconstruction with the VNN (voxel-nearest-neighbor) and PNN (pixel-nearest-neighbor) methods, and real-time incremental high-quality reconstruction by distance weighted orthogonal projections or based on the probe trajectory. Furthermore, the reconstructing volume could be visualized in real-time by orthogonal MPR slices (planar slices through the volume) or volume ray casting on the GPU.

By utilizing the GPU, a speedup of up to 50 was achieved by VNN on the new Fermi architecture by Nvidia. PNN obtained 14 times, and the incremental methods got between 6 and 8 times the performance compared to a pure CPU implementation. This meant that the reconstruction of non-incremental PNN and VNN volumes was performed in only 0.9 and 0.6 seconds, and the incremental methods achieved times of 3.3 seconds (incremental PNN), 24.7 seconds (DWOP of 4 scans), 34.5 seconds (DWOP of 8 scans) and 26.1 seconds (PT).

As for quality, the PT method demonstrated the best results, especially when handling sparse input. While the parallel nature of ultrasound reconstruction has proved suitable for the GPU, incremental reconstruction was limited by the overhead associated with data transfer between device and host for each increment.

7.2 Future Work

There are many possible avenues to investigate further. Here are some suggestions:

Reduce device memory required for PNN

Older or low-budget GPUs do not have the amount of device memory required for some of the reconstruction methods presented in this thesis. Also, for a mobile system, one can imagine GPUs in handheld devices with small memory sizes being used for the reconstruction. The PNN method as described here requires a total of 719 MB, and this is impossible or difficult to fit on GPUs with 512 or 768 MB of total memory (that may also be used by other applications). Although memory use can be reduced by a smaller problem size, it is also desirable to reduce the requirements of the algorithm itself. Most of the memory used by the PNN is for the three-dimensional coordinates of each pixel. This can be optimized by calculating these coordinates only when needed. Given a reference point for each b-scan in addition to its plane equation and given pixel spacings, it is possible to calculate the coordinates of any pixel on that b-scan. This could reduce the memory needed by as much as 59 % for the PNN method of reconstruction.

Dynamic volume allocation and orientation

Currently, the volume extents and orientation are predefined before reconstruction, and data acquired outside of this is disregarded. A dynamic volume allocation technique could be devised and implemented, such that the volume is increased (and perhaps also reduced) according to needs. If feasible on a performance basis, one could even rotate the volume such that the scanned data always fits optimally.

Reducing the device-to-host transfer bottleneck

The main bottleneck with incremental reconstruction is the transfer of the volume from the device to host. Solutions for reducing the bandwidth needed for this could be devised and implemented. Incremental PNN without hole filling already employs a simple scheme for sending only the relevant data from device to host for each increment, but this is more complex when using the high-quality incremental methods as presented in this thesis. It should however be possible to design an efficient technique for sending only the required data for each increment, which is an amount substantially smaller than the entire volume.

Handle U-turn scanning

When scanning, the probe can be moved in a U-turn. This is problematic for the voxel-based reconstruction because the b-scans are modeled with plane equations that stretch infinitely out. The voxels that belong to one arm of the U might find that b-scans in the other arm are closer (according to the plane equations). When the ROI is then investigated, it will be concluded that the voxel is outside the ROI of the b-scan from the other arm even though it might be inside the ROI of a b-scan from the closest arm. These challenges should be addressed and solved.

Dynamic adaptation of memory allocation

As mentioned above, some GPUs do not have enough memory for some of the data buffers used in reconstruction as described in this thesis. Furthermore, even though the total memory is enough, there are often restrictions on how much of the memory that can be allocated for a single buffer. The solution is to split up the input and output data, and process them in smaller pieces. To allow for portability of the system, the splitting should be handled automatically according to the capabilities of the installed GPU. Such an automatic system needs to allocate buffers of the right sizes, communicate to the GPU kernels what parts of memory they are to read from and write to, and merge together the output pieces.

7.3 Final Thoughts

The purpose of this thesis was to utilize the parallel processing power of the GPU for fast ultrasound reconstruction. We developed techniques that enable reconstruction of an entire volume in only fractions of a second using both pixel-based and voxel-based approaches, and for incremental reconstruction, real-time performance was obtained using our methods. This thesis showed how the parallel nature of ultrasound reconstruction can be exploited, and the techniques developed can benefit anyone who wants to utilize the power of the GPU for reconstruction.

Bibliography

[1] Advanced Micro Devices, Inc. *ATI Radeon HD 5870 GPU Feature Summary*, 2010. `http://www.amd.com/uk/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-specifications.aspx`, retrieved 2010-06-17.

[2] Advanced Micro Devices, Inc. *ATI Stream SDK Release Notes*, 2010. `http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_Release_Notes_Developer.pdf`, retrieved 2010-06-17.

[3] Anderson, M. E., McKeag, M. S., and Trahey, G. E. The impact of sound speed errors on medical ultrasound imaging. *The Journal of the Acoustical Society of America 107*, 6 (2000), 3540–3548.

[4] Barry, C., Allott, C., John, N., Mellor, P., Arundel, P., Thomson, D., and Waterton, J. Three-dimensional freehand ultrasound: Image reconstruction and volume analysis. *Ultrasound in Medicine & Biology 23*, 8 (1997), 1209 – 1224.

[5] Coupé, P., Hellier, P., Azzabou, N., and Barillot, C. *3D Freehand Ultrasound Reconstruction Based on Probe Trajectory*, vol. 3749/2005 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 597–604.

[6] Eberly, D. H. *Game Physics*. Morgan Kaufmann, 2004.

[7] Fagerlund, O. A. An investigation of the emerging open computing language. Norwegian University of Science and Technology (NTNU), 2008.

[8] Fenster, A., and Downey, D. 3-D ultrasound imaging: a review. *Engineering in Medicine and Biology Magazine, IEEE 15*, 6 (nov/dec 1996), 41–51.

[9] Fenster, A., Surry, K., Smith, W., Gill, J., and Downey, D. B. 3D ultrasound imaging: applications in image-guided therapy and biopsy. *Computers & Graphics 26*, 4 (2002), 557 – 568.

[10] Gobbi, D. G., and Peters, T. M. *Interactive Intra-operative 3D Ultrasound Reconstruction and Visualization*, vol. 2489/2002 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 156–163.

[11] Gonzalez, R. C., and Woods, R. E. *Digital Image Processing*, third ed. Prentice-Hall, 2008.

[12] Hearn, D., and Baker, M. *Computer Graphics with OpenGL*, third ed. Prentice-Hall, 2004.

[13] Herikstad, Å. Parallel techniques for estimation and correction of aberration in medical ultrasound imaging. Master's thesis, Norwegian University of Science and Technology (NTNU), 2009.

[14] Huang, W., Zheng, Y., and Molloy, J. 3D ultrasound image reconstruction from non-uniform resolution freehand slices. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on* (18-23, 2005), vol. 2, pp. 125 – 128.

[15] Karamalis, A., Wein, W., Kutter, O., and Navab, N. Fast hybrid freehand ultrasound volume reconstruction. In *Medical Imaging 2009: Visualization, Image-Guided Procedures, and Modeling* (2009), M. I. Miga and K. H. Wong, Eds., SPIE.

[16] Khronos OpenCL Working Group. *The OpenCL Specification*, 2009. `http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`, retrived 2010-06-17.

[17] Kirk, D. B., and mei W. Whu, W. *Programming Massively Parallel Processors*. Elsevier Inc., 2010.

[18] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987), ACM, pp. 163–169.

[19] Ludvigsen, H., and Elster, A. C. Real-time ray tracing using Nvidia OptiX. In *Eurographics Short Papers* (2010), The Eurographics Association, pp. 65–68.

[20] McCann, H., Sharp, J., Kinter, T., McEwan, C., Barillot, C., and Greenleaf, J. Multidimensional ultrasonic imaging for cardiology. *Proceedings of the IEEE 76*, 9 (sep 1988), 1063 –1073.

[21] Mercier, L., Langø, T., Lindseth, F., and Collins, D. L. A review of calibration techniques for freehand 3-D ultrasound systems. *Ultrasound in Medicine & Biology 31*, 4 (2005), 449 – 471.

[22] Nelson, T., and Elvins, T. Visualization of 3D ultrasound data. *Computer Graphics and Applications, IEEE 13*, 6 (nov 1993), 50 –57.

[23] Nielsen, E. A. R. Real-time wavelet filtering on the gpu. Master's thesis, Norwegian University of Science and Technology (NTNU), 2007.

[24] Nvidia Corporation. *Quadro FX 5800 Data Sheet*, 2008. `http://www.nvidia.com/docs/IO/40049/NV_DS_QFX_5800_US_Sep08_LowRes.pdf`, retrieved 2010-06-17.

[25] Nvidia Corporation. *CUDA Programming Guide*, 2009. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`, retrieved 2010-06-17.

[26] Nvidia Corporation. *OpenCL Best Practices Guide*, 2009. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf`, retrieved 2010-06-17.

[27] Nvidia Corporation. *Tesla C2050 Board Specification*, 2010. `http://www.nvidia.com/docs/IO/43395/Tesla_C2050_Board_Specification.pdf`, retrieved 2010-06-17.

[28] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J. Gpu computing. *Proceedings of the IEEE 96*, 5 (may 2008), 879 –899.

[29] Prager, R. W., Gee, A., and Berman, L. Stradx: real-time acquisition and visualization of freehand three-dimensional ultrasound. *Medical Image Analysis 3*, 2 (1998), 129 – 140.

[30] Preim, B., and Bartz, D. *Visualization in Medicine.* Morgan Kaufmann, 2007.

[31] Rohling, R., Gee, A., and Berman, L. A comparison of freehand three-dimensional ultrasound reconstruction techniques. *Medical Image Analysis 3*, 4 (1999), 339 – 359.

[32] Sakas, G., and Walter, S. Extracting surfaces from fuzzy 3D-ultrasound data. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), ACM, pp. 465–474.

[33] Sanches, J. M., and Marques, J. S. A rayleigh reconstruction/interpolation algorithm for 3D ultrasound. *Pattern Recognition Letters 21*, 10 (2000), 917 – 926.

[34] Sherebrin, S., Fenster, A., Rankin, R. N., and Spence, D. Freehand three-dimensional ultrasound: implementation and applications. In *Medical Imaging 1996: Physics of Medical Imaging* (1996), R. L. V. Metter and J. Beutel, Eds., vol. 2708, SPIE, pp. 296–303.

[35] Solberg, O. V., Lindseth, F., Torp, H., Blake, R. E., and Hernes, T. A. N. Freehand 3D ultrasound reconstruction algorithms – a review. *Ultrasound in Medicine & Biology 33*, 7 (2007), 991–1009.

[36] Stantchev, G., Juba, D., Dorland, W., and Varshney, A. Using graphics processors for high-performance computation and visualization of plasma turbulence. *Computing in Science and Engineering 11* (2009), 52–59.

[37] Trobaugh, J. W., Trobaugh, D. J., and Richard, W. D. Three-dimensional imaging with stereotactic ultrasonography. *Computerized Medical Imaging and Graphics 18*, 5 (1994), 315 – 323.

[38] Wein, W., Pache, F., Röper, B., and Navab, N. *Backward-Warping Ultrasound Reconstruction for Improving Diagnostic Value and Registration*, vol. 4191/2006 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 750–757.

[39] Welch, J., Johnson, J., Bax, M., Badr, R., and Shahidi, R. A real-time freehand 3D ultrasound system for image-guided surgery. In *Ultrasonics Symposium, 2000 IEEE* (oct 2000), vol. 2, pp. 1601 –1604 vol.2.

[40] Wilkinson, B., and Allen, M. *Parallel Programming*, second ed. Prentice-Hall, 2005.

Appendix A

Annotated Bibliography

In this chapter, there is a collection of selected references that are annotated with a brief explanation and summary of results. These references can also be found in the ordinary bibliography above.

Stradx: Real-time Acquisition and Visualization of Freehand Three-dimensional Ultrasound [29]

This paper presents the Stradx system, which instead of constructing a 3D volume from ultrasound scans generates MPR slices directly. By doing so, they try to link the acquisition and visualization phases by reducing or eliminating the processing time. Their implementation also exploited graphics acceleration hardware when constructing the slices, although the details are scarce.

Interactive Intra-operative 3D Ultrasound Reconstruction and Visualization [10]

This paper presents an implementation of simultaneous real-time 3D ultrasound reconstruction and visualization on the CPU. As in this thesis, the reconstruction occurs while the data is acquired and is done by PNN. In contrast to this thesis, the visualization rendered only three orthogonal slices of the volume, not the volume itself. And the reconstruction is in real-time, but the visualization is only "interactive" at a lower framerate. Furthermore, reconstruction and visualization were both done on a dual (933 MHz Pentium

III) CPU. In the paper, they present performance obtained by four pixel-based reconstruction method variants. A 256x193x256 volume was reconstructed from cropped 320x240 b-scans at 12 to 30 scan/s and visualization was updated simultaneously at 5 fps.

Three-Dimensional Imaging With Stereotactic Ultrasonography [37]

This paper describes a reconstruction method based on distance weighted orthogonal projections. For each voxel, its distance is calculated to the two surrounding b-scans (if existing), and the pixel values on the projected points are weighted by the distance to give the voxel's value. The authors compare this method to a simple PNN method. Using a Sun SPARCstation 1, they reconstructed a 128x128x50 voxel volume from 256x256 pixel scans at 54 seconds per scan.

3D Freehand Ultrasound Reconstruction Based on Probe Trajectory [5]

This paper presents a new reconstruction method that improves quality, especially on sparse b-scan input. The method is voxel-based and takes an estimate of the probe trajectory into account when reconstructing. As in [10], the implementation was on a CPU, in this case a single Pentium 4 3.2 GHz with 2GB RAM. The implementation reconstructed a volume of unspecified size from 204 and 222 510x441 b-scans in 111 to 124 and 138 to 149 seconds. As future work, the paper suggests acceleration by GPU, as done in this thesis.

Backward-Warping Ultrasound Reconstruction for Improving Diagnostic Value and Registration [38]

This paper presents a novel voxel-based reconstruction method with better quality/speed properties. The authors used an AMD64 3200+ CPU with 1GB RAM, and reconstructed a 16 and a 134 million voxel volume from 1024 256x256 and 454x454 b-scans in 226 to 942 seconds for "multiple" mode and 119-510 seconds for "single" mode. The authors' goal was obtaining high performance, and their results can be compared to those obtained in this thesis.

Fast Hybrid Freehand Ultrasound Volume Reconstruction [15]

This recent paper uses a hybrid between forward and backward reconstruction with the goal of high performance. The hardware used was an Intel Xeon 3.2 GHz with 2GB RAM and a Nvidia GeForce 8800GTX with 768MB RAM. Their implementation reconstructed a 256x256x256 volume from 293 b-scans in 0.35 seconds with simple interpolation and 0.82 seconds with advanced interpolation.

Visualization in medicine [30]

This a thorough book on visualization with focus on its use in medicine. It describes the various algorithms as well as their clinical applications, and covers acquisition, processing and rendering of medical data in two and three dimensions. Much focus in the book is on volumetric data, as is commonly used in medicine, and all of the important direct and indirect visualization techniques are explained.

Freehand 3D Ultrasound Reconstruction Algorithms: A Review [35]

This paper is a comprehensive review of reconstruction algorithms for ultrasound. As a part of this, the algorithms are categorized into three distinct groups: voxel based, pixel based and function based methods. The paper explains implementation of many algorithms and their variants, and focuses on evaluation and comparison of these with regards to their efficiency and effectiveness. As a part of the work, some of the methods have been implemented and tested on a laboratory phantom model, and the results are documented and discussed.

