NTNU – Trondheim
Norwegian University of
Science and Technology

Improved Distance Weighted GPU-based 3D Ultrasound Reconstruction Methods

Tord Øygard

Master of Science in Computer Science
Submission date: February 2014
Supervisor: Frank Lindseth, IDI

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

Problem description

3D Ultrasound Reconstruction is used to generate 3-dimensional volumes from a set of 2-dimensional ultrasound slices and the positions of these slices. Used in an intra-operative setting, this can complement existing CT and MRI images acquired pre-operatively, since the ultrasound images can be obtained during surgery.

Earlier efforts have achieved fast GPU-based reconstruction using voxel-based distance-weighted methods. However, they currently have issues with complex probe movement patterns as well as poor reconstruction quality, especially with regards to intersecting planes. The goal of this thesis is to devise a GPU-based reconstruction algorithm which

1. handles complex probe movement patterns
2. generally gives higher-quality reconstructions
3. does not consume significantly more time reconstruction than existing GPU-based distance-weighted methods.

The algorithm will be described and evaluated both in terms of reconstruction quality and reconstruction speed. Technical issues concerning the implementation, such as GPU architecture and memory management will also be addressed.

Abstract

Ultrasound is a flexible medical imaging modality with many uses, one of them being intra-operative imaging for use in navigation. In order to obtain the highest possible spatial resolution and avoiding big, clunky 3D ultrasound probes, reconstruction of many 2D ultrasound images obtained by a conventional 2D ultrasound probe with a tracking system attached has been employed.

Earlier work in this field has yielded fast Graphical Processing Unit(GPU)-based implementations of voxel-based reconstruction algorithms such as Voxel Nearest Neighbor(VNN), Pixel Nearest Neighbor(PNN), VNN2 and Distance Weighted(DW) reconstruction. However, it is desirable to improve upon the reconstruction quality of the methods mentioned above. To do so, we propose in this thesis an adaptive algorithm called VGDW, which tries to intelligently smooth away speckles and noise, yet retains detail in high-frequency regions, while being not being much slower than the above mentioned algorithms. It also has a tunable weight function enabling value collisions to be handled gracefully.

Using our novel adaptive algorithm, we are able to produce very high-quality reconstructions, which are *unanimously* preferred over the output of the above mentioned algorithms by both a group of medical personnel and a group of technologists working with ultrasound, while having comparable computation time to VNN2 and DW, i.e. 16%, 10% and 5% difference from DW when computing a volume with 128 millions of voxels from a small, medium-sized and very large input data set using an AMD Radeon 6470M GPU. The algorithm also performs especially well with complex scanning patterns with overlapping data when using a customized weight function. As for future work, there are some aspects of the weight function that can benefit from improvement. Also, turning the problem upside down and looking at it from a pixel-based perspective could potentially give huge benefits both in terms of probe movement robustness and performance.

Acknowledgments

I have by no means been alone with this thesis, and a number of people and institutions have been very important in this work. The work has been done at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway, and I thank them for the resources provided. The work has been done in collaboration with the Department of Medical Technology at SINTEF in Trondheim, Norway, to whom I am also extremely grateful for their helpfulness and great flexibility.

My supervisor has been Dr. Frank Lindseth (SINTEF, IDI), who has provided me with invaluable feedback, insight and also help in practical matters during the course of the work. He has also been an inspiration for me, being a world leader in medical ultrasound and having a very close connection with the surgeons who will be using the results of this work. I would also like to thank Ole Vegard Solberg (SINTEF) for his sharp insight and vast knowledge of the field of 3D Ultrasound Reconstruction. He, together with Christian Askeland (SINTEF) and Janne Beate Lervig Bakeng (SINTEF) has helped me greatly in setting up and understanding the *CustusX* system, answered my questions diligently and most importantly, probably without their own knowledge, they have thought me many important lessons which were not only of use to this work in particular, but from which I will benefit for the years to come.

Finally, I would like to thank my wife, Marianne, for her patience, understanding, and many cups of tea and coffee served during my countless hours in front of the computer.

Trondheim, February 2014

Tord Kvestad Øygard

Contents

1 Introduction **1**

1.1 Goals 2
1.2 Contributions 2
1.3 Thesis outline 3

2 Background **5**

2.1 Ultrasound imaging 6
2.1.1 How does ultrasound work 6
2.1.2 Speckle 7
2.1.3 3D ultrasound 8
2.2 Tracking systems 11
2.3 GPU computing 11
2.3.1 OpenCL 12
2.3.2 Memory model 14
2.3.3 3D Ultrasound Reconstruction using OpenCL 16
2.4 Basic linear algebra and geometry 16
2.4.1 Coordinate frames 16
2.4.2 Homogeneous coordinates and transformation matrices 17
2.4.3 The plane equation 18
2.4.4 Finding the distance between a point and a plane . . . 19
2.4.5 Orthogonal projection onto a plane 19
2.5 Image statistics 21
2.6 Weight functions 22
2.7 Simple interpolation techniques 24
2.8 CustusX 26
2.8.1 Coordinate frames 27

3 Method **29**

3.1 3D Reconstruction algorithm 30
3.1.1 Overview 30
3.1.2 Finding the closest image planes: Restricted multi-start local search 32
3.1.3 Determining the voxel value: VGDW 42
3.1.4 OpenCL Implementation 44
3.2 Evaluation 45
3.2.1 The data sets 45

3.2.2 Reconstruction from existing volume 46
3.2.3 Synthetic volume 49
3.2.4 Slice removal 50
3.2.5 Visual evaluation on real data 51
3.2.6 Performance evaluation 52

4 Results 55
4.1 Reconstruction from existing volume 57
4.1.1 Reconstruction from existing ultrasound volume . . . 57
4.1.2 Reconstruction from existing MRI volume 62
4.2 Synthetic volume 66
4.2.1 Results 66
4.2.2 Images 67
4.3 Slice removal 72
4.3.1 Results 72
4.3.2 Images 73
4.4 Visual evaluation 75
4.4.1 Results 75
4.4.2 Images 78
4.5 Performance 81
4.5.1 32M Volume 81
4.5.2 128M Volume 82

5 Discussion 83
5.1 Reconstruction from existing volume 84
5.1.1 Reconstruction from existing ultrasound volume . . . 84
5.1.2 Reconstruction from existing MRI volume 84
5.2 Synthetic volume 84
5.3 Slice removal 86
5.4 Visual evaluation 86
5.5 Performance 88
5.6 General discussion 88
5.6.1 What is reconstruction quality? 89
5.6.2 Parameters 89
5.6.3 GPU considerations 90
5.6.4 32M vs 128M volumes 91

6 Conclusions/Further Work 93
6.1 Conclusions 94
6.2 Further work 94

6.2.1 Looking at the problem from the other side: a pixel-based approach . . . 95
6.2.2 The weight functions . . . 95
6.2.3 Performance on recent GPUs, APUs and even FPGAs 96
6.2.4 Applying Probe Trajectory estimation . . . 96
6.2.5 Investigating the effect of noise from the tracking system 96
6.3 Final thoughts . . . 97

Bibliography **101**

A Annotated Bibliography **103**

B Additional Results **105**
B.1 32M images vs 128M images . . . 105

C Code Listings **111**

List of Figures

2.1 An illustration of Ultrasonic imaging 6
2.2 A B-scan of the phantom "Kaisa" 7
2.3 Enlarged speckle region from Figure 2.2 8
2.4 Illustration of a Freehand 3D Ultrasound system 9
2.5 An illustration of work sizes 14
2.6 A conceptual illustration of a device in the OpenCL abstraction. 15
2.7 A right handed coordinate system 17
2.8 Illustration of the distance of a point P to a plane 20
2.9 Inverse distance functions 23
2.10 Linear distance functions 24
2.11 Gaussian distance functions 25
2.12 CustusX coordinate frames 27