Appendix B

Large Figures

For clearity, some of the figures in the thesis have been cropped and scaled down. In this appendix, there are large, uncropped versions of those figures.

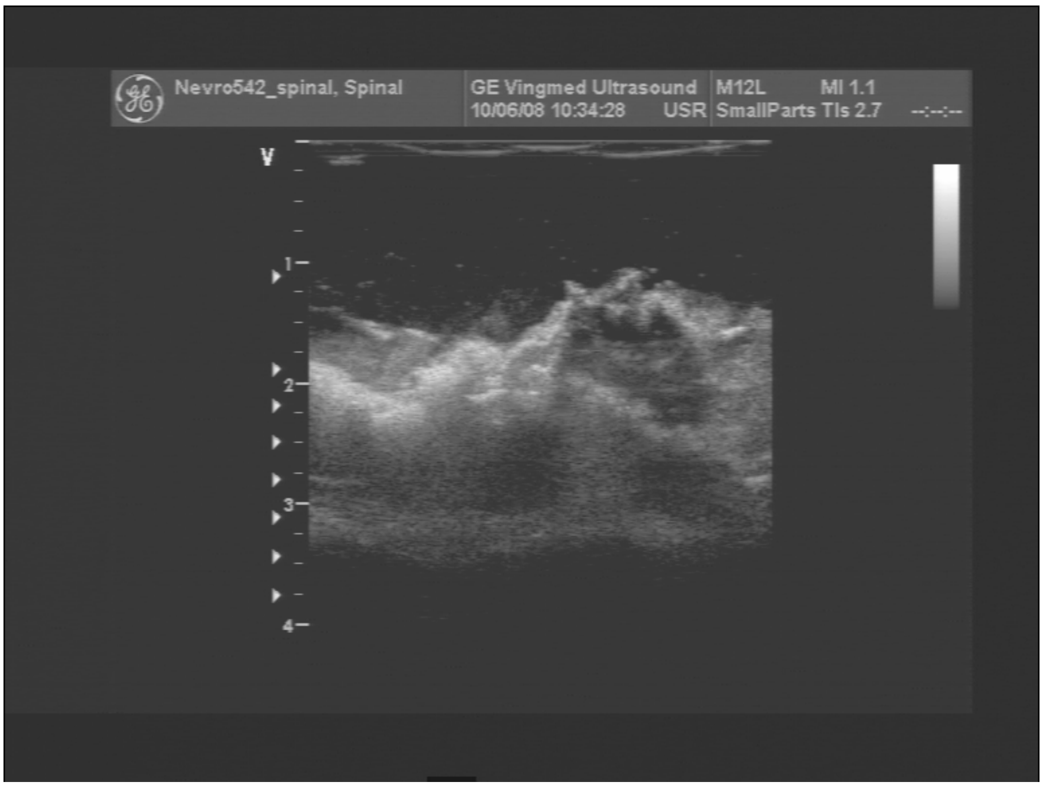


Figure B.1: B-scan number 1/434

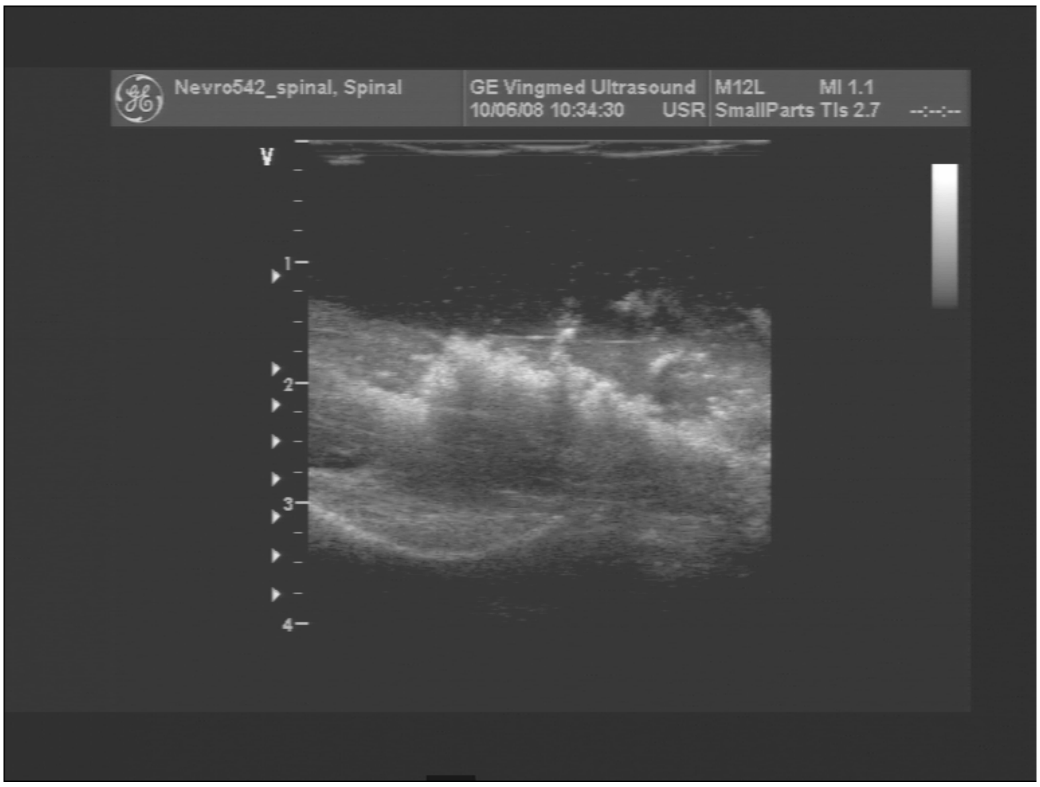


Figure B.2: B-scan number 60/434

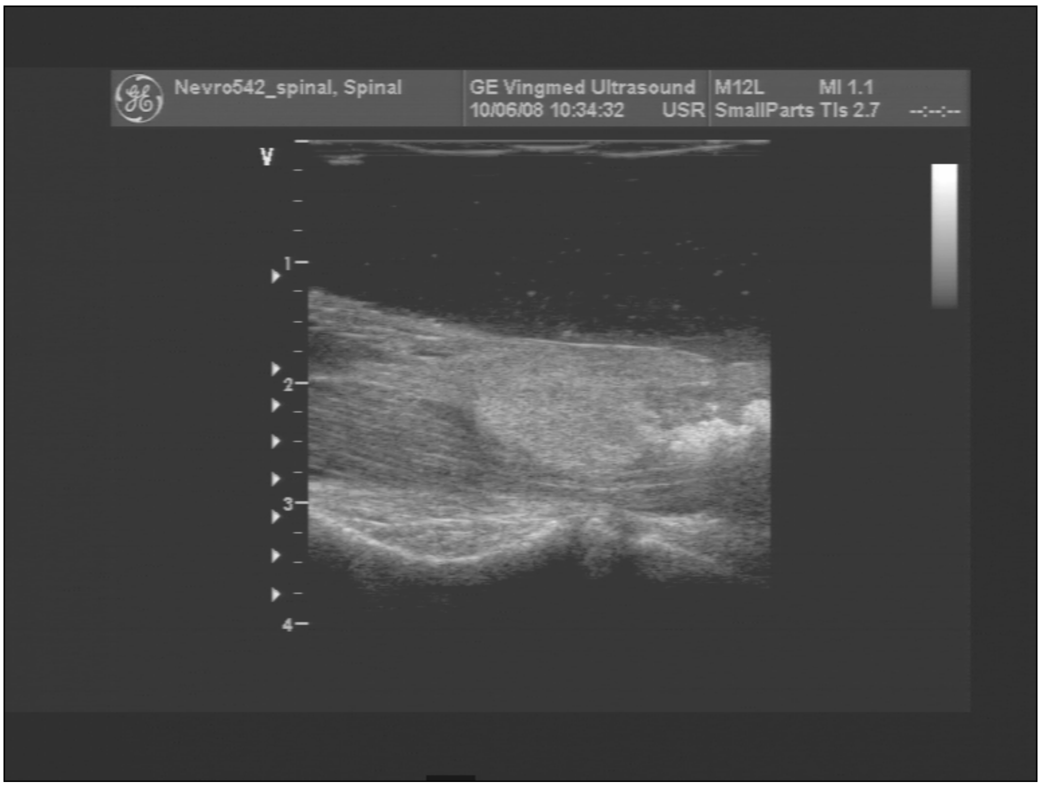


Figure B.3: B-scan number 100/434

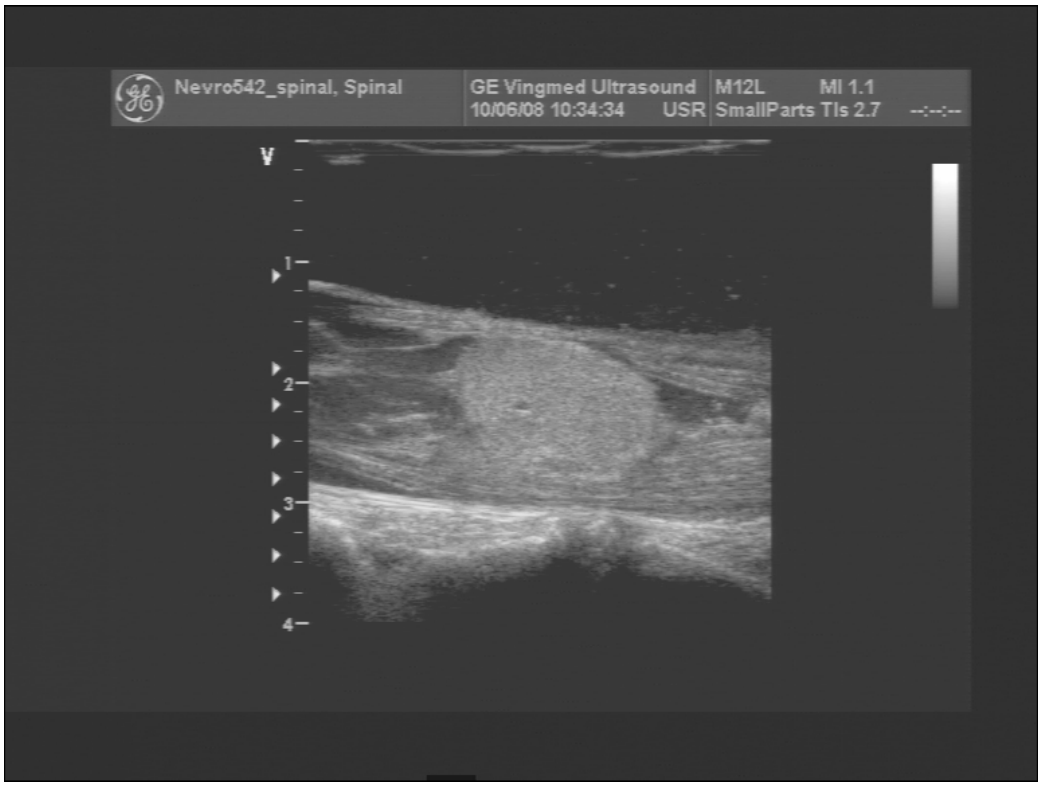


Figure B.4: B-scan number 165/434

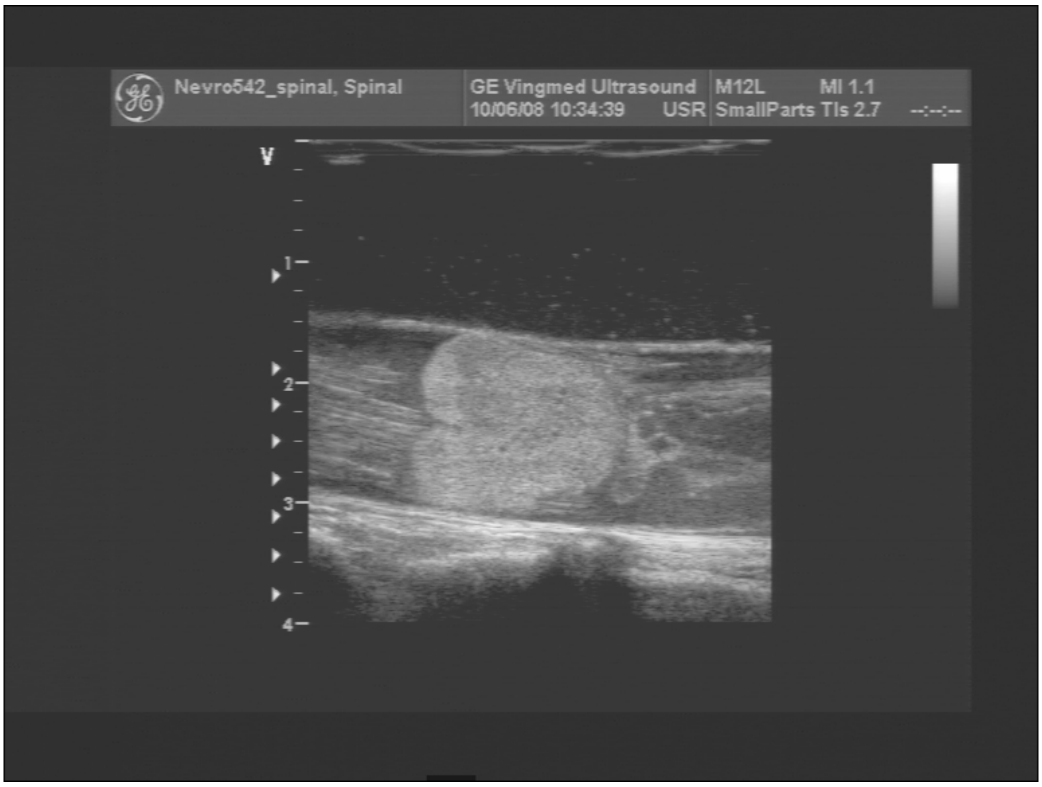


Figure B.5: B-scan number 280/434

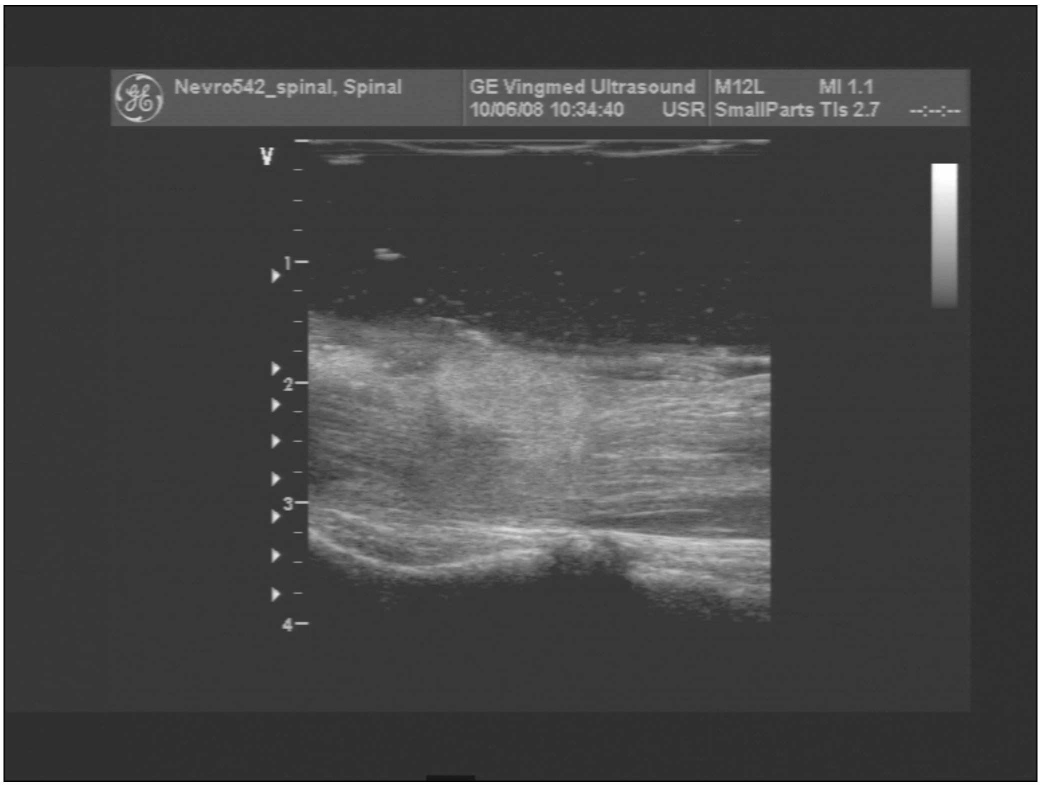


Figure B.6: B-scan number 315/434

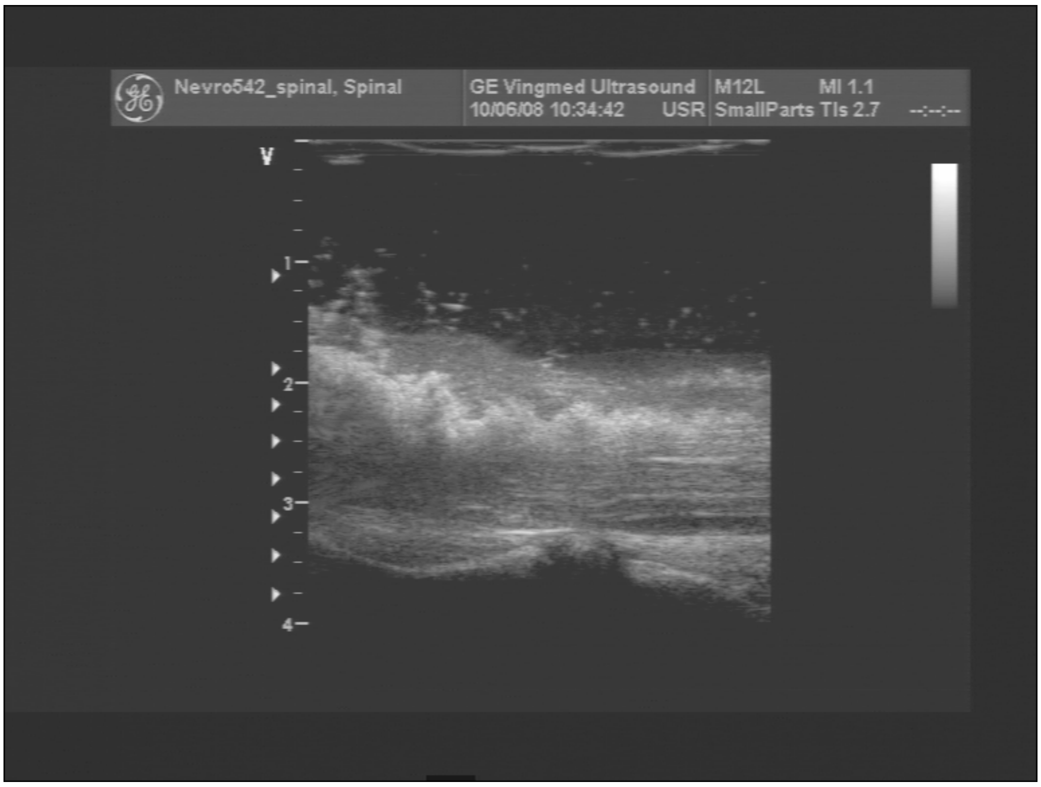


Figure B.7: B-scan number 360/434

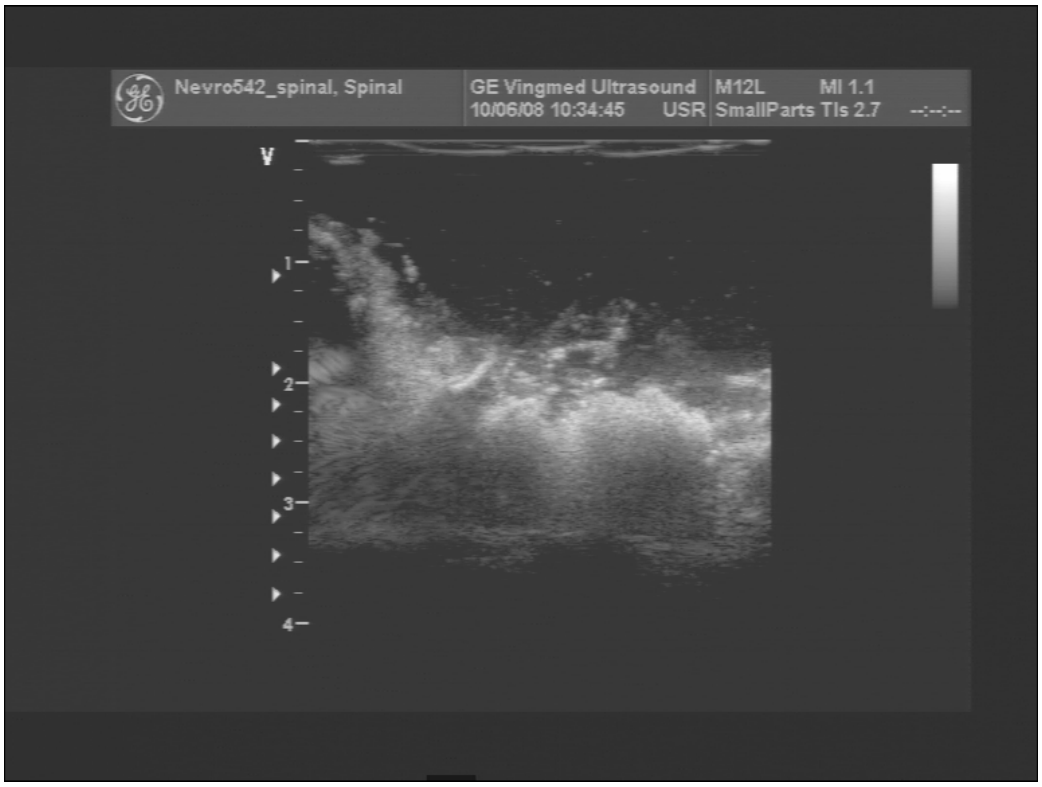


Figure B.8: B-scan number 434/434

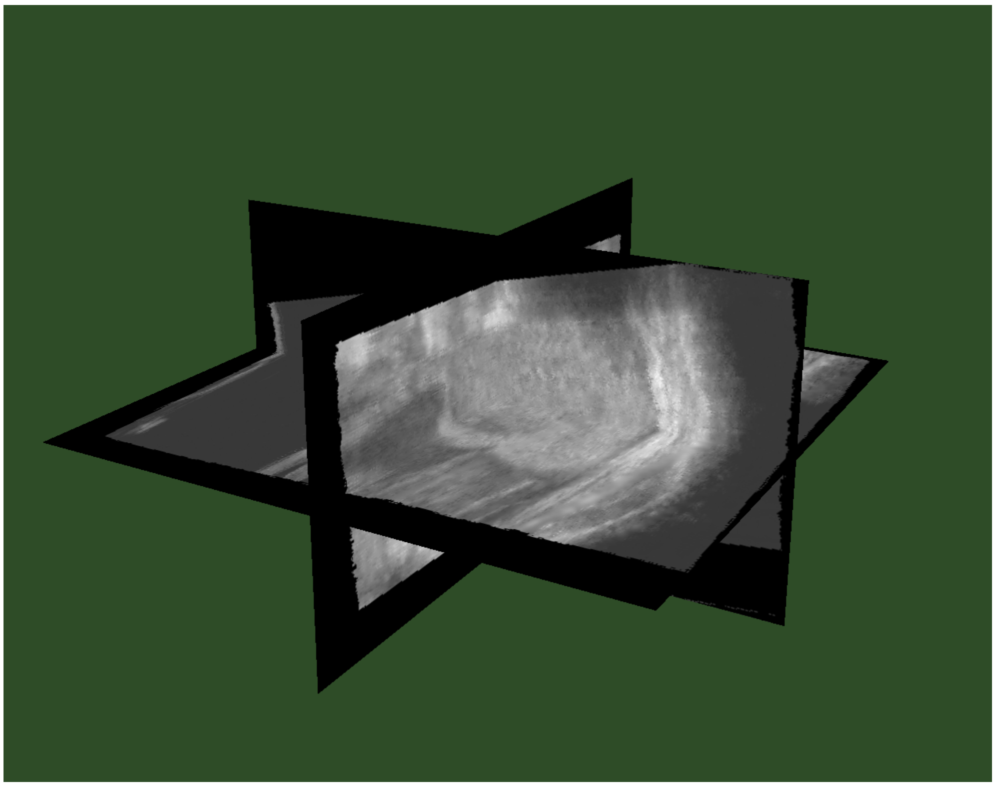


Figure B.9: Orthogonal MPR slices screenshot 1

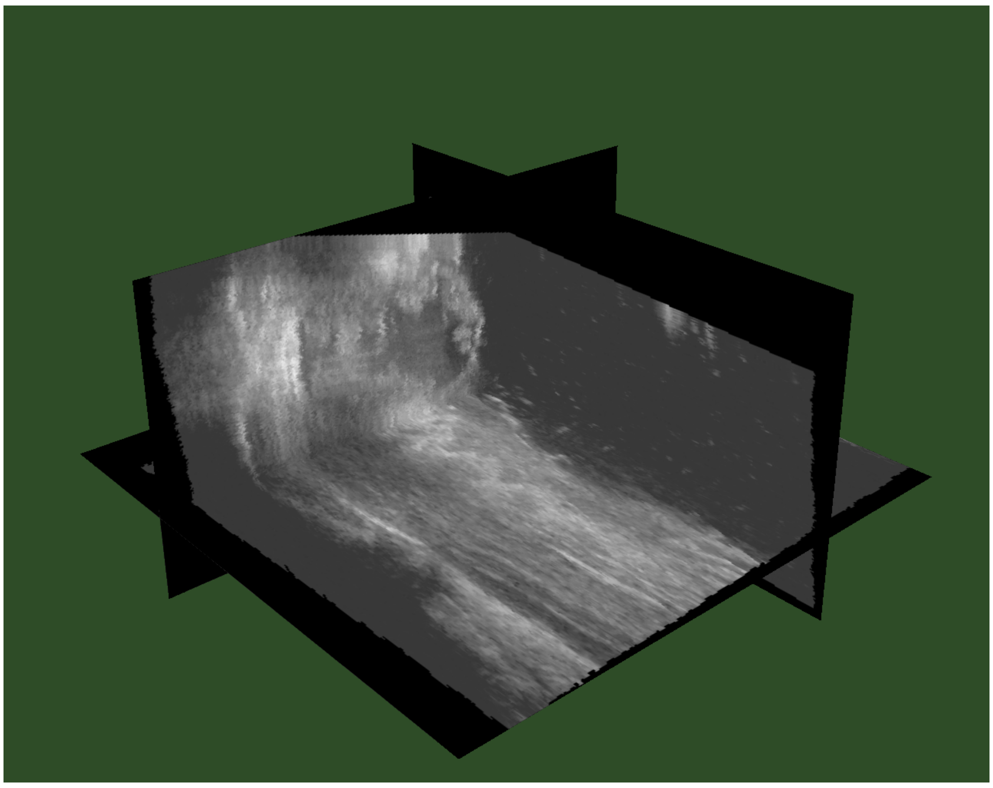


Figure B.10: Orthogonal MPR slices screenshot 2

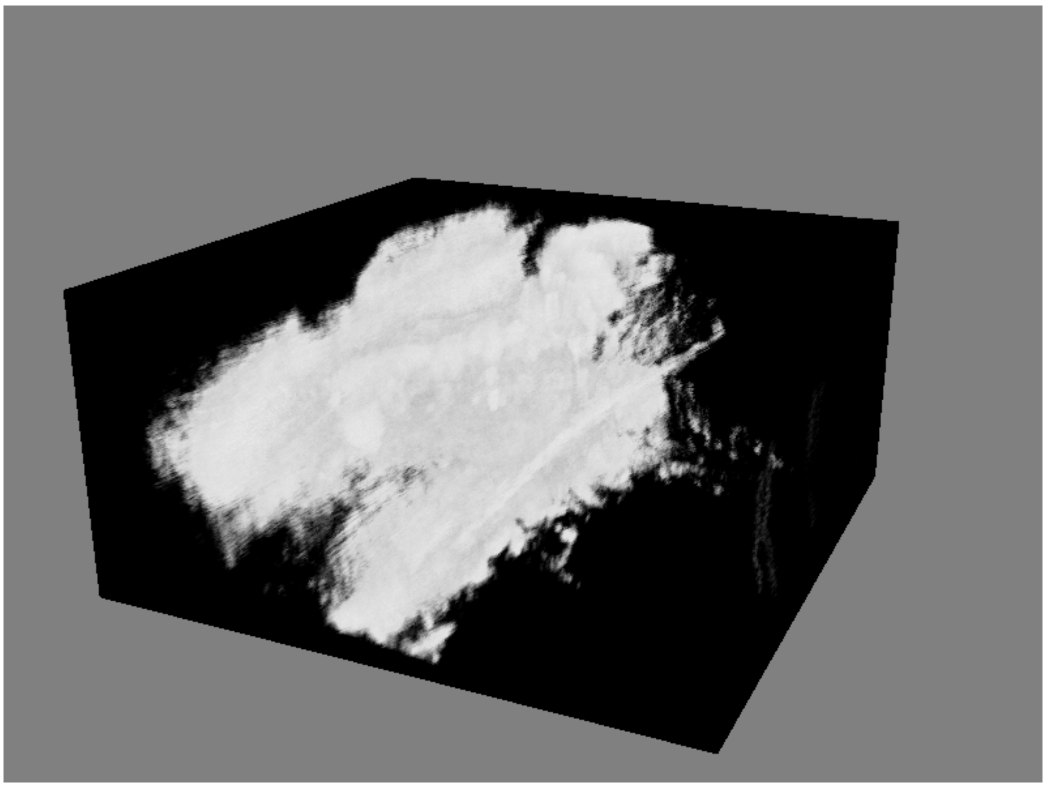


Figure B.11: Ray casting screenshot 1

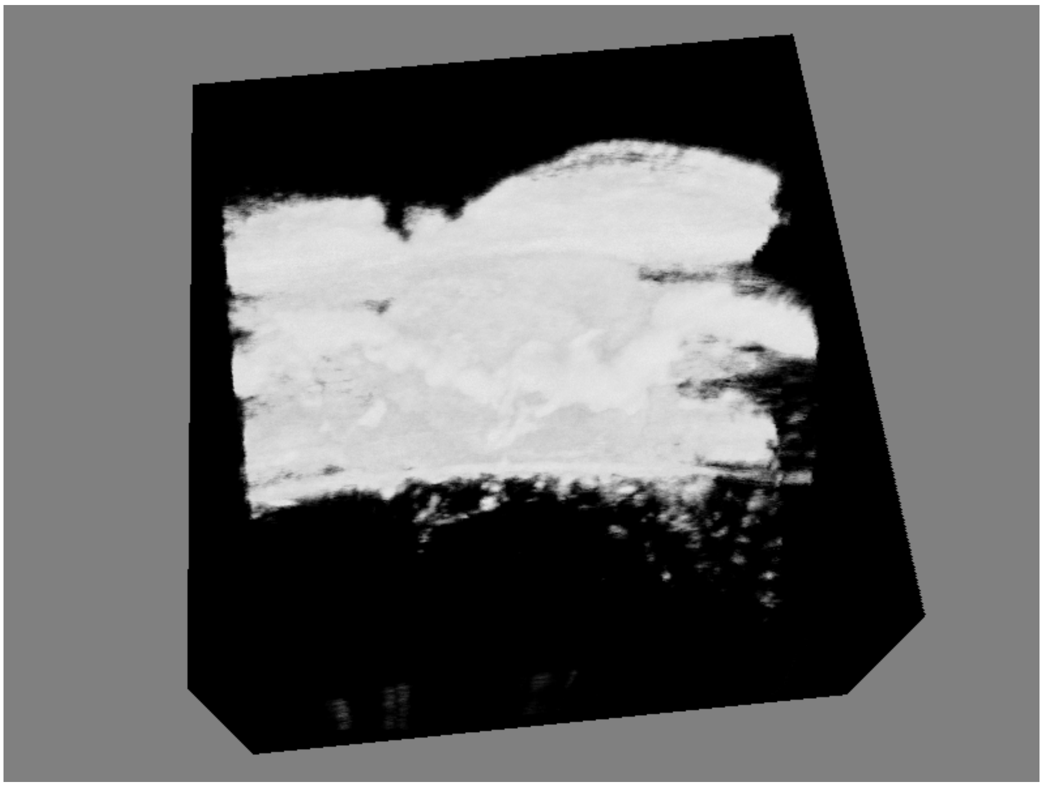


Figure B.12: Ray casting screenshot 2

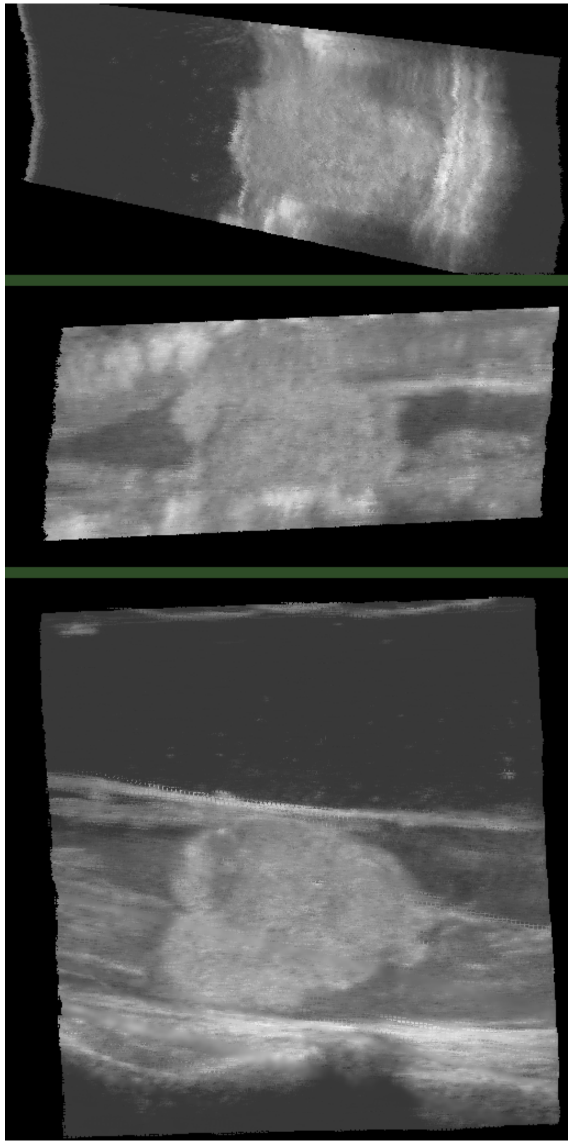


Figure B.13: Volume reconstructed with PNN

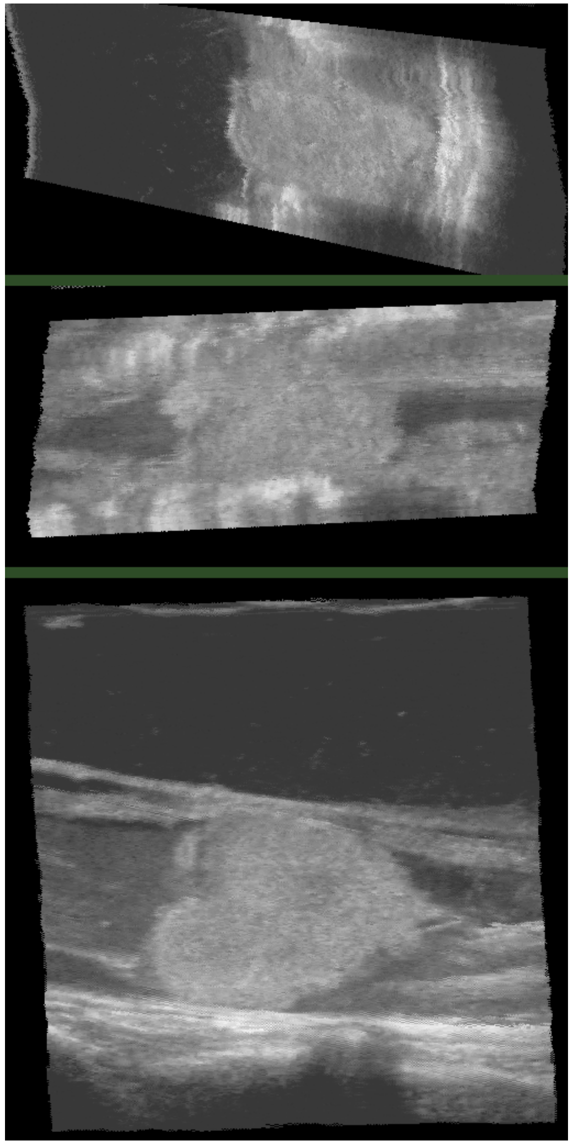


Figure B.14: Volume reconstructed with VNN

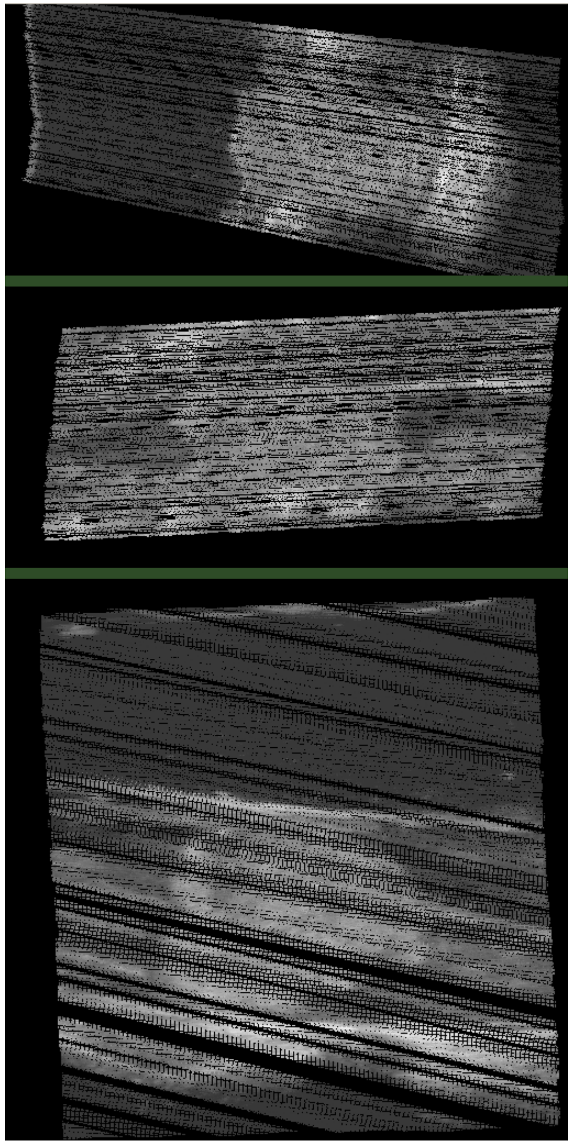


Figure B.15: Volume reconstructed with PNN without hole filling

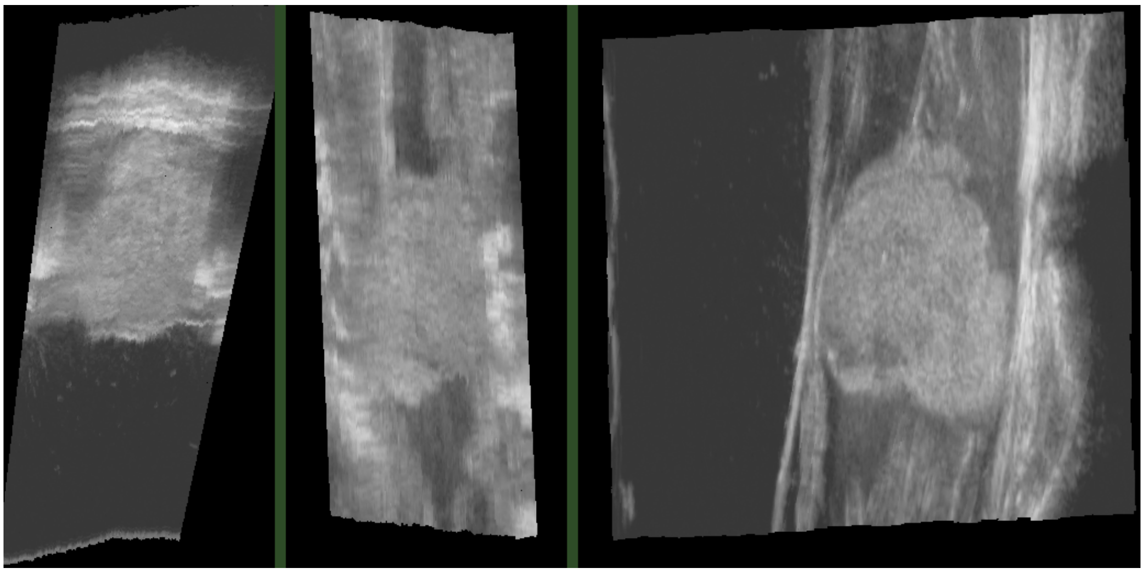


Figure B.16: Volume reconstructed with $DWOP_4$

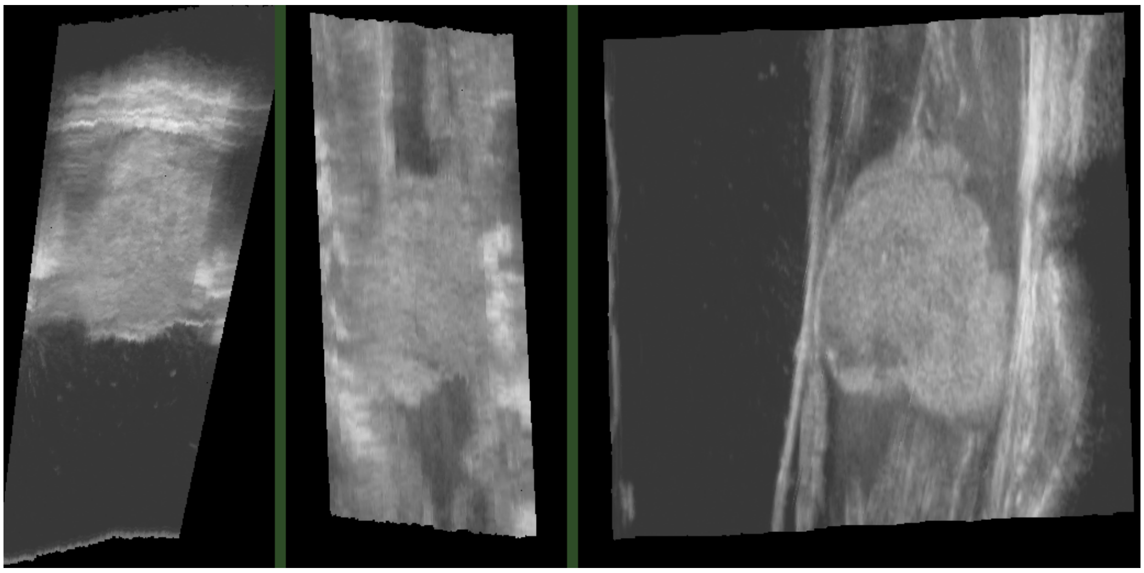


Figure B.17: Volume reconstructed with $DWOP_8$

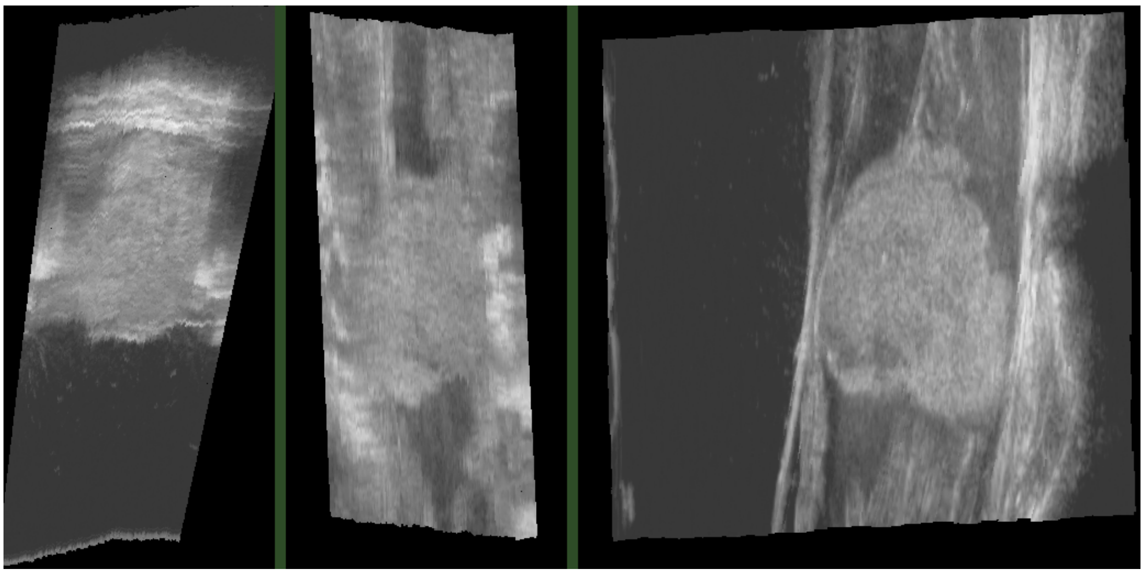


Figure B.18: Volume reconstructed with PT

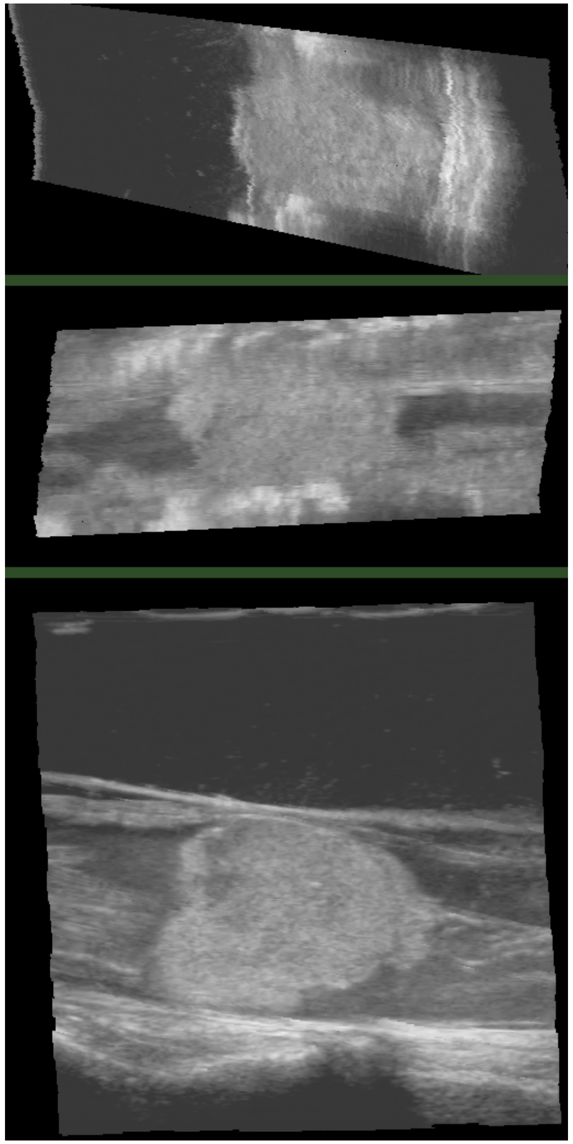


Figure B.19: Result of *overwrite* compounding method

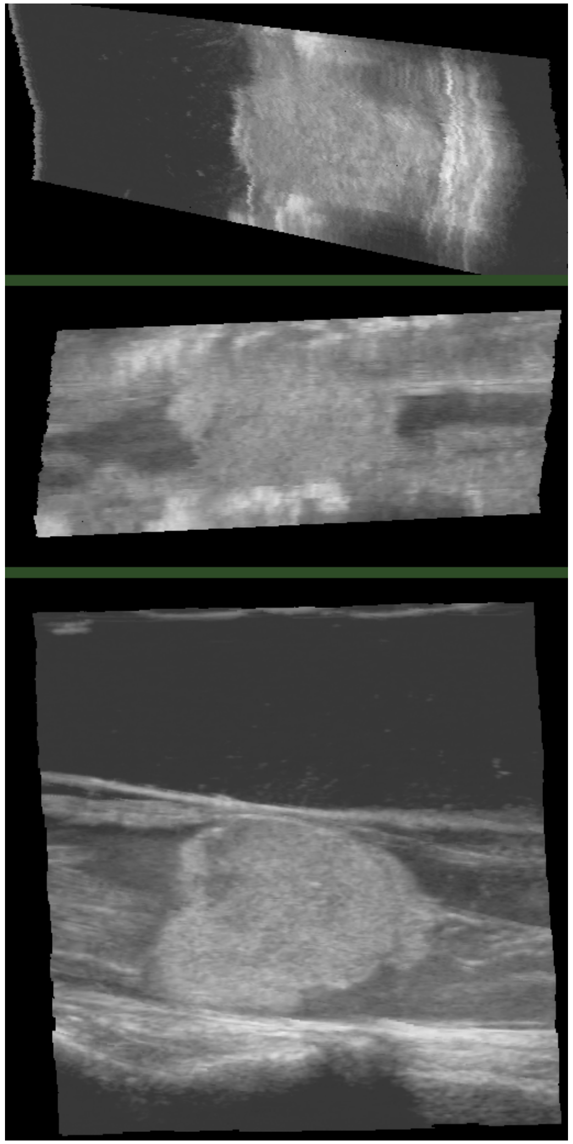


Figure B.20: Result of *avg* compounding method

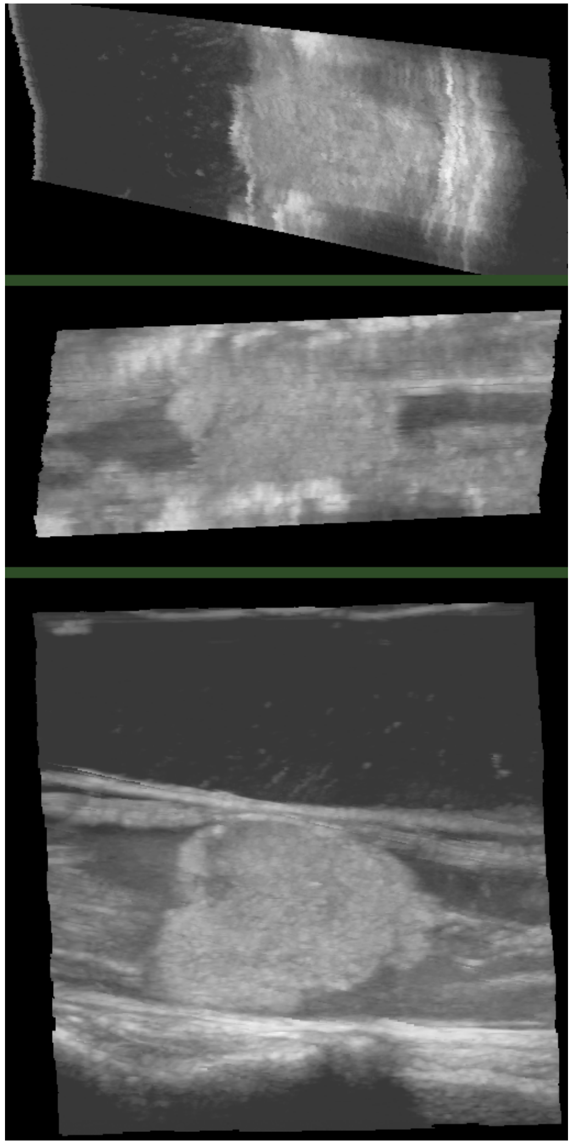


Figure B.21: Result of *max* compounding method

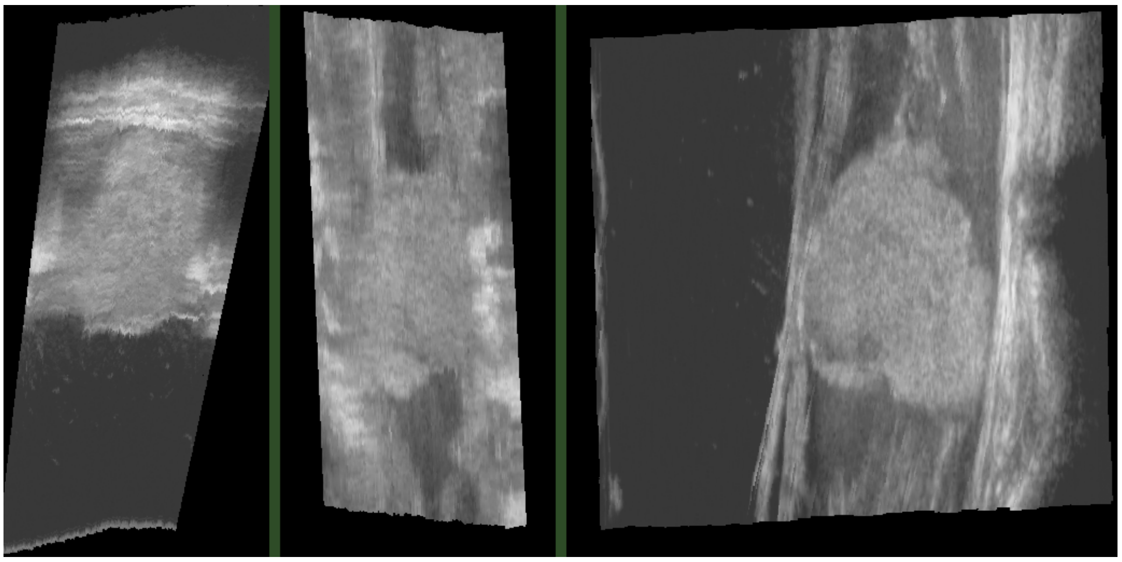


Figure B.22: Result of $ifempty$ compounding method

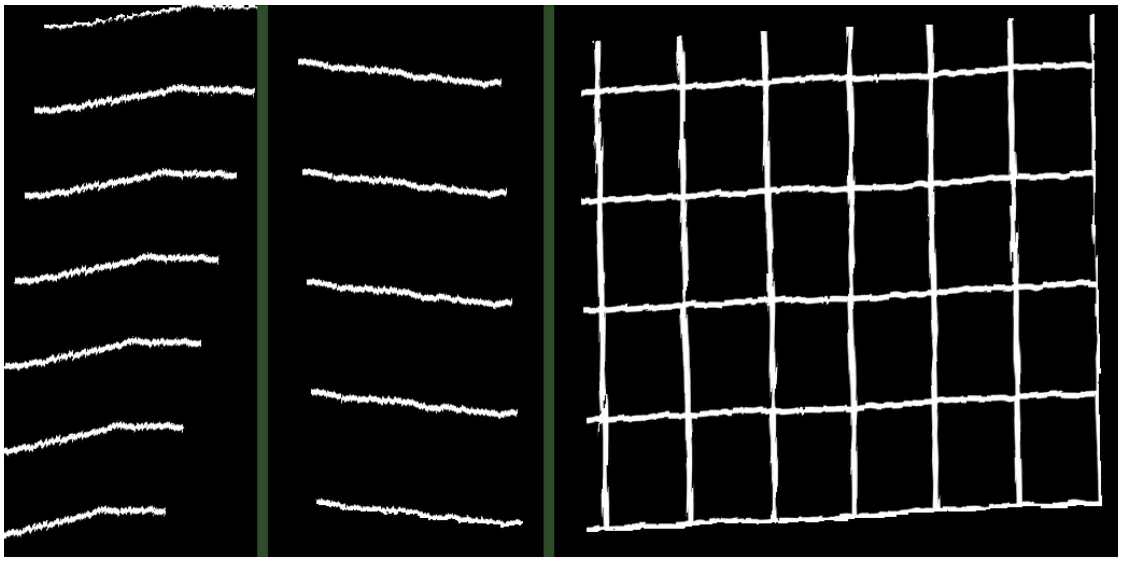


Figure B.23: Reconstructed volume of lines with real tracking data

Appendix C

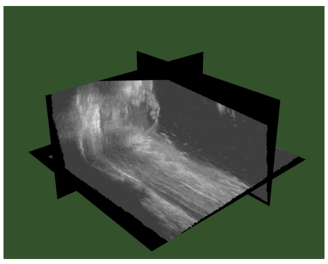
Poster

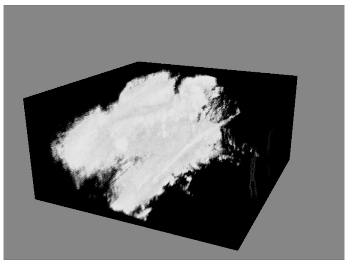
On the next page, one can find a poster from this thesis with focus on the incremental reconstruction. The poster summarizes the main principles of the methods and the most important results.

Real-Time GPU-Based 3D Ultrasound Reconstruction

Holger Ludvigsen Supervisors: Dr. Anne C. Elster, IDI, NTNU & Dr. Frank Lindseth, SINTEF

Ultrasound scanning is frequently used in medical practice because it is a non-invasive, safe and low-cost solution, but convential probes only provide 2D scans. Ultrasound reconstruction is to process such scans (*b-scans*) into 