3.1 Illustration of pixel inclusion strategies 31
3.2 Distance function for actual data acquired by a single sweep . 33
3.3 Distance function for actual data acquired by U-turn scanning 34
3.4 Figure to illustrate Algorithm 1 37
3.5 High and low variance between image planes 42
3.6 An illustration of the process used in the "simulated ultrasound from existing volume" evaluation 48
3.7 3D rendering of synthetic volume 49

4.1 The RMS Errors for the reconstruction from existing ultrasound volume evaluation 58
4.2 Images from the reconstruction from existing ultrasound volume evaluation in the X direction 59
4.3 Images from the reconstruction from existing ultrasound volume evaluation in the Y direction 60
4.4 Images from the reconstruction from existing ultrasound volume evaluation in the Z direction 61
4.5 The RMS Errors for the reconstruction from existing MRI volume evaluation 62
4.6 Images from the Reconstruction from existing MRI volume evaluation in the X direction 63
4.7 Images from the reconstruction from existing MRI volume evaluation in the Y direction 64

4.8 Images from the Reconstruction from existing MRI volume evaluation in the Z direction 65
4.9 The RMS Errors for the Synthetic volume test 66
4.10 Images from set 1 of the Synthetic Volume test in the X direction 67
4.11 Images from set 1 of the Synthetic Volume test in the Y direction 68
4.12 Images from set 1 of the Synthetic Volume test in the Z direction 68
4.13 Images from set 2 of the Synthetic Volume test in the X direction 69
4.14 Images from set 2 of the Synthetic Volume test in the Y direction 69
4.15 Images from set 2 of the Synthetic Volume test in the Z direction 70
4.16 Images from set 3 of the Synthetic Volume test in the X direction 70
4.17 Images from set 3 of the Synthetic Volume test in the Y direction 71
4.18 Images from set 3 of the Synthetic Volume test in the Z direction 71
4.19 The RMS Errors for the slice removal test 72
4.20 Images from the frame skip test with 0 frames skipped 73
4.21 Images from the frame skip test with 1 frames skipped 74
4.22 Images from the frame skip test with 3 frames skipped 74
4.23 Images from the frame skip test with 5 frames skipped 75
4.24 Results of the visual evaluation. 77
4.25 Slices reconstructed from the “Tumor” data set, X direction . 78
4.26 Slices reconstructed from the “Tumor” data set, Y direction . 79
4.27 Slices reconstructed from the “Tumor” data set, Z direction . 80
4.28 The results of the 32M performance evaluation 81
4.29 The results of the 128M performance evaluation 82

B.1 Images comparing 128M volumes to 32M volumes, X direction 106
B.2 Images comparing 128M volumes to 32M volumes, X direction 107
B.3 Images comparing 128M volumes to 32M volumes, Y direction 108
B.4 Images comparing 128M volumes to 32M volumes, Y direction 109
B.5 Images comparing 128M volumes to 32M volumes, Z direction 109
B.6 Images comparing 128M volumes to 32M volumes, Z direction 110

List of Tables

3.1 Parameters for reconstruction from existing ultrasound volume evaluation. 47
3.2 Parameters for reconstruction from existing MRI volume evaluation. 47
3.3 Parameters for Synthetic volume evaluation. 50
3.4 Parameters for slice skipping evaluation. 51
3.5 Parameters for visual evaluation. 52
3.6 Specifications of computer used for performance evaluation . 53

4.1 Results from the reconstruction from existing US volume evaluation 57
4.2 Results from the reconstruction from existing MRI volume evaluation 62
4.3 Results from the Synthetic volume test described in Section 3.2.3 66
4.4 Results from the Slice skip evaluation described in Section 3.2.3 72
4.5 Results from visual evaluation 76
4.6 The results of the 32M performance evaluation 81
4.7 The results of the 128M performance evaluation 82

List of Algorithms

1	Finding the initial guesses for the multi-start local search . .	35
2	Find close planes that are close to the guess, single start . . .	38
2	Find close planes that are close to the guess (continued) . . .	39
3	Find close planes that are close to guesses, multi-start	41

List of Abbreviations

VGDW Varying Gaussian Distance Weighted, a 3D Ultrasound Reconstruction algorithm presented in this thesis

AGDW Adaptive Gaussian Distance Weighted, a 3D Ultrasound Reconstruction algorithm

B-scan A 2D ultrasound image

CPU Central Processing Unit

CT Computed Tomography

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

DSP Digital Signal Processor

DW Distance Weighted, a 3D Ultrasound Reconstruction algorithm

FPGA Field-Programmable Gate Array

GPGPU General Purpose Graphics Processing Unit

GPU Graphics Processing Unit

MRI Magnetic Resonance Imaging

PNN Pixel-nearest-neighbor, a simple 3D Ultrasound Reconstruction algorithm

PT Probe Trajectory, a 3D Ultrasound Reconstruction algorithm

US Ultrasound

VNN Voxel-nearest-neighbor, a simple 3D Ultrasound Reconstruction algorithm

Chapter 1

Introduction

The 3D Ultrasound Reconstruction for intra-operative applications problem can be formulated as:

> Given a set of images $\mathbf{I}$ that have been sampled from a real-world volume V, as well as the positions of the images $\mathbf{P}$ and other obtainable data (for instance timestamps for each image plane), one wants to compute a 3D image of the volume V, V', that is of as much use to a physician as possible, without spending too much time doing it.

This definition is intentionally vague and has an important human component: the physician. There are many ways to argue that some reconstruction may be better than another, but in the end, if the physician is able to make a correct decision based on one reconstruction, and a faulty decision based on another, the first reconstruction is clearly best. Therefore, things like noise suppression, contrast, and geometrical accuracy is important – we want to present the physician with the information he needs in an as accessible as possible fashion without distracting noise and artifacts. Also, we want the information to be available as fast as possible, meaning that the time consumption should be kept as low as possible. This rules out options that may produce very high-quality results, but spend too much time computing.

This is the problem for which we propose an approach in this thesis. We will leverage the computing power of modern GPU-s, as well as employ advanced image filtering techniques to perform the reconstruction, while making trade-

offs to ensure the computation time stays low. We will draw inspiration from previous work in the field, most importantly from Huang et. al-s AGDW algorithm[Huang et al., 2009].

We will see an adaptive reconstruction algorithm that adjusts its "smoothness" depending on whether there is much detail in a region.

1.1 Goals

The goals of this thesis are to

- Examine the problem of reconstruction quality in distance-weighted methods in 3D Ultrasound Reconstruction, especially in the context of complex scanning patterns.
- Describe a new 3D Ultrasound Reconstruction algorithm addressing reconstruction quality while keeping computation time in check.
- Implement the algorithm on a GPU.
- Evaluate implementation, primarily in terms of reconstruction quality but also in terms of computational speed.

1.2 Contributions

The main contributions of this thesis are

- A novel, dynamic 3D Ultrasound Reconstruction weighting scheme called Varying Gaussian Distance Weighing(VGDW) which tries to find a trade-off between sharp detail and noise suppression, as well as using weight functions to emphasize certain elements in the input data.
- A GPU implementation of this algorithm as well as other reconstruction algorithms, fully integrated into the existing system *CustusX* developed by SINTEF Medical Technology.
- Two novel evaluation techniques based on synthetic and simulated ultrasound acquisition.

1.3 Thesis outline

The rest of this thesis is outlined as follows

Chapter 2: Background contains useful background theory such as linear algebra, local image statistics, weight functions, interpolation techniques, and also investigates Ultrasonic Imaging and earlier efforts in 3D Ultrasound Imaging, from which we draw inspiration. It gives a brief introduction to GPU Computing and OpenCL.

Chapter 3: Method delves into the details of VGDW and its subproblems, including the problem of finding the closest image planes, and the interpolation once the image planes are known. It also describes the methods employed to evaluate the algorithm against other known algorithms.

Chapter 4: Results presents the results of our five image quality evaluation methods, where one is synthetic and four deal with real-world data. It also presents the results of performance evaluation.

Chapter 5: Discussion interprets and discusses the findings from Chapter 4, as well as some general thoughts on the subject of VGDW and 3D Ultrasound Reconstruction in general.

Chapter 6: Conclusion draws conclusions from the discussion in Chapter 5, and presents some avenues for future work in this direction.

Chapter 2

Background

In this chapter we will review some background theory related to the work of this thesis. First, we will give a brief overview of how ultrasound imaging works, and some earlier work in the realm of 3D Ultrasound Reconstruction, and we will say a few words about an important piece in the puzzle: the tracking system. Further, we will see some basics concerning GPU computing, and we will review some of the relevant mathematical tools. Finally, we will briefly discuss the system *CustusX*, the framework within which the work of this thesis was done.

2.1 Ultrasound imaging

The ultrasonic imaging modality is a powerful imaging technique that enables a physician to visualize tissues inside a patient in real time. A typical ultrasonic imaging system is composed of a probe and a screen, as well as a control panel to adjust the image acquisition parameters. There are two basic things that one can visualize using ultrasonic imaging: tissue (B-mode)[Havlice and Taenzer, 1979] and flow (Doppler)[Strandness Jr. et al., 1967]. I will now very briefly present the very basic principle of ultrasonic imaging.

2.1.1 How does ultrasound work

Ultrasound consists of sound waves at a high frequency (typically 2-18 MHz for imaging). As we know, sound waves are longitudinal waves, i.e. they traverse the medium in the same direction as the motion of the waves. The speed of travel of sound waves (and thus also ultrasonic waves) are dependent on the medium in which it travels. When the sound wave is traveling in a medium 1 and meets a medium 2, which has a different speed of sound from medium 1, parts of the sound wave will be reflected off the interface between the two media. This "echo" is what is exploited in ultrasonic imaging. In 1D US imaging (A-mode), a transducer transmit an ultrasonic pulse, and then switches over to listening for echoes. When an echo comes back, A-mode ultrasound displays an intensity relative to the intensity received back, using the time difference from the pulse was transmitted until the echo was received to estimate the depth of the interface. This is illustrated in Figure 2.1.

This can be extended to 2D by using several transducer elements arranged in some form of array (linear array, phased array). 2D ultrasound tissue imaging is usually called B-mode imaging, where the "B" stands for "Brightness". In other words, the intensity of the echoes are translated into the brightness on the image. An example of a B-scan can be seen in Figure 2.2.

2.1.2 Speckle

B-mode images usually have a grainy appearance. This is caused by a phenomenon called "speckle", which is a form of multiplicative, locally-correlated noise, as seen in Figure 2.3. There have been many approaches to

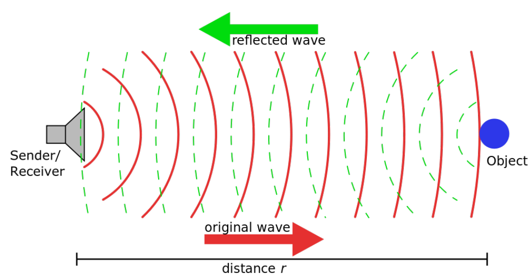


Figure 2.1: An illustration of Ultrasonic imaging. Illustration originally by G. Wiora[Wiora, 2005].

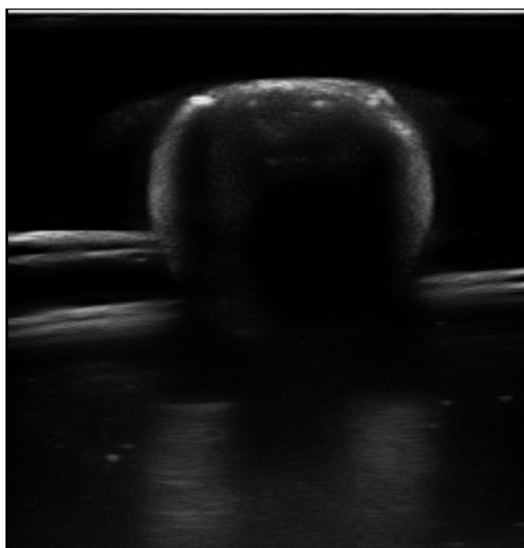


Figure 2.2: A B-scan of the phantom "Kaisa"

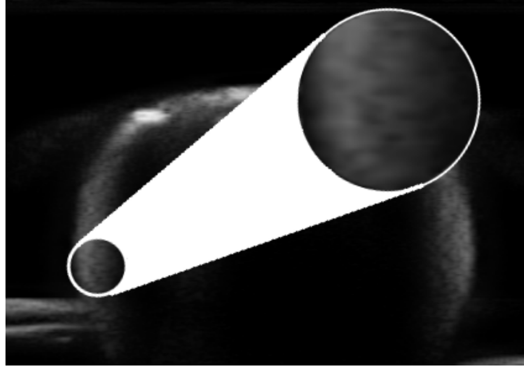


Figure 2.3: Enlarged speckle region from Figure 2.2

reduce speckle in ultrasound images, including adaptive median filters[Loupas et al., 1989], and variations on anisotropic diffusion [Yu and Acton, 2002], which show good results in reducing speckle in ultrasound images.

2.1.3 3D ultrasound

In the recent years, 3D ultrasound has become a viable imaging modality. At the moment, there are two principal methods for obtaining 3D ultrasound Images – using 3D ultrasound probes, or using a 2D probe in conjunction with a tracking system and reconstruct a 3D volume, often called Freehand 3D ultrasound. Different kinds of dedicated 3D ultrasound probes exist and have been discussed by Fenster et.al.[Fenster et al., 2001]. One variant is a mechanical probe which consists of an 1D or a 2D probe and a motor to sweep it over the volume. This has the disadvantage of being quite bulky and/or inflexible, and thus unusable in many clinical settings. Another variant, often called 4D ultrasound, has a 2D array of transducer elements, can obtain real-time images at the cost of some spatial resolution, which is undesirable in many clinical settings. These probes are also quite expensive.

That leaves us with Freehand 3D Ultrasound, where one uses a 2D ultrasound probe and a tracking system to obtain the image data(see Figure 2.4, and then reconstruct it into a regular 3D voxel grid. This has the advantage of being relatively inexpensive and flexible, and allows leveraging existing high-resolution 2D ultrasound probes. We will now investigate some principles and previous approaches to Freehand 3D Ultrasound Reconstruction. O.

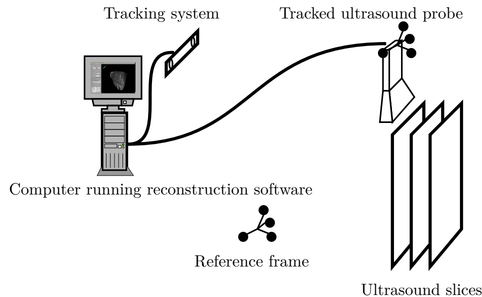


Figure 2.4: Illustration of a Freehand 3D Ultrasound system

Solberg did a summary of the research in this field[Solberg et al., 2007], as did D. Miller et. al[Miller et al., 2012] and we will reiterate some of the algorithms here, as well as some more.

Pixel Nearest Neighbor (PNN)

Pixel Nearest Neighbor is a simple reconstruction algorithm, and works by iterating over each image plane, and transforming it into the voxel space. In essence, it asks the question "I have this data, where should it go?". In concrete words, for each pixel on the image plane, the nearest voxel in the voxel grid is found, and the pixel value is put into that voxel. If the voxel already has a value, different approaches are possible: Taking the average, taking the maximum, taking the most recent value, or taking the first value. Usually this is followed by a Hole Filling Step, where the voxels that have no value get a value from the neighboring voxels.