3D volumes of patient internals. The volume can be used for acquiring out-of-angle views, 3D rendering of the anatomy and image guided surgery. Being able to reconstruct in *real-time* as the data is acquired incrementally means that on can rescan areas of interest as observed on simultaneous real-time visualization.





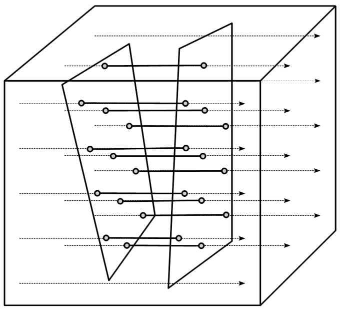


Fig. 1: Finding voxels between two ultrasound scans

The essence of the method is to transform incoming b-scan planes into the volume, an then only process voxels located *between* the b-scans. For each incoming b-scan, rays are constructed along columns of voxels and used for ray-plane intersection calculations with the current and previous b-scan planes (Fig. 1). To fill the voxels between the intersections, voxel-based interpolation methods can be used, and one alternative is based on distance weighted orthogonal projections [1] (DWOP), and a second one is based on cubic interpolation of the probe trajectory [2] (PT).

In DWOP, orthogonal projections are made of each voxel onto nearby b-scans. The pixel value at the projected points are weighted by the distance to the b-scan plane (Eq. 1).

$$voxel_j = \frac{\sum (pixel_i \cdot distance_i)}{\sum distance_i}$$

Eq. 1: Distance Weighted Orthogonal Projections

In PT, a virtual b-scan is created in the middle of four b-scans surrounding the voxel. The virtual b-scan is evaluated by cubic interpolation (Fig. 3), and the pixel values of the four interpolated b-scans are then weighted by orthogonal distance.