Voxel Nearest Neighbor (VNN)

VNN is another simple algorithm, which asks the opposite question of PNN: "Where should this voxel get its data from?". Concretely, it iterates over the output volume voxels and for each voxel, it finds the image plane that is closest to that voxel, and then the pixel on that plane that is closest to the voxel. Usually, there is a maximum distance D involved, where if there is no image plane closer to the voxel than D, the voxel gets no value.

VNN2

VNN2 is a slightly more complex variant of VNN, where instead of simply taking the closest pixel, all the image planes that are closer than D is obtained, and a distance-weighted average of the closest pixel from each of these planes is computed.

Distance Weighted Reconstruction (DW)

There appears to be some confusion about what DW really is: The original article by Barry et. al.[Barry et al., 1997] suggests that every pixel inside the sphere with radius D is included and a distance-weighed sum is computed. Other authors, such as Coupé et. al[Coupé et al., 2005] and D. Miller et. al. [Miller et al., 2012] take it to mean almost the same as VNN2, with the exception that instead of taking the closest pixel on each image plane, one performs bi-linear interpolation on each image plane. In this thesis, we will generally refer to the variant described by Coupé et. al. and D. Miller et. al, unless otherwise is stated.

Probe Trajectory (PT)

Coupé et.al.[Coupé et al., 2005] proposes an alternative to the orthogonal projection used by VNN2 and DW, as it uses cubic interpolation on the timing data to estimate the trajectory of the probe, and thus getting a more “correct” position on the image plane. This method has shown very good results in terms of reconstruction quality, and a fast GPU implementation already exists [Ludvigsen, 2010].

Weighted Median Reconstruction

Weighted Median Reconstruction is a more sophisticated reconstruction technique, where a distance-weighted median value of the nearby pixels are computed. This was proposed by Huang et.al[Huang and Zheng, 2008], and is good at suppressing speckles, but is a rather slow reconstruction technique.

Adaptive Distance-Weighted Gaussian Reconstruction

Huang et.al has also proposed an Adaptive Distance-Weighed Gaussian Reconstruction [Huang et al., 2009], which is based on recursively computing some local statistics (mean and variance) on progressively smaller neighborhoods. If the variance to mean ratio becomes very small, the voxel is considered to be located in a homogeneous region, and a trimmed mean filter is applied. If the variance to mean ratio never becomes small, the voxel is considered to be located in an inhomogeneous region, and a Gaussian convolution kernel is applied. The paper does not state the computational time required, but our instinct is that it is quite computationally demanding.

All these earlier approaches have a critical part in common; the tracking system, which we will now briefly investigate.

2.2 Tracking systems

A tracking, or a positioning system is some system that keeps track of the position and orientation of some objects. In the context of Freehand 3D Ultrasound Imaging, the position of the probe is what's of interest. Of course, if navigation is also required, one needs to also keep track of the position of the patient, as well as some tool.

There are many types of tracking systems (see [Cinquin et al., 1995] for an overview), but in this thesis we will treat the tracking system as some opaque system that provides us with the transformation from some known reference to the position of some tool. More precisely, the affine transformation matrix that takes us from from World space to Image space, ${}^{W}\mathbf{M}_{I}$ such that ${}^{W}\mathbf{P} = {}^{W}\mathbf{M}_{I}{}^{I}\mathbf{P}$, where ${}^{W}\mathbf{P}$ is some point $\mathbf{P}$ in World space, and ${}^{I}\mathbf{P}$ is the same point in Image space.

Naturally, tracking systems are not perfect, and may introduce noise. The tracking system used in this thesis, NDI Polaris Spectra, has a RMS error of 0.25 millimeters[NDI, 2014]. This noise has not been further modelled in this thesis, we assume that the tracking system produces adequate positioning for our purposes.

We have now seen how to obtain the necessary data to perform 3D Ultrasound Reconstruction. We now investigate a platform upon which we can compute the reconstruction – the GPU.

2.3 GPU computing

Most modern personal computers are equipped with at least one GPU (Graphics Processing Unit). A GPU is a specialized hardware unit that historically has been dedicated to rendering 2D and 3D graphics. Over the last years, GPUs have become increasingly powerful. While that is to expect, they have had a much steeper performance improvement than CPU-s[nVidia, 2009].

In order to leverage this computational power for other purposes than computer graphics, the concept of General Purpose GPU-computing (GPGPU) has arrived, along with a set of standards and programming API-s.

The most notable standards are OpenCL[Khronos OpenCL Working Group, 2012], CUDA[nVidia, 2013] and the recent OpenAAC[OpenAAC, 2013] standard. Out of these, OpenCL is the most widely adopted – CUDA is developed by nVidia for nVidia hardware only, and OpenAAC is a fairly recent standard yet to be adopted by most. OpenCL on the other hand is supported by GPUs from Intel, nVidia, AMD, ARM, Imagination and Qualcomm to mention some, and is supported on Microsoft Windows, Apple Mac OS X and Linux, of course provided appropriate drivers are installed. This thesis will therefore restrict itself to utilizing the OpenCL standard for GPU computing.

2.3.1 OpenCL

OpenCL is a standard for parallel computing backed by The Khronos Group. It describes an execution model where there is two sides to an application: the *host* side, and the *device* side. A typical application has some native code running on the CPU, which is considered to be on the *host* side. This code may call functions inside the *host-side runtime libary*, which includes a compiler for the OpenCL C Language, as well as functions to set up data transfers to and from the GPU memory, and, of course, execute the compiled OpenCL C code on the GPU. The function that is to be executed on the GPU is called a *kernel*. A *kernel* is a function written in the OpenCL C code that is exposed to the host-side via the host-side runtime library. It should be noted that the OpenCL standard is not restricted to GPUs – multiple types of accelerators are supported. DSPs, CPUs and even FPGAs may be targeted by OpenCL. For our application, GPUs is what we will focus on.

Host-side runtime library

The host-side runtime library exposes functions to perform the following tasks:

- Enumerate available OpenCL devices, and query them for properties
- Compile OpenCL C code into binary code executable by the chosen OpenCL device, i.e. compile the kernel
- Transfer data from and to the GPU
- Enqueue calls to kernels
- Synchronization
- Coordination with OpenGL

The host-side runtime library maintains a Command Queue, to which one can enqueue GPU operations. One can enqueue data transfers and kernel executions. The queue can be un-ordered, if supported by the implementation, in which case one has to specify dependencies between the operations – but that is not of relevance in this thesis.

For a typical application, the use of the host-side runtime library is restricted to finding a suitable OpenCL device, compiling the OpenCL C kernel for that device, setting up the data transfers for the inputs to the kernel, enqueueing the kernel and a data transfer to read the data back, and then waiting for the data transfer to be finished.

Device-side programming language

The OpenCL C Language is a language very similar to C99, with a few exceptions:

- The `__kernel` keyword, marking a function as an entry point (i.e. it can be called by the host-side application)
- The memory space qualifiers, `__global`, `__constant`, `__local` and `__private` designating which memory space a variable or array should be stored in (see Section 2.3.2)
- Vector data types, such as `float4`, `float16`, allowing vector operations to be written explicitly in the code.

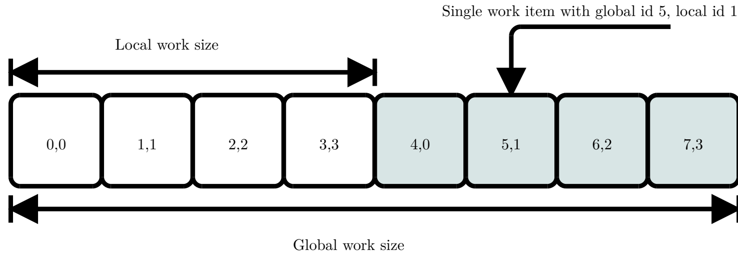


Figure 2.5: An illustration of work sizes: Global work size is 8 and local work size is 4.