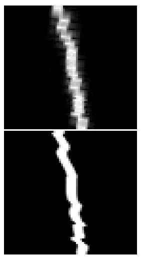


Fig. 2: Result of reconstructing straight lines (top: DWOP, bottom: PT)

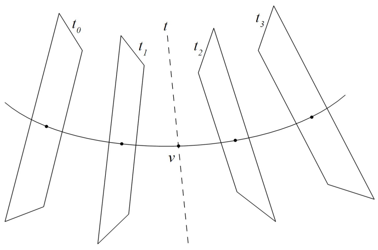


Fig. 3: Cubic interpolation for PT method

By utilizing the new Fermi GPU architecture by Nvidia, the incremental reconstruction is performed is real-time as the data is acquired as shown in Fig. 5. With 434 tracked b-scans. As for quality, the PT method demonstrated the best results (Fig. 2). The main bottleneck for performance is the overhead associated with data transfer between device and host for each increment.

This work shows how the parallel nature of ultrasound reconstruction can be exploited for real-time performance, and the techniques developed can benefit all who want to utilize the power of the GPU for reconstruction.

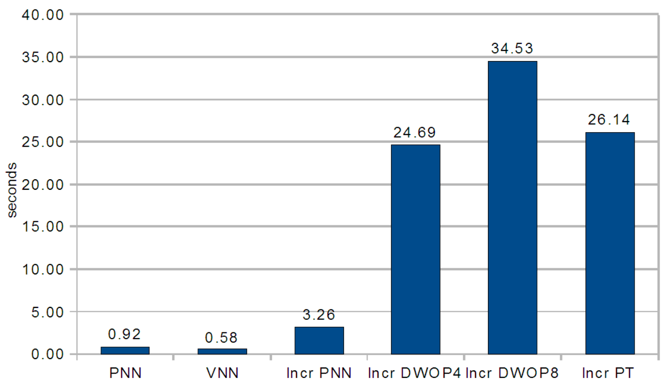


Fig. 5: Performance on Fermi (Tesla C2050)

[1] Trobaugh, J. W., Trobaugh, D. J., and Richard, W. D. Three-dimensional imaging with stereotactic ultrasonography. Computerized Medical Imaging and Graphics, 1994

[2] Coupe, P., Hellier, P., Azzabou, N., and Barillot, C. 3D Freehand Ultrasound Reconstruction Based on Probe Trajectory, Lecture Notes in Computer Science, 2005

Acknowledgements
We would like to thank Nvidia and AMD contributing GPU hardware.

Appendix D

Additional Test Measurements

On the next page, one can find additional numerical measurements taken during testing. These include performance speedups of C2050 compared to CPU and FX5800, and of HD5870 compared to FX5800, as well as core occupancy numbers, register use and global read and write throughput on the GPU.

	CPU	C2050/CPU speedup	C2050	C2050/FX5800 speedup	FX5800	HD5870/FX5800 speedup	HD5870
PNN	12.77	13.88	0.92	1.63	1.50	0.17	8.72
VNN	29.61	51.05	0.58	2.47	1.43	2.27	0.63
Incr PNN	N/A	N/A	3.26	1.01	3.28	0.49	6.67
Incr DWOP4	150.74	6.11	24.69	1.23	30.25	0.49	61.12
Incr DWOP8	255.44	7.40	34.53	1.33	45.83	0.48	95.21
Incr PT	181.48	6.94	26.14	1.18	30.90	0.55	56.43
100 ray casts	N/A	N/A	2.81	2.40	6.75	0.48	14.05

PNN	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	1.36%	N/A	N/A	N/A	N/A
memcpyDtoHasync	0.45%	N/A	N/A	N/A	N/A
fill_pixel_ill_pos	42.51%	0.17	16	4.27	414.00
fill_holes	31.75%	1	18	74.07	408.88
fill_volume	12.87%	1	11	14.14	105.29
round_off_translate	6.23%	1	11	15.92	110.05
transform	4.78%	1	17	22.30	185.64

VNN	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	2.16%	N/A	N/A	N/A	N/A
memcpyDtoHasync	0.72%	N/A	N/A	N/A	N/A
vnn	97.11%	0.5	31	56.92	126.53

Incr PNN	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	12.22%	N/A	N/A	N/A	N/A
memcpyDtoHasync	23.05%	N/A	N/A	N/A	N/A
fill_volume	27.96%	1	6	10.07	70.48
transform	26.90%	1	13	16.57	71.52
round_off_translate	9.85%	1	3	27.15	163.35

Incr PT	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	0.45%	N/A	N/A	N/A	N/A
memcpyDtoHasync	41.17%	N/A	N/A	N/A	N/A
adv_fill_voxels	57.23%	0.25	49	43.96	27.92
trace_intersections	1.13%	1	14	5.68	153.98

Incr DWOP4	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	0.46%	N/A	N/A	N/A	N/A
memcpyDtoHasync	42.02%	N/A	N/A	N/A	N/A
adv_fill_voxels	56.34%	0.25	33	44.27	85.94
trace_intersections	1.16%	1	14	5.69	154.66

Incr DWOP8	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	1.08%	N/A	N/A	N/A	N/A
memcpyDtoHasync	26.92%	N/A	N/A	N/A	N/A
adv_fill_voxels	71.24%	0.25	33	44.07	87.04
trace_intersections	0.74%	1	14	5.69	154.44

100 ray casts	**Time**	**Occupancy**	**Registers per work item**	**glob mem read throughput**	**glob mem write throughput**
memcpyHtDasync	0.01%	N/A	N/A	N/A	N/A
memcpyDtoHasync	0.01%	N/A	N/A	N/A	N/A
build_ray_dirs	0.01%	0.75	20	0	423.14
cast_rays	0.99%	1	16	14.44	65.39

Appendix E

Real-Time Ray Tracing Using Nvidia OptiX

On the next pages, the paper *Real-Time Ray Tracing Using Nvidia OptiX* [19] is included. This paper was written by this thesis' authors about their experience with Nvidia's OptiX library for GPU ray tracing. An implementation of volume casting was included as an example of the capabilities of OptiX, and Figure 2.6 in this thesis was generated from this implementation.

The paper was submitted to and accepted by the 2010 Eurographics conference where the authors held a presentation.

Real-Time Ray Tracing Using Nvidia OptiX

H. Ludvigsen[1] and A. C. Elster[1]

[1]Dept. of Computer and Info. Science, Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Modern GPUs with their several hundred cores and more accessible programming models are becoming attractive devices for compute-intensive applications. They are particularly well suited for applications, such as image processing, where the end result is intended to be displayed via the graphics card. One of the more versatile and powerful graphics techniques is ray tracing. However, tracing each ray of light in a scene is very computational expensive and have traditionally been preprocessed on CPUs over hours, if not days. In this paper, Nvidia's new OptiX ray tracing engine is used to show how the power of modern graphics cards, such as the Nvidia Quadro FX 5800, can be harnessed to ray trace several scenes that represent real-life applications in real-time speeds ranging from 20.63 to 67.15 fps. Near-perfect speedup is demonstrated on dual GPUs for scenes with complex geometries. The impact on ray tracing of the recently announced Nvidia Fermi processor, is also discussed.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Ray tracing makes it possible to render realistic shadows, reflections and glass-like objects, which requires trickery when only rasterization is used. In rasterization, one computes the area on the screen where each object is to be shown, but do not analyze light's impact on the scene. Ray tracing, however, provides this features "by nature", because it approximates how light actually behaves. The main drawback of ray tracing is its computational complexity. The computations have traditionally been done on the CPU, but modern graphical processing units (GPUs) with their several hundred cores and now also more accessible programming models, are attractive devices for compute-intensive applications. By off-loading the ray tracing calculations to a modern GPU, ray tracing is becomng viable for computer games and real-time visualizations. In addition, ray tracing has applications in optical and acoustical design, radiation research, volume calculations and collision analysis.

Ray tracing is parallelizable because each ray may be traced independently. There is typically one ray per pixel, so common 1024 x 1024 pixel images would require tracing 10^6 rays. Ray tracing is hence a very attractive application for the massively parallel newer GPUs. Nvidia thus recently (Sept. 2009) released OptiX, a ray tracing engine for their Quadro and Tesla GPUs. This paper describes our initial experiences with this engine for real-time ray tracing.

Traditionally, ray traced images are computed a priori. This process might take a couple of minutes or several days per rendered image. Real-time ray tracing (> 20 fps) facilitates that realistic graphics can be manipulated interactively, like in a computer game. Similarly, feedback could be given instantly during optical design, radiation research and the other areas mentioned above. Last, but not least, the realistic visual effects possible by ray tracing could be added to games and other real-time visualization applications.

2. Previous work related to real-time ray tracing

Both CPU and customized hardware were used until recently for real-time ray tracing. However, the performance obtained in both cases have not been satisfactory compared to rasterization. Wald et al. [WSBW01] presented a highly optimized CPU implementation where the algorithms take advantage of caches, SIMD instructions and coherence in image and object space. Their implementation outperformed the earlier ray tracers, and even rasterization with graphics hardware for complex scenes. In their simplest scene with 40 thou-

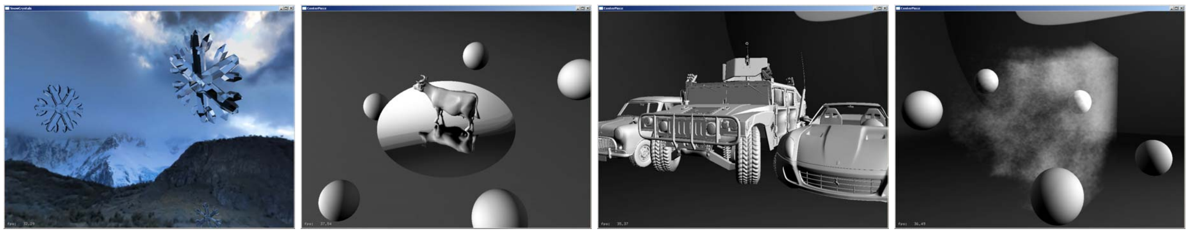


Figure 1: *Screenshots from the snow crystals scene and three centerpiece modes implemented in OptiX*

sand triangles, they obtained 1.8 fps at a resolution of 512 x 512 pixels on an 800 MHz Pentium III.

A different approach was used by Woop et al. [WSS05], who developed a programmable ray processing unit (RPU) chip specialized for real-time ray tracing. At only 66 Mhz, their prototype was capable of rendering a simple scene of 806 triangles at 21 fps, and a highly complex scene of 187 million triangles obtained 4 fps. However, this was at the modest resolution of 512 x 384.

In the recent years, attempts have been made to do ray tracing on the GPU, and the results have been promising. Purcell et al. [PBMH05] explained how ray tracing can be mapped to programmable graphics hardware and used a simulator to analyze the performance one might obtain on future graphics hardware. Their conclusion was that graphics hardware indeed look promising. Gunther et al. [GPSS07] followed up on this work by presenting a GPU ray tracer using optimized BVH-strctures to obtain 13.6 fps in a 2 million triangle scene at 1024 x 1024 pixels using an Nvidia Geforce 8800 GTX.

Some of the most recent work have obtained *true* real-time performance using commodity GPUs. Shih et al. [SCCC09] implemented a high performance CUDA-based ray tracer and obtained 30 to 43 fps in scenes ranging from 66 to 871 thousand triangles at 1024 x 1024 pixels on the older Nvidia Geforce 8800 GTS. Aila and Laine (Nvidia Research) [AL09] have developed a CUDA-based ray tracer which is hand-optimized at the assembly level and pushed performance towards (their) theoretical limit. Using the newer Nvidia Geforce GTX285, they obtained from 75 to 142 millions of primary rays per second, which at a resolution of 1024 x 768 correspond to 95 to 180 fps. Modern GPUs may also be used to do real-time implicit surfaces rendering [SN10]. Singh and Narayanan [SN10] also include several other recent related references.

3. Nvidia OptiX

OptiX [Nvi09c] is a recent programmable ray tracing engine that runs on top of Nvidia CUDA [Nvi09a]. It is a set of library functions for both graphics rendering or other applications that trace rays. The OptiX engine currently only runs on newer (GT200 core) Nvidia Quadro and Tesla cards. To use OptiX, the programmer writes *programs* that handle the various events of the ray tracing. These programs are really CUDA kernels, but are called programs in OptiX terminology. The events they handle include ray generation, ray hit, ray miss, etc. In the host code, the API is set up through a *context* structure that holds the configuration and components of the ray tracing. Such components include the previously mentioned programs, geometry that the rays hit and materials that define the surface properties of the geometry.

4. Our implementations using OptiX

Two ray traced scenes have been implemented, where one of them has several modes that each demonstrate how OptiX handles different applications of ray tracing. Some screenshots are given in Figure 1. Our snow crystals scene was implemented from scratch with transparent snow crystals which fall slowly across the screen. Our centerpiece scene depicts a centered object that changes with different modes of the scene. The object used include a cow model that comes with the OptiX SDK. The camera rotates around the centerpieces. Possible modes of the centerpiece scene are:

1. Phong shaded cow model and spheres
2. Reflective cow model and Phong shaded spheres
3. Reflective cow model and glass spheres
4. Cow model with diffuse reflection on floor
5. Cow model with diffuse shadow on floor
6. High definition car models (1 million triangle polygons)
7. Voxel map of cloud fractal

To compare our results to previous work on real-time GPU ray tracing, the polygon meshes from [SCCC09] and [AL09] were obtained, and OptiX used to ray trace them. The camera angle was adjusted to be approximately equal to the camera angles used in screenshots given in [SCCC09] and [AL09]. Note, however, that the results in [SCCC09] are on older hardware, so in this case, our results are as much about what scenes one can implement efficiently rather than fair comparisons. Screenshots of some of the scenes as rendered in the test bench, are given in Figure 4.

Currently most PCs support up to two GPU cards. We hence also tested OptiX with two identical Nvidia Quadro FX 5800 GPUs. The performance in fps was measured for

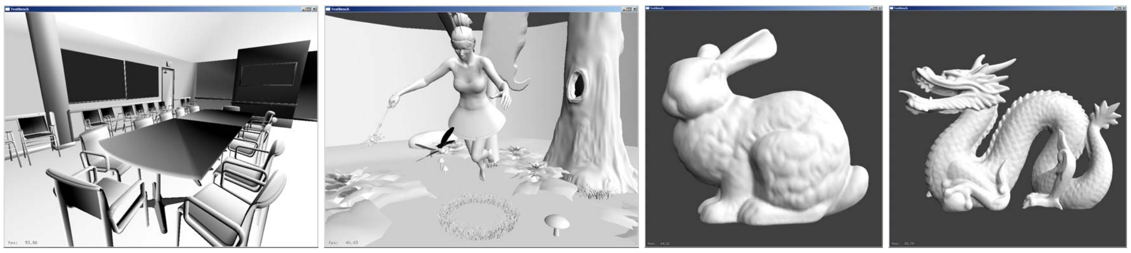


Figure 2: *Screenshots of conference, fairy, bunny and dragon scene from OptiX test bench*

some of the scenes that come with the OptiX SDK in addition to the scenes implemented in this project. The speedup was calculated as the ratio between performance with 2 and 1 GPUs.

All OptiX testing was done on a system with:

- GPU: Nvidia Quadro FX 5800
- CPU: Intel Core 2 Quad Q9550 2.83 GHz
- Memory: 4 x Corsair 2 GB DDR3 1333 MHz
- OS: Microsoft Windows XP 64 bit
- Compiler: Microsoft Visual Studio 2008

For all scenes, the performance was measured over 100 frames after a 3 second warm-up. The inverse of the average frame render time gives the fps. Table 1 summarizes the performance of the snow crystals and centerpiece scenes with its modes.

Table 1: *Performance in fps at 1024 x 768 pixels*

Scene	fps
Snow crystals	22.10
Centerpiece Phong	67.51
Centerpiece reflective	38.73
Centerpiece reflective and refractive	24.41
Centerpiece diffuse reflection	23.10
Centerpiece diffuse shadows	25.32
Centerpiece 1 million polygons	24.99
Centerpiece voxels	22.08

The main issue faced in our snow crystals scene is refraction and reflection of rays when they hit the snow crystals. At the initial intersection the ray is branched into two rays. And when the refracted ray hits exits the crystal, there is another branching into two rays. This branching imposes a performance penalty, especially when the crystals cover much of the screen area. The performance obtained is real-time with an average of 22 fps. A major problem is how the fps varies depending on what happens in the scene. When a large crystal is close to the screen, the fps is low at around 20 fps. When this crystal exits the screen, the fps spikes up to around 40. This behaviour results in unstable performance, and is one of the major drawbacks of the ray tracing algorithm.

In our centerpiece scene, the performance varies for each of the modes. All of the modes offer real-time performance at 22-25 fps, but the cheaper Phong shading and reflection only mode results in 67.51 and 38.73 fps, respectively. Diffuse reflection and diffuse shadows were a disappointment performance-wise. In both cases the ray branching is set to only into 4 new rays, but the fps is nevertheless barely real-time. This shows how ray branching is the major challenge of ray tracing performance. However, the scene consisting of three car models and a scooter which has a total of 1 million polygons, shows that OptiX is indeed capable of rendering real-time scenes with high definition and complex models. All representable for real-life objects. The voxel mode has "only" 250,047 voxels, but does not benefit from primitives being covered by other primitives such as in the mode with 1 million polygons. At an fps of 22.08, this shows that OptiX is capable of rendering voxels scenes in real-time.

4.1. Performance vs. optimized GPU ray-tracers

Table 2 shows the number of triangle polygons and measured OptiX performance of scenes from [AL09], and Table 3 shows the same for scenes from [SCCC09]. Also shown in these tables is the performance the authors of [AL09] and [SCCC09] obtained with their ray tracer.