- A set of library functions for vector operations on the vector data types, various mathematical functions, and data transfers between the memory spaces, synchronization and very importantly getting the work-item ID (see Section 2.3.1).

Execution model

The purpose of OpenCL is to allow exploitation of parallelism. When the execution of a kernel is enqueued, two important parameters dictate the parallelism of the execution: the *global work size* and the *local work size*. The global work size is the size of the work altogether – in other words, how many threads will be executed. The local work size describes how many of these threads will be grouped together into a *work group*. The threads in a work group can be synchronized with each other, access the same local memory and enforce consistency on that local memory. Naturally, the global work size has to be a multiple of the local work size. Each thread is assigned two ID-s: its global ID and its local ID. The global ID refers to the thread ID respective to the global work size, and the local ID refers to the thread ID respective to the work group. This is illustrated in Figure 2.5.

2.3.2 Memory model

OpenCL distinguishes between 4 different memories: the *global* memory, the *constant* memory, the *local* memory and the *private* memory. A concep-

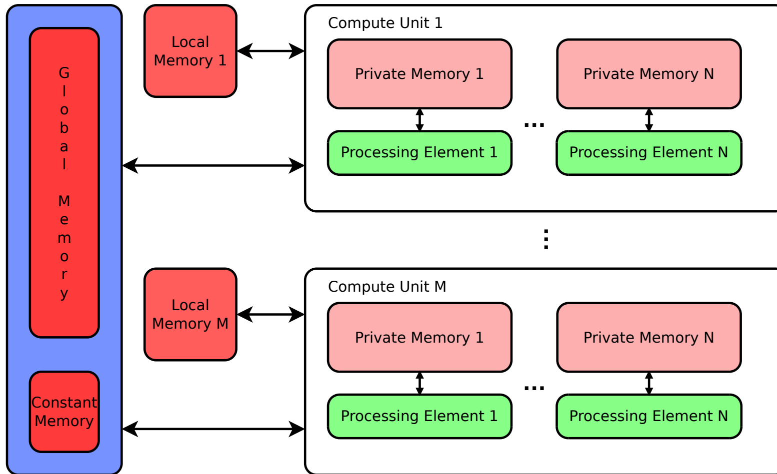


Figure 2.6: A conceptual illustration of a device in the OpenCL abstraction.

tual drawing is shown in Figure 2.6, and we will now describe the different memories.

Global memory The *global* memory is readable and writable by all threads, and it's large. As usual with memory hierarchies, the largest memory is also the slowest one in most implementations[1]. The global memory is also where the input data to the kernel is stored, and where the output data is read from (the host-side library cannot access any other memory space). An important thing to notice is that it is not possible to synchronize accesses to global memory, at least not from inside the kernel. As of OpenCL 1.2, images are also available. This allows a programmer to use features such as bi-linear interpolation in hardware, as well as exploiting any texture caches available on the GPU.

Constant memory The constant memory is a small, fast read-only memory suitable for storing parameters and small structures that remain constant during execution.

[1]Some GPU's do not have different memories in hardware, so local memory is implemented using the same memory as global memory

Local memory The *local* memory is readable and writable by *all threads in the same work group.* In other words, there is a fixed amount of local memory available for each work group. This memory is very fast, and should be used whenever possible. It is also quite small, which results in the need to prioritize what to use it for. Within a local work group, it is possible to enforce memory consistency with the local memory, as barriers and memory fences are available for local work groups.

Private memory Finally, the private memory is a very small memory that is local to each thread. This corresponds to the register space in each GPU processing core. This memory is typically very small, and its size is not transparent to the programmer. Most commonly this causes *register spilling*, which means that the data has to be written out to memory and accessed from there. Typically registers spill to global memory, so there is a severe performance penalty associated with having large data structures in private memory as opposed to having them in local memory. On many implementations it is also limited in terms of addressing flexibility. Using arrays in the private memory space may therefore lead to the array being spilled to global memory to accommodate indirect addressing.

2.3.3 3D Ultrasound Reconstruction using OpenCL

H. Ludvigsen et.al.[Ludvigsen, 2010] has shown successful implementation of PNN, VNN, DW and PT, with quite impressive results in terms of speedup compared to standard CPU implementations. T.K. Valderhaug et.al [Valderhaug, 2010] improved upon Ludvigsens work and achieved even better performance by leveraging OpenCL 1.2-s image samplers and some general optimizations.

2.4 Basic linear algebra and geometry

Since we will be dealing with points, lines and planes in a 3D setting, we have to establish some basic knowledge of geometry and linear algebra[Hearn, 2011]. Since the required knowledge is very similar to that in my specialization project[Øygard, 2013], much of this section is repeated from that.

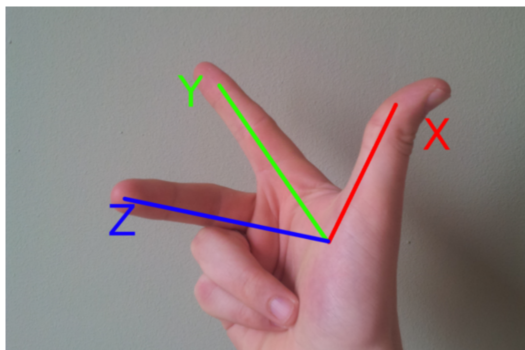


Figure 2.7: A right handed coordinate system

2.4.1 Coordinate frames

A coordinate frame is a frame of reference in which one can specify points. As we will see Section 2.8.1, we may have more than one frame of reference and thus we need to provide ways to move between them. A (Cartesian) coordinate frame is uniquely determined relative to a world reference system by two things: the position of the origin and the orientation of the axes. For 3D Cartesian coordinate frames, there exists many different representations, but the most commonly used is to describe them using three unit vectors, all of them perpendicular to each other, one per axis, pointing in the direction of that axis. This still leaves ambiguity - if the X axis points to the right, and the Y axis points up, where does the Z axis point? In or out? We adopt the convention of *right handed coordinates*, so named because if you use your right hand thumb and index finger to point along the X and Y axes respectively, the the Z axis points along the middle finger (see Figure 2.7).

2.4.2 Homogeneous coordinates and transformation matrices

Homogeneous coordinates in 3D are coordinates that have an extra component used for scaling. However, for all the transformations we will consider here, the extra component will remain the constant 1. So, to represent the point $\mathbf{P} = (P_x, P_y, P_z)$ in homogeneous coordinates, we write

$$\mathbf{P} = [P_x, P_y, P_z, 1]^T$$

The main advantage of using homogeneous coordinates is the fact that one can then represent affine transformations (such as translation and rotation) as a single 4x4 matrix. As mentioned earlier, a coordinate frame is uniquely determined relative to a world reference system by exactly the position of the origin (translation), and the orientation of the axes(rotation). Thus, we can use a homogeneous transformation matrix to transform points from one coordinate system to another. These transformations are written on the form

$$^{j}\mathbf{P} = {}^{j}\mathbf{M}_{i}{}^{i}\mathbf{P}$$

where $^{j}\mathbf{P}$ is the point $\mathbf{P}$ in the coordinate system j, and $^{i}\mathbf{P}$ is the same point $\mathbf{P}$ in the coordinate system i, $^{j}\mathbf{M}_{i}$ is a the transformation matrix transforming from coordinate system i to j, on the form

$$^{j}\mathbf{M}_{i} = \begin{bmatrix} r_{x1} & r_{x2} & r_{x3} & t_{x} \\ r_{y1} & r_{y2} & r_{y3} & t_{y} \\ r_{z1} & r_{z2} & r_{z3} & t_{z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

where the r_{kj}-s are the unit vectors pointing in the direction of the x, y and z axes in coordinate system j as seen from coordinate system i , and the t_k-s are the translation offsets from the origin of coordinate system i to coordinates system j as seen from coordinate system i.