Table 2: *Triangles and performance in Mray/s of scenes in [AL09] at 1024 x 768 pixels*

Scene	Conference	Fairy	Sibenik
Triangle polygons	282,759	174,117	80,133
Mray/s OptiX	28.22	20.69	38.12
Mray/s [AL09]	142.2	74.6	117.5

Table 3: *Triangles and performance in fps of scenes in [SCCC09] at 1024 x 1024 pixels*

Scene	Bunny	Sponza	Dragon
Triangle polygons	69,451	66,454	871,414
Fps OptiX	49.89	28.16	36.43
Fps [SCCC09]	45.30	42.47	31.88

As seen in Table 2, our results using OptiX are 3-4 times slower than the implementation in [AL09]. The hardware in both cases is the GT200 generation GPU. This shows that

OptiX has potential for much higher performance. An explanation of the discrepancy can be the flexibility of OptiX, and the fact that the implementations in [AL09] was hand optimized at the assembly level for performance only.

Table 3 shows that our OptiX implementations outperform [SCCC09] slightly in the bunny and dragon scene, but lags behind by about 30 % in the sponza scene. However, the GPU used in [SCCC09] is a Nvidia Geforce 8800 GTS that is several generations older and substantially slower than the Quadro FX 5800 used in our test bench. Again, one would assume that the implementation in [SCCC09] is heavily optimized and not as flexible.

4.2. Multiple GPUs

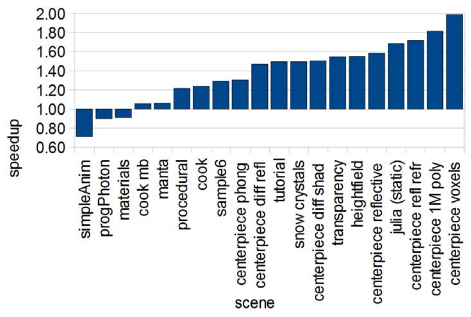


Figure 3: *Multiple GPU speedup in various scenes*

Figure 3 compares our results on 2 GPUs with 1 GPU in various scenes. At the lower extreme, there is actually a *slowdown* compared to using 1 GPU. However, at the other end of the chart, we achieve an almost perfect 200 % speedup. Common for the scenes that speed up well is that they spend a lot of time on GPU computations compared to other tasks such as image display, data transfer and CPU computations. Hence in order to take advantage of the computational power of the GPU, scenes need to have enough computational complexity that can be done on the GPU, as can be seen in our centerpiece scene.

5. Conclusions and future work

This paper studied implementations of animated real-time scenes that represent actual real-life application areas for ray tracing. Nvidia's recently released OptiX ray tracing engine allows users to harness the power of modern GPUs. Our results demonstrate that several ray tracing applications may be performed in real-time on the GPU using OptiX. All of our test cases gave real-time speeds ranging from 20.63 to 67.51 fps on 1 GPU. Our dual GPU results indicated that OptiX can give near-perfect speedup on multiple GPUs for scenes with enough computational complexity. Even though OptiX has showed to be capable of real-time ray tracing, our initial implementations were slower (3 to 5 times) than some hand optimized ray tracers such as in [AL09], indicating room for improvement. Our results do, however, demonstrate that OptiX is a flexible engine capable of real-time ray tracing on both single and multiple Nvidia GPUs.

A major difference between CPUs and GPUs is that GPUs cannot do branching efficiently. Efficient branching is important in ray tracing since the directions the rays are reflected and refracted is not known in advance. Fortunately, newer GPUs such as the Nvidia CUDA architecture handle branching better than previous generations. NVIDIA recently announced their new Fermi [Nvi09b] GPU which includes L1 and L2 cache, better double precision number support and concurrent kernel execution. The on-chip GPU cache should be beneficial for ray tracing since previously read or spatially coherent data can then be quickly accessed when traversing acceleration structures. When it is publicly available, its impact on OptiX and real-time ray tracing performance should be investigated.

Future ray tracer designs should also incorporate the ideas from recent work such as [AL09] and [SCCC09]. Incorporating ray tracing into full-scale applications such as medical and seismic visualizations, should also be investigated. Finally, we would like to thank Nvidia and other sponsors of our HPC-lab.

References

[AL09] Aila T., Laine S.: Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 145–149. 2, 3, 4

[GPSS07] Gunther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing on gpu with bvh-based packet traversal. *Symposium on Interactive Ray Tracing 0* (2007), 113–118. 2

[Nvi09a] Nvidia Corporation: *CUDA Programming Guide version 2.3.1*, August 2009. 2

[Nvi09b] Nvidia Corporation: *Fermi Compute Architecture Whitepaper*, 2009. 4

[Nvi09c] Nvidia Corporation: *OptiX Ray Tracing Engine Programming Guide version 1.0*, September 2009. 2

[PBMH05] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses* (New York, NY, USA, 2005), ACM, p. 268. 2

[SCCC09] Shih M., Chiu Y.-F., Chen Y.-C., Chang C.-F.: Real-time ray tracing with cuda. In *ICA3PP '09: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 327–337. 2, 3, 4

[SN10] Singh J., Narayanan P.: Real-time ray tracing of implicit surfaces on the gpu. *Visualization and Computer Graphics, IEEE Transactions on 16*, 2 (march-april 2010), 261 –272. 2

[WSBW01] Wald I., Slusallek P., Benthin C., Wagner M.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum* (2001), pp. 153–164. 1

[WSS05] Woop S., Schmittler J., Slusallek P.: Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph. 24*, 3 (2005), 434–444. 2

Appendix F

Code Listings

In this appendix, one can find code listings from the most important parts of the implementation made for this thesis.

F.1 Pixel-Nearest-Neighbor Kernels

The following code listing contains the kernels for pixel-nearest-neighbor reconstruction as described in Section 4.2.

```
_kernel void round_off_translate(__global float * pixel_pos0,
                                 __global float * pixel_pos1,
                                 __global float * pixel_pos2,
                                 __global float * pixel_pos3,
                                 __global float * pixel_pos4,
                                 __global float * pixel_pos5,
                                 float volume_spacing,
                                 int mask_size,
                                 float origo_x,
                                 float origo_y,
                                 float origo_z,
                                 int bscan_n) {
  int n = get_global_id(0);
  if (n >= bscan_n*mask_size) return;

  int a = n/mask_size;
  int i = n%mask_size;

  __global float * pixel_pos[6] = {pixel_pos0, pixel_pos1,
      pixel_pos2, pixel_pos3, pixel_pos4, pixel_pos5};

  pixel_pos_c(a,i,0) = (int)((pixel_pos_c(a,i,0)-origo_x)/
      volume_spacing);
  pixel_pos_c(a,i,1) = (int)((pixel_pos_c(a,i,1)-origo_y)/
      volume_spacing);
  pixel_pos_c(a,i,2) = (int)((pixel_pos_c(a,i,2)-origo_z)/
      volume_spacing);
}

__kernel void fill_volume(__global float * pixel_pos0,
                          __global float * pixel_pos1,
                          __global float * pixel_pos2,
                          __global float * pixel_pos3,
                          __global float * pixel_pos4,
                          __global float * pixel_pos5,
                          __global unsigned char * pixel_ill,
                          int mask_size,
                          __global unsigned char * volume,
                          int volume_n,
                          int volume_h,
                          int volume_w,
                          int bscan_n) {

  int n = get_global_id(0);
  if (n > bscan_n*mask_size) return;

  int a = n/mask_size;
  int i = n%mask_size;

  __global float * pixel_pos[6] = {pixel_pos0, pixel_pos1,
      pixel_pos2, pixel_pos3, pixel_pos4, pixel_pos5};

  int x = pixel_pos_c(a,i,0);
  int y = pixel_pos_c(a,i,1);
  int z = pixel_pos_c(a,i,2);
  if (inrange(x,0,volume_w) && inrange(y,0,volume_h) && inrange(
      z,0,volume_n))
    volume_a(x,y,z) = pixel_ill[a*mask_size + i];
}

__kernel void transform(__global float * pixel_pos0,
                        __global float * pixel_pos1,
                        __global float * pixel_pos2,
                        __global float * pixel_pos3,
                        __global float * pixel_pos4,
                        __global float * pixel_pos5,
                        __global float * pos_matrices,
                        int mask_size,
                        int bscan_n) {
  int n = get_global_id(0);
  if (n >= bscan_n*mask_size) return;

  int a = n/mask_size;
  int i = n%mask_size;

  __global float * pixel_pos[6] = {pixel_pos0, pixel_pos1,
      pixel_pos2, pixel_pos3, pixel_pos4, pixel_pos5};

  float sum0, sum1, sum2;
  for (int y = 0; y < 3; y++) {
    float sum = 0;
    for (int x = 0; x < 3; x++)
      sum += pos_matrices_a(a,x,y)*pixel_pos_c(a,i,x);
    sum += pos_matrices_a(a,3,y);
    if (y==0) sum0=sum; else if (y==1) sum1=sum; else sum2=sum;
  }
  pixel_pos_c(a,i,0) = sum0; pixel_pos_c(a,i,1) = sum1;
      pixel_pos_c(a,i,2) = sum2;
}

__kernel void fill_pixel_ill_pos(__global unsigned char *
    bscans0,
                                 __global unsigned char *
                                    bscans1,
                                 __constant unsigned char * mask
                                    ,
                                 __global float * pixel_pos0,
                                 __global float * pixel_pos1,
                                 __global float * pixel_pos2,
                                 __global float * pixel_pos3,
                                 __global float * pixel_pos4,
                                 __global float * pixel_pos5,
                                 __global unsigned char *
                                    pixel_ill,
                                 int mask_size,
                                 int bscan_n,
                                 int bscan_h,
                                 int bscan_w,
                                 float bscan_spacing_x,
                                 float bscan_spacing_y) {

  int n = get_global_id(0);
  if (n >= bscan_n) return;

  __global float * pixel_pos[6] = {pixel_pos0, pixel_pos1,
      pixel_pos2, pixel_pos3, pixel_pos4, pixel_pos5};

  int mask_counter = 0;
  unsigned char foo;
  unsigned char mask_bit;
  unsigned char ill;
  for (int y = 0; y < bscan_h; y++) {
    for (int x = 0; x < bscan_w; x++) {
      foo = 1 << (y*bscan_w + x)%8;
      mask_bit = mask[(x + y*bscan_w)/8] & foo;
      if (mask_bit != 0) {
        if (n < bscan_n/2) {
          ill = bscans0[x + y*bscan_w + n*bscan_w*bscan_h];
        } else {
          ill = bscans1[x + y*bscan_w + (n-bscan_n/2)*bscan_w*
              bscan_h];
        }
        pixel_ill[n*mask_size + mask_counter] = ill;
        pixel_pos_c(n,mask_counter,0) = 0;
        pixel_pos_c(n,mask_counter,1) = x*bscan_spacing_x;
        pixel_pos_c(n,mask_counter,2) = y*bscan_spacing_y;
        mask_counter++;
      }
    }
  }
}

__kernel void fill_holes(__global unsigned char * volume,
                         int volume_n,
                         int volume_h,
                         int volume_w) {
  #define kernel_size 5
  #define half_kernel (kernel_size/2)
  #define cutoff (kernel_size*kernel_size*kernel_size/2.0f -
      half_kernel)

  int n = get_global_id(0);

  int z = n/(volume_h*volume_w) + half_kernel;
  int y = (n/volume_w)%volume_h + half_kernel;
  int x = n%volume_w + half_kernel;
  if (z >= volume_n-half_kernel || y >= volume_h-half_kernel ||
      x >= volume_w-half_kernel) return;

  if (volume_a(x,y,z) == 0) {
    int sum = 0;
    int sum_counter = 0;
    for(int i = -half_kernel; i <= half_kernel; i++)
      for(int j = -half_kernel; j <= half_kernel; j++)
        for(int k = -half_kernel; k <= half_kernel; k++)
          if (volume_a(x+i,y+j,z+k) != 0) {
            sum += volume_a(x+i,y+j,z+k);
            sum_counter++;
        }
    if (sum_counter > cutoff && sum/(float)sum_counter <= 255) volume_a(x,y,z) = sum/(float)sum_counter;
  }
}
```

F.2 Voxel-Nearest-Neighbor Kernel

The following code listing is the kernel for voxel-nearest-neighbor reconstruction as described in Section 4.3.

```
#define _distance(v, plane) (plane.x*v.x + plane.y*v.y + plane.z*v.z + plane.w)/sqrt(plane.x*plane.x + plane.y*plane.y + plane.z*plane.z)
#define plane_points_c(n,i) (plane_points[(n)*3 + (i)])

__kernel void vnn(__global unsigned char * bscans0,
                  __global unsigned char * bscans1,
                  __global unsigned char * mask,
                  int bscan_w,
                  int bscan_h,
                  int bscan_n,
                  float bscan_spacing_x,
                  float bscan_spacing_y,
                  __global unsigned char * volume,
                  int volume_n,
                  int volume_h,
                  int volume_w,
                  float volume_spacing,
                  __global float4 * plane_eq,
                  __global float4 * plane_points
                  ) {
  int id = get_global_id(0);

  int z = id/volume_w;
  int x = id%volume_w;
  if (z >= volume_n) return;
  if (x >= volume_w) return;

  #define kernel_radius (volume_spacing*5)

  int current_bscan = -1;
  float dist = 10000;
  for (int n = 0; n < bscan_n; n++) {
    float4 voxel_000 = {x*volume_spacing, 0.0f, z*volume_spacing, 0.0f};
    float temp = fabs(_distance(voxel_000, plane_eq[n]));
    if (temp < dist) {
      current_bscan = n;
      dist = temp;
    }
  }

  int print_counter = 0;

  for (int y = 0; y < volume_h; y++) {
    float4 voxel_coord = {x*volume_spacing, y*volume_spacing, z*
        volume_spacing, 0.0f};
    dist = 10000;
    float temp;
    int done_up = 0;
    int done_down = 0;
    int min_bscan = current_bscan;
    int n;
    for (int i = 1; !done_down || !done_up; i++) {
      if (!done_up) {
        n = current_bscan + i;
        temp = fabs(_distance(voxel_coord, plane_eq[n]));
        min_bscan = temp < dist ? n : min_bscan;
        dist = min(dist, temp);
        done_up = temp-dist > kernel_radius || n >= bscan_n-1;
      }

      if (!done_down) {
        n = current_bscan - i;
        temp = fabs(_distance(voxel_coord, plane_eq[n]));
        min_bscan = temp < dist ? n : min_bscan;
        dist = min(dist, temp);
        done_down = temp-dist > kernel_radius || n <= 0;
      }
    }
    current_bscan = min_bscan;
    dist = _distance(voxel_coord, plane_eq[current_bscan]);

    float4 normal = {plane_eq[current_bscan].x, plane_eq[
        current_bscan].y, plane_eq[current_bscan].z, 0.0f};
    float4 corner0 = plane_points_c(current_bscan,0);
    float4 cornerx = plane_points_c(current_bscan,1);
    float4 cornery = plane_points_c(current_bscan,2);

    float4 p = voxel_coord + -dist*normal - corner0;
    float4 x_vector = normalize(cornerx - corner0);
    float4 y_vector = normalize(cornery - corner0);

    int px = dot(p, x_vector)/bscan_spacing_x;
    int py = dot(p, y_vector)/bscan_spacing_y;

    if (px >= 0 && px < bscan_w && py >= 0 && py < bscan_h)
      if (fabs(dist) < kernel_radius)

        if (mask[px + py*bscan_w] != 0)
          if (current_bscan < bscan_n/2) {
            volume_a(x,y,z) = bscans0[px + py*bscan_w +
                current_bscan*bscan_w*bscan_h];
          } else {
            volume_a(x,y,z) = bscans1[px + py*bscan_w + (
                current_bscan-bscan_n/2)*bscan_w*bscan_h];
          }
  }
}
```

F.3 Ray Casting Kernels

The following code listing contains the kernels for ray casting as described in Section 5.4.

```
__kernel void build_ray_dirs(__global unsigned char * volume,
                              int volume_w,
                              int volume_h,
                              int volume_n,
                              __global float4 * ray_dirs,
                              int bitmap_w,
                              int bitmap_h,
                              float4 camera_pos,
                              float4 camera_lookat,
                              __global float * printings) {
  int n = get_global_id(0);
  if (n >= bitmap_w*bitmap_h) return;

  int ray_x = n%bitmap_w;
  int ray_y = n/bitmap_w;

  float4 camera_forward = normalize(camera_lookat - camera_pos);
  float4 temp_up = {0, 1, 0, 0};
  float4 camera_right = normalize(cross(temp_up, camera_forward)
      );
  float4 camera_up = normalize(cross(camera_right,
      camera_forward));

  float fov_hor = 45/2;
  float fov_ver = fov_hor*bitmap_h/(float)bitmap_w;
  fov_hor = fov_hor/180.0f*3.14f;
  fov_ver = fov_ver/180.0f*3.14f;

  float4 step_forward = camera_forward;
  float temp = (ray_x-bitmap_w/2)/(float)(bitmap_w/2);
  float4 step_right = temp * fov_hor * camera_right;
  temp = (ray_y-bitmap_h/2)/(float)(bitmap_h/2);
  float4 step_up = temp * fov_ver * camera_up;
  float4 ray_dir = normalize(step_forward + step_right + step_up
     );

  ray_dirs[n] = ray_dir;
}

__kernel void cast_rays(__global unsigned char * volume,
                        int volume_w,
                        int volume_h,
                        int volume_n,
                        __global float4 * ray_dirs,
                        __global unsigned char * bitmap,
                        int bitmap_w,
                        int bitmap_h,
                        float4 camera_pos,
                        float4 camera_lookat,
                        __global float * printings) {
  int n = get_global_id(0);
  if (n >= bitmap_w*bitmap_h) return;

  int ray_x = n%bitmap_w;
  int ray_y = n/bitmap_w;

  float4 ray_dir = ray_dirs[n];

  #define step_size 1.0f
  #define transparent_level 54
  #define transparency_ajustment 0.4f
  #define ray_strength_cutoff (255/10.0f)

  unsigned char accum = 0;
  float t = 0;

  float4 volume_0 = {0, 0, 0, 0};
  float4 volume_1 = {volume_w-1, volume_h-1, volume_n-1, 0};

  float4 foo0 = (volume_0 - camera_pos)/ray_dir;
  float4 foo1 = (volume_1 - camera_pos)/ray_dir;
  foo0 = min(foo0, foo1);
  t = max(foo0.x, max(foo0.y, foo0.z)) + 2;

  float4 t_pos = camera_pos + t*ray_dir;

  if (t_pos.x > 0 && t_pos.x < volume_w-1 &&
      t_pos.y > 0 && t_pos.y < volume_h-1 &&
      t_pos.z > 0 && t_pos.z < volume_n-1) {
    float ray_strength = 255;
    unsigned char voxel;
    float transparency;
    while(true) {
      if (t_pos.x < 0 || t_pos.x > volume_w-1 ||
          t_pos.y < 0 || t_pos.y > volume_h-1 ||
          t_pos.z < 0 || t_pos.z > volume_n-1) {
        break;
      }
      if (ray_strength < ray_strength_cutoff) break;

      voxel = volume_a((int)t_pos.x, (int)t_pos.y, (int)t_pos.z)
          ;
      if (voxel < transparent_level) voxel = 0;

      transparency = min((1 - voxel/255.0f) +
          transparency_ajustment, 1.0f);

      accum += ray_strength * (1-transparency);
      ray_strength *= transparency;

      t += step_size;
      t_pos = camera_pos + t*ray_dir;
    }
  } else {
    accum = ((ray_x+ray_y)%2)*150;
  }

  bitmap[n] = accum;
}
```

F.4 Incremental Reconstruction Kernels

The following code listing contains the kernels for incremental reconstruction with PT and DWOP as described in Section 5.3.

```
__kernel void trace_intersections(__global float4 *
    intersections,
                                  int volume_w,
                                  int volume_h,
                                  int volume_n,
                                        float volume_spacing,
                                        __global float4 *
                                           bscan_plane_equation_queue,
                                        int axis) {

    float4 Rd = {axis == 0, axis == 1, axis == 2, 0};

    int iter_end[3] = {(axis != 0)*volume_w+(axis==0), (axis != 1)
        *volume_h+(axis==1), (axis != 2)*volume_n+(axis==2)};

    int n = get_global_id(0);
    if (n >= iter_end[0]*iter_end[1]*iter_end[2]) return;

    int x = (axis != 0);
    int y = (axis != 1);
    int z = (axis != 2);

    if (axis == 0) {
      y = n%volume_h;
      z = n/volume_h;
    }
    if (axis == 1) {
      x = n%volume_w;
      z = n/volume_w;
    }
    if (axis == 2) {
      x = n%volume_w;
      y = n/volume_w;
    }

    bool invalid = false;
    for (int f = 0; f < 2; f++) {
      int i = f==0 ? BSCAN_WINDOW/2-1 : BSCAN_WINDOW/2; // Fill
           voxels between two middle bscans
      //int i = f==0 ? BSCAN_WINDOW/2-BSCAN_WINDOW/4-1 :
           BSCAN_WINDOW/2+BSCAN_WINDOW/4; // Alternatively fill
           voxels between BSCAN_WINDOW/2 middle bscans
      //int i = f==0 ? 0 : BSCAN_WINDOW-1; // Alternatively fill
           voxels between first and last bscan
      float4 Pn = {bscan_plane_equation_queue[i].x,
           bscan_plane_equation_queue[i].y,
           bscan_plane_equation_queue[i].z, 0};
      float4 R0 = {x*volume_spacing, y*volume_spacing, z*
           volume_spacing, 0};
      float Vd = dot(Pn, Rd);
      float V0 = -(dot(Pn, R0) + bscan_plane_equation_queue[i].w);
      float t = V0/Vd;
      if (Vd == 0) invalid = true;

    float4 intersection = R0 + t*Rd;
    intersections[n*2 + f] = intersection;
  }
}

__kernel void adv_fill_voxels(__global float4 * intersections,
                              __global unsigned char * volume,
                              float volume_spacing,
                              int volume_w,
                              int volume_h,
                              int volume_n,
                              __global float4 * x_vector_queue,
                              __global float4 * y_vector_queue,
                              __global plane_pts *
                                 plane_points_queue,
                              __global float4 *
                                 bscan_plane_equation_queue,
                              float bscan_spacing_x,
                              float bscan_spacing_y,
                              int bscan_w,
                              int bscan_h,
                              __global unsigned char * mask,
                              __global unsigned char *
                                 bscans_queue,
                              __global float *
                                 bscan_timetags_queue,
                              int intersection_counter) {

  int i = get_global_id(0);
  if (i >= intersection_counter) return;

  float4 intrs0 = intersections[i*2 + 0]/volume_spacing;
  float4 intrs1 = intersections[i*2 + 1]/volume_spacing;

  int x0 = min(intrs0.x,intrs1.x);
  int x1 = max(x0+1.0f, max(intrs0.x,intrs1.x));
  int y0 = min(intrs0.y,intrs1.y);
  int y1 = max(y0+1.0f, max(intrs0.y,intrs1.y));
  int z0 = min(intrs0.z,intrs1.z);
  int z1 = max(z0+1.0f, max(intrs0.z,intrs1.z));

  int safety = 0;
  for (int z = z0; z <= z1; z++) {
    for (int y = y0; y <= y1; y++) {
      for (int x = x0; x <= x1; x++) {
        float4 voxel_coord = {x*volume_spacing,y*volume_spacing,
            z*volume_spacing,0};
        if (inrange(x, 0, volume_w) && inrange(y, 0, volume_h)
            && inrange(z, 0, volume_n)) {
```

```
          float contribution = 0;
          if (PT_OR_DW) { // DW
            float dists[BSCAN_WINDOW];
            unsigned char bilinears[BSCAN_WINDOW];
            bool valid = true;
            float G = 0;
            for (int n = 0; n < BSCAN_WINDOW; n++) {
              int q_idx = n;

              float4 normal = {bscan_plane_equation_queue[q_idx
                  ].x, bscan_plane_equation_queue[q_idx].y,
                  bscan_plane_equation_queue[q_idx].z, 0};

              float dist0 = fabs(distance_pp(voxel_coord,
                  bscan_plane_equation_queue[q_idx]));
              float4 p0 = voxel_coord + -dist0*normal -
                  plane_points_queue[q_idx].corner0;
              float px0 = dot(p0, x_vector_queue[q_idx])/
                  bscan_spacing_x;
              float py0 = dot(p0, y_vector_queue[q_idx])/
                  bscan_spacing_y;
              float xa = px0-floor(px0);
              float ya = py0-floor(py0);
              int xa0 = (int)px0;
              int ya0 = (int)py0;

              bool valid0 = false;

              if (inrange(xa0, 0, bscan_w) && inrange(ya0, 0,
                  bscan_h) && inrange(xa0+1, 0, bscan_w) &&
                  inrange(ya0+1, 0, bscan_h)) {
                if (mask[xa0 + ya0*bscan_w] != 0 && mask[xa0+1 +
                     (ya0+1)*bscan_w] != 0 && mask[xa0+1 + ya0*
                    bscan_w] != 0 && mask[xa0 + (ya0+1)*bscan_w]
                    != 0) {
                  bilinears[n] = bscans_queue_a(q_idx,xa0,ya0)
                      *(1-xa)*(1-ya) + bscans_queue_a(q_idx,xa0
                      +1,ya0)*xa*(1-ya) + bscans_queue_a(q_idx,
                      xa0,ya0+1)*(1-xa)*ya + bscans_queue_a(q_idx
                      ,xa0+1,ya0+1)*xa*ya;
                  valid0 = true;
                }
              }

              valid &= valid0;
              dists[n] = dist0;

             G += 1/dists[n];
              contribution += bilinears[n]/dists[n];
          }

          if (!valid) continue;

          contribution /= G;

        } else { // PT
          // Find virtual plane time stamp:
          float dists[4];
          bool valid = true;
          for (int n = 0; n < 4; n++) {
            int q_idx = BSCAN_WINDOW/2-2+n;

            float4 normal = {bscan_plane_equation_queue[q_idx
                ].x, bscan_plane_equation_queue[q_idx].y,
                bscan_plane_equation_queue[q_idx].z, 0};

            float dist0 = fabs(distance_pp(voxel_coord,
                bscan_plane_equation_queue[q_idx]));
            float4 p0 = voxel_coord + -dist0*normal -
                plane_points_queue[q_idx].corner0;
            float px0 = dot(p0, x_vector_queue[q_idx])/
                bscan_spacing_x;
            float py0 = dot(p0, y_vector_queue[q_idx])/
                bscan_spacing_y;
            float xa = px0-floor(px0);
            float ya = py0-floor(py0);
            int xa0 = (int)px0;
            int ya0 = (int)py0;

            bool valid0 = false;
            float bilinear0;

            if (inrange(xa0, 0, bscan_w) && inrange(ya0, 0,
                bscan_h) && inrange(xa0+1, 0, bscan_w) &&
                inrange(ya0+1, 0, bscan_h))
              if (mask[xa0 + ya0*bscan_w] != 0 && mask[xa0+1 +
                  (ya0+1)*bscan_w] != 0 && mask[xa0+1 + ya0*
                  bscan_w] != 0 && mask[xa0 + (ya0+1)*bscan_w]
                  != 0)
                valid0 = true;

            dists[n] = dist0;

            valid &= valid0;
          }
          if (!valid) continue;
          float G = dists[1] + dists[2];
```

```
            float t = dists[2]/G*bscan_timetags_queue[
                BSCAN_WINDOW/2-1] + dists[1]/G*
                bscan_timetags_queue[BSCAN_WINDOW/2];

            // Cubic interpolate 4 bscan plane equations,
                corner0s and x- and y-vectors:
            float4 v_plane_eq = {0,0,0,0};
            float4 v_corner0 = {0,0,0,0};
            float4 v_x_vector = {0,0,0,0};
            float4 v_y_vector = {0,0,0,0};
            for (int k = 0; k < 4; k++) {
              int q_idx = BSCAN_WINDOW/2-2+k;
              float phi = 0;
              float a = -1/2.0f;
              float abs_t = fabs((t-bscan_timetags_queue[q_idx])
                  )/(bscan_timetags_queue[1]-bscan_timetags_queue
                  [0]);
              if (inrange(abs_t, 0, 1))
                phi = (a+2)*abs_t*abs_t*abs_t - (a+3)*abs_t*
                    abs_t + 1;
              else if (inrange(abs_t, 1, 2))
                phi = a*abs_t*abs_t*abs_t - 5*a*abs_t*abs_t + 8*
                    a*abs_t - 4*a;
              v_plane_eq += bscan_plane_equation_queue[q_idx]*
                  phi;
              v_corner0 += phi*plane_points_queue[q_idx].corner0
                  ;
              v_x_vector += phi*x_vector_queue[q_idx];
              v_y_vector += phi*y_vector_queue[q_idx];
            }

            // Find 2D coordinates on virtual plane:
            float4 p0 = voxel_coord - v_corner0;
            float px0 = dot(p0, v_x_vector)/bscan_spacing_x;
            float py0 = dot(p0, v_y_vector)/bscan_spacing_y;
            float xa = px0-floor(px0);
            float ya = py0-floor(py0);
            int xa0 = (int)px0;
            int ya0 = (int)py0;

            // Distance weight 4 bilinears:
            float F = 0;
            for (int n = 0; n < 4; n++) {
              int q_idx = BSCAN_WINDOW/2-2+n;
              float bilinear0 = bscans_queue_a(q_idx,xa0,ya0)
                  *(1-xa)*(1-ya) + bscans_queue_a(q_idx,xa0+1,ya0
                  )*xa*(1-ya) + bscans_queue_a(q_idx,xa0,ya0+1)
                  *(1-xa)*ya + bscans_queue_a(q_idx,xa0+1,ya0+1)*
                  xa*ya;
                F += 1/dists[n];
                contribution += bilinear0/dists[n];
              }
              contribution /= F;
            }

            if (COMPOUND_METHOD == COMPOUND_AVG)
              if (volume_a(x,y,z) != 0) volume_a(x,y,z) = (
                  volume_a(x,y,z) + contribution)/2;  else volume_a
                  (x,y,z) = contribution;
            if (COMPOUND_METHOD == COMPOUND_MAX)
              if (contribution > volume_a(x,y,z)) volume_a(x,y,z)
                  = contribution;
            if (COMPOUND_METHOD == COMPOUND_IFEMPTY)
              if (volume_a(x,y,z) == 0) volume_a(x,y,z) =
                  contribution;
            if (COMPOUND_METHOD == COMPOUND_OVERWRITE)
              volume_a(x,y,z) = contribution;
          }
        }
      }
    }
}
```