2.4.3 The plane equation

We will also be dealing with planes in 3 dimensions, since the tracking system gives us the position of the image planes. To represent the image planes we will use the plane equation, which is given as

$$ax + by + cz + d = 0 \tag{2.2}$$

such that for every point $\mathbf{P} = [x\ y\ z]^T$ that is on the plane, the equation holds. Thus, a plane is determined by the four coefficients a, b, c and d. Now, given that one has a transformation matrix $^{i}\mathbf{M}_{j}$ that transforms from some coordinate system j to i, and we want to find the coefficients a, b, c and d such that the plane equation represents the XY plane of the coordinate system j in coordinate system i, we interpret the plane equation as follows: a, b and c represents an unit vector that is normal to the plane, and d represents the distance along that vector which the plane is translated from the origin.

Then, we can determine the XY plane by using the unit vector corresponding to the Z axis as a, b and c:

$$a = r_{x3} \tag{2.3}$$

$$b = r_{y3} \tag{2.4}$$

$$c = r_{z3} \tag{2.5}$$

Lastly, we need to determine d, which is given as:

$$d = -aT_x - bT_y - cT_z \tag{2.6}$$

This method produces a normalized plane method (i.e. $\sqrt{a^2 + b^2 + c^2} = 1$, which is a requirement for the coefficients we selected in the rotation matrix), and is the only method used in this project. We will therefore from here on assume that plane equations are normalized.

2.4.4 Finding the distance between a point and a plane

If one has computed a plane equation as described in the previous section, finding the distance between a point and a plane is trivial. Recalling that the plane equation can be viewed as a vector $\vec{n} = [a, b, c]^T$ and a distance traveled along that vector d, if we take some point $\mathbf{x} = [x, y, z]^T$ and insert in into the plane equation, what we are computing is "How much too far has this point traveled along the normal vector to be on the plane?". This is illustrated in Figure 2.8, and can be expressed as

$$dist(\mathbf{x}, \mathbf{P}) = \vec{n} \bullet \mathbf{x} + d \tag{2.7}$$

where $\mathbf{x}$ the point, $\mathbf{P}$ is the plane with plane equation coefficients a, b, c, d, and $\vec{n} = [a, b, c]^T$.

Further, if one represents the plane equation as a vector $\mathbf{P} = [a, b, c, d]^T$, and the point $\mathbf{x}$ with homogeneous coordinates $\mathbf{x} = [x, y, z, 1]^T$ we observe that Equation 2.7 becomes

$$dist(\mathbf{x}, \mathbf{P}) = \mathbf{x} \bullet \mathbf{P} = x \cdot a + y \cdot b + z \cdot c + 1 \cdot d \tag{2.8}$$

Finally, one important thing to notice about the function $dist(\mathbf{x}, \mathbf{P})$ is that it is signed – if the point $\mathbf{x}$ lies on the side of the plane that the vector $\vec{n}$ points away from, meaning that the point would have to travel further along the vector, the distance will be negative. Conversely, if the point has "traveled too far" along the normal vector in to be on the plane, the distance will be positive.

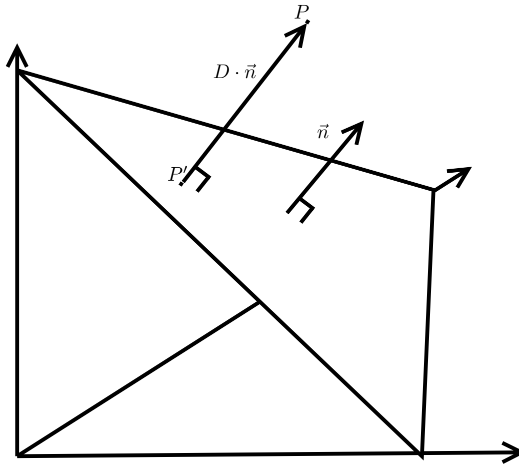


Figure 2.8: Illustration of the distance of a point P to a plane. The point P' is the point P projected orthogonally onto the plane. As we see, the point P is translated by D along the normal unit vector $\vec{n}$.

2.4.5 Orthogonal projection onto a plane

When dealing with 3D, it happens that one has to make 3D points correspond to some points on a 2D surface. For instance, in computer graphics one needs to display the 3D scenes to a 2D screen. This process is called "projection". There are many conceivable ways to do this, but the two most common are the *perspective* projection, and the *orthogonal* projection. The perspective projection is useful for realistic visualization, and we will not go into it here. The orthogonal projection is so named because each projected point is projected along a line that is orthogonal to the plane it is projected to. Thus, the orthogonal projection of a point $\mathbf{x}$ onto a plane $\mathbf{P}$ can be computed as

$$\mathbf{x}_P = projectOrthogonallyOnto(\mathbf{x}, \mathbf{P}) = \mathbf{x} - dist(\mathbf{x}, \mathbf{P}) \cdot \vec{n} \tag{2.9}$$

where $\mathbf{x}_P$ is the projected point, $dist(\mathbf{x}, \mathbf{P})$ is the distance between the point and the plane (as computed using Equation 2.8), and $\vec{n}$ is the unit vector normal to the plane.

Finally, we observe that for a plane represented by Equation 2.2, assum-

ing that the coefficients are normalized, the vector $\vec{n} = [a, b, c]^T$ is a unit vector orthogonal to the plane, meaning that we do not need any further information than the plane equation to project points onto a plane.

2.5 Image statistics

Some local statistics are useful when performing operations on images. But as important as the statistics one computes, is what input data one uses to compute the statistics. For example, one may compute the variance over the pixels in a 3x3 region on a 2D image. The resulting value tells us something about how different the pixels in that region is – if the variance is big, it may indicate a noisy region, or that there is an edge in the picture.

One may also compute the variance over all pixels from image planes that are closer to a specific point than some radius. If that variance is high, it may indicate that some of the image planes have shadows in them, or that the data is noisy, or even that this is a region in which there is detail, giving a hint that this region should not be smoothed too much. With that said, we will formally describe some concepts. Note that we will restrict ourselves to the discrete variants - the continuous statistics are not of interest in this thesis.

Arithmetic mean

The arithmetic mean μ of a set of N values $\mathbf{x} = (x_0, x_1, \cdots x_{N-1})$ is computed as

$$\mu = E(\mathbf{x}) = \frac{1}{N} \sum_{i=0}^{N-1} x_i \tag{2.10}$$

Weighted average

The weighted average μ^* of a set of N values $\mathbf{x} = (x_0, x_1, \cdots x_{N-1})$ with N weights $\mathbf{w} = (w_0, w_1, \cdots w_{N-1})$ is computed as

$$\mu^* = \frac{1}{\sum_{i=0}^{N-1} w_i} \sum_{i=0}^{N-1} w_i x_i \tag{2.11}$$

The weighted average is the premise for algorithms such as VNN2 and DW, a weighted average of pixels from nearby planes are computed, with a weight function increasing in distance. We will discuss weight functions closer in Section 2.6.

Trimmed mean

The trimmed mean μ' of a set of N values $\mathbf{x} = (x_0, x_1, \cdots x_{N-1})$ with some cutoff value K is computed by making a new set of N' values

$$\mathbf{x}' = \{x \mid x \in \mathbf{x} \wedge |x - E(\mathbf{x})| < K\}$$

and then computing $\mu' = E(\mathbf{x}')$. The trimmed mean is useful for computing a mean that isn't too sensitive to outliers, as these will be ignored. A good value for K is $K = \sqrt{\sigma^2} = \sigma$.

Variance

The variance σ^2 of a set of N values $\mathbf{x} = (x_0, x_1, \cdots x_{N-1})$ is computed as

$$\sigma^2 = Var(\mathbf{x}) = \frac{1}{N-1}\sum_{i=0}^{N-1}(\mu - x_i)^2 \tag{2.12}$$

2.6 Weight functions

When computing a weighted average, the result is very much dependent on what weights are being used. Weight functions allow a way to give higher priority to elements that has some desired property. In the context of 3D Ultrasound Reconstruction, the most commonly used property is being close in distance, but even those functions may be different. We will now present some distance functions suitable for use with algorithms similar to DW.

Inverse distance functions

For purely distance-weighted methods, functions on the form

$$w_x = 1/dist(x)$$

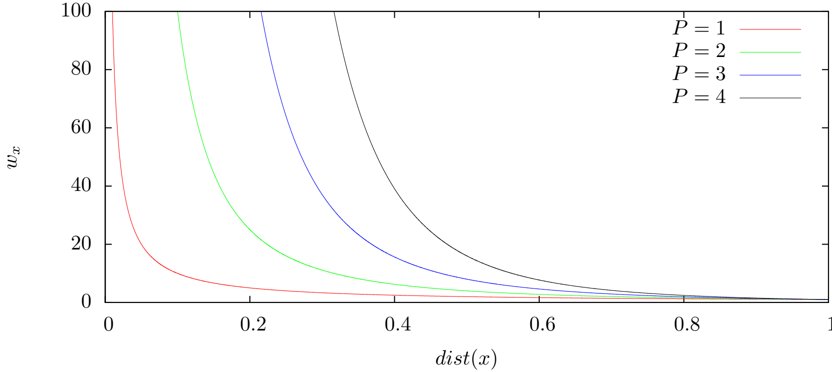


Figure 2.9: Inverse distance functions

where w_x is the weight of pixel value x, and $dist(x)$ is the distance from the pixel x on its image plane to the target voxel, are proposed by [Barry et al., 1997]. One may even consider introducing a parameter P, such that

$$^{P}w_x = 1/dist(x)^P \tag{2.13}$$

As we may see in Figure 2.9, the weight function becomes steeper with increasing P, meaning that with for any two distances d_1 and d_2, with $d_1 < d_2$, if we use the different P-s p_1 and p_2 with $p_1 < p_2$, we have $\frac{^{p_1}w_1}{^{p_1}w_2} < \frac{^{p_2}w_1}{^{p_2}w_2}$, meaning that ^{p_1}w gives a relatively higher weight to the closer value. One practical consideration when dealing with inverse distance functions is that the weight values become very high when approaching $dist(x) = 0$. This may cause inaccuracies in floating point calculations in a computer, and also makes it harder to extend the weight function to account for other properties than distance.

Linear distance functions

Another weight function one can consider is a simple linear distance function, on the form

$$^{R,P}w_x = R^P - dist(x)^P \tag{2.14}$$

where R is the radius – i.e. the maximum distance for which any value will be considered. As we may see in Figure 2.10, the linear functions are not nearly

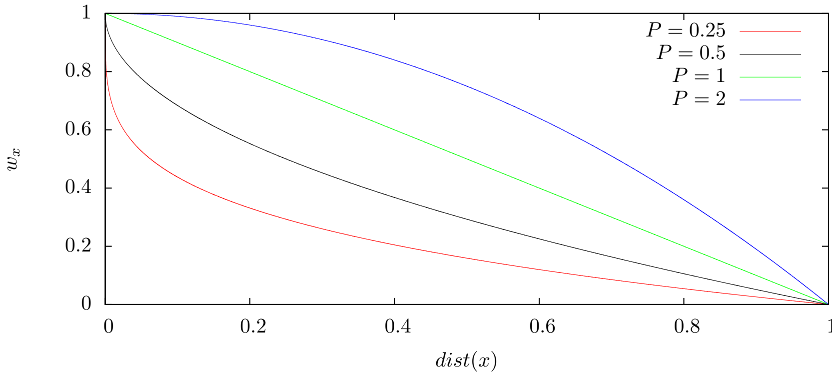


Figure 2.10: Linear distance functions

as steep as the inverse functions, and an increased smoothing effect is to be expected. However, the linear distance functions do not suffer from having very high values in any condition, and are thus very suitable for computer evaluation.

Gaussian distance functions

Yet another very interesting weight function is a Gaussian distance function, which is a function on the form

$$^{\sigma}w_x = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{dist(x)^2}{2\sigma^2}} \tag{2.15}$$

where σ determines the steepness.

We notice in Figure 2.11 that with small σ, these functions can become very steep, while not becoming large enough to cause a problem for floating point systems, or "overshadow" additional terms in the weight function.

2.7 Simple interpolation techniques

What do we do if a voxel in our volume cannot be mapped exactly to a pixel in the input images? This is a problem we can solve using interpolation

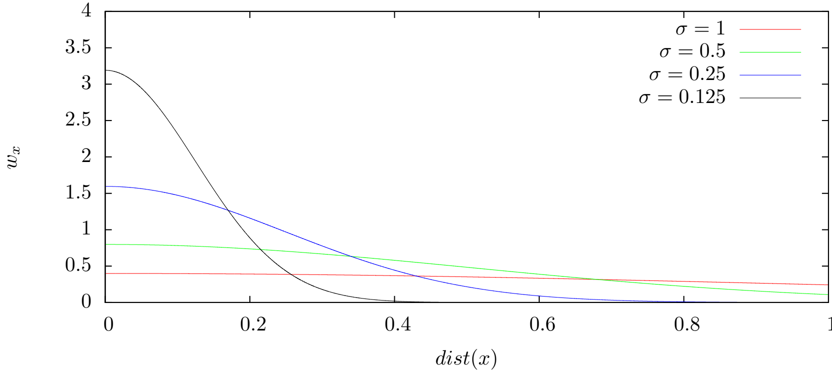


Figure 2.11: Gaussian distance functions

techniques. Interpolation is a set of techniques that solves an *interpolation problem*. An interpolation problem may be defined as follows:

> Given a function $f(\mathbf{x})$ that is only defined at some set of points $\mathbf{P}_f$, we want to estimate the function $F(\mathbf{x})$ which is defined over some interval I with $\mathbf{P}_f \subset I$, and furthermore for any $\mathbf{x}' \in \mathbf{P}_f$, $f(\mathbf{x}') = F(\mathbf{x}')$.

There are many approaches to this, some more complicated than others, but for this thesis we will present two options: Nearest-neighbor interpolation and (bi)linear interpolation.

Nearest-neighbor interpolation

Nearest neighbor interpolation can be defined as follows:

$$\begin{aligned} F_{nearest}(\mathbf{x}) &= nearestNeighborInterpolation(\mathbf{x}, f) \\ &= f(argmin_{\mathbf{x}'}(|\mathbf{x} - \mathbf{x}'|)) \end{aligned} \tag{2.16}$$

Informally, this means simply returning closest defined value.

(Bi)linear interpolation

Linear interpolation is an interpolation scheme where one observes that the point $\mathbf{x}$ lies between some set of points $\mathbf{X} \subset \mathbf{P}_f$, and then blend the sur-

rounding points according to their distance to the point we are interpolating for. Formally, in one dimension:

$$F_{linear}(x) = linearInterpolation(x, f) = \frac{x - x_i}{x_{i+1} - x_i} f(x_i) + \frac{x_{i+1} - x}{x_{i+1} - x_i} f(x_{i+1}) \tag{2.17}$$

where x_i, x_{i+1} are the two points on opposite sides of x, and $x_i, x_{i+1} \in \mathbf{P}_f$.

Extending this to 2D, it becomes

$$\begin{aligned} F_{bi-linear}(x, y) = bilinearInterpolation(x, y, f) = \\ \frac{x - x_i}{x_{i+1} - x_i} \frac{y - y_i}{y_{i+1} - y_i} f(x_i, y_i) \\ + \frac{x_{i+1} - x}{x_{i+1} - x_i} \frac{y - y_i}{y_{i+1} - y_i} f(x_{i+1}, y_i) \\ + \frac{x - x_i}{x_{i+1} - x_i} \frac{y_{i+1} - y}{y_{i+1} - y_i} f(x_i, y_{i+1}) \\ + \frac{x_{i+1} - x}{x_{i+1} - x_i} \frac{y_{i+1} - y}{y_{i+1} - y_i} f(x_{i+1}, y_{i+1}) \end{aligned} \tag{2.18}$$

where (x_i, y_i), (x_i, y_{i+1}), (x_{i+1}, y_i) and (x_{i+1}, y_{i+1}) are the four points surrounding (x, y), and $(x_i, y_i), (x_i, y_{i+1}), (x_{i+1}, y_i), (x_{i+1}, y_{i+1}) \in \mathbf{P}_f$.

2.8 CustusX

CustusX is an image processing and navigation software system developed by SINTEF Medical Technology, Health Research, Trondheim, Norway. It is aimed at physicians for use in both pre-operational planning, and intra-operational imaging and navigation. To that end, it features visualization of medical images (MRI, CT and ultrasound), acquisition of ultrasound data, and spatial tracking of objects, most importantly for this thesis tracking of ultrasound probes.

Most importantly for this thesis, it has routines for acquisition of ultrasound data, tracking data and time stamps together, as well as calibration of the tracking system. Time skews between the data from the different data sources are accounted for – the position data are interpolated such that they match exactly with the ultrasound images.

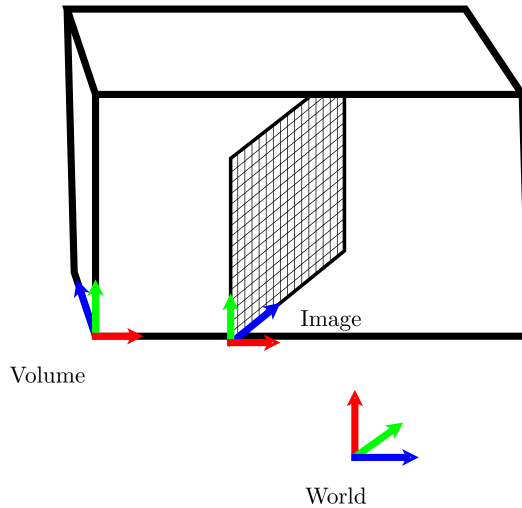


Figure 2.12: CustusX coordinate frames. The *pixel space* is shown as grid lines on the image.

This project uses CustusX as a framework in which the 3D Ultrasound Reconstruction algorithms are implemented. CustusX provides an easy-to-use interface for writing 3D Ultrasound Reconstruction algorithms and is thus very suitable for this thesis.

2.8.1 Coordinate frames

Since *CustusX* features both tracking and imaging, we have to introduce the coordinate systems that are of relevance for this project. In all essence, there are three coordinate frames that are of interest: the *world* coordinate system, the *Volume* coordinate system, and the *Image* coordinate system, visualized in Figure 2.12.

The *World* coordinate frame is based on the fact that the tracking system uses a reference frame which is physically attached to the patient. All the other coordinate systems are defined relative to this coordinate system.

The *Volume* coordinate frame is provided by CustusX, and oriented such that the volume extends from (0,0,0) to (S_x, S_y, S_z). The reconstruction algorithms work within this coordinate frame.

The *Image* coordinate frame is also provided by the tracking system. By *Image* space, we mean the space of each ultrasound image, meaning there are as many *Image* spaces as there are images. The ultrasound probe has a tracker attached to it which enables the tracking system to provide the transformation from the *Image* coordinate system to the *World* coordinate system, ${}^{W}\mathbf{M}_I$. CustusX pre-processes these transformations prior to reconstruction, and provides the transformation from the *Image* coordinate system to the *Volume* coordinate system, ${}^{V}\mathbf{M}_I$, to the reconstruction algorithm. The image space is oriented such that the origin is at the lower left of the image, and the axes are oriented as follows:

- The X axis points towards the right of the image
- The Y axis points downwards (away from the probe)
- The Z axis points out of the image

To transform from *Image* space to *Pixel space* we need to know the size of the pixels from the scanner. The scanner provides us with this data, S_x and S_y. So, for a point ${}^{I}\mathbf{P}$ in *Image* space, in order to transform it to the same point ${}^{P}\mathbf{P}$ in pixel space, we take the X and Y (the Z coordinate is zero for any point in the image space that is actually part of the image) coordinates and scale them such that ${}^{P}P_x ={}^{I} P_x/S_x$, and ${}^{P}P_y ={}^{I} P_y/S_y$.

We have now seen the background knowledge required to understand the problem of 3D Ultrasound Reconstruction well enough to embark on the task of designing a new algorithm, which we will do in the next chapter.

Chapter 3

Method

This chapter describes the work that has been done for this thesis, and is divided into two sections: The first section describes the VGDW algorithm and its sub-algorithms, and the second section describes the evaluation techniques employed for this thesis.

3.1 3D Reconstruction algorithm

We will now present the VGDW algorithm. We will first see an overview of the algorithm, in which we will describe the sub-problems it consists of, and we will then inspect how the algorithm solves those sub-problems.

3.1.1 Overview

The algorithm is voxel-based according to the classification made by [Solberg et al., 2007], meaning that it asks the question "What data should go in this voxel?" for every voxel in the target volume. The first problem one needs to solve to answer that question is "What input data do I have that is relevant to that voxel?" – or more formally: Find the set of Image Planes ${}^{R}\mathbf{I}_v$ such that

$$ {}^{R}\mathbf{I}_v = \{\mathbf{I} \mid \mathbf{I} \in \mathbf{IMAGES} \wedge dist(\mathbf{I}, v) < R\} \tag{3.1} $$

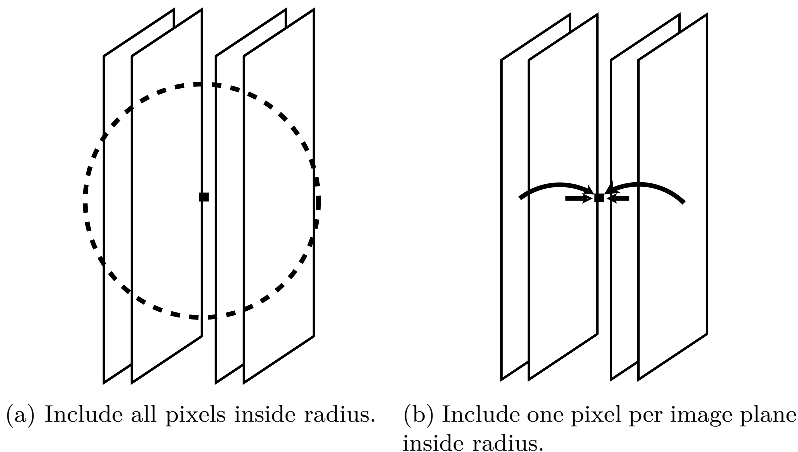
where R is the maximum allowed distance, v is the target voxel, **IMAGES** is the set of all image planes, and $dist(\mathbf{I}, v)$ denotes the distance between the voxel v and the image plane I as described in Section 2.4.4.

Once the set ${}^{R}\mathbf{I}_v$ is known, one needs to find the set of pixel values ${}^{R}\mathbf{PIXELS}_v$ relevant to the voxel. Different approaches are possible here. One approach as described by [Barry et al., 1997] is to take all pixel values closer than the radius R, giving:

$$ \begin{aligned} &{}^{R}\mathbf{PIXELS}_v = \\ &\{PIXEL \mid PIXEL \in \mathbf{I} \wedge \mathbf{I} \in {}^{R}\mathbf{I}_v \wedge dist(PIXEL, v) < R\} \end{aligned} \tag{3.2} $$

where ${}^{R}\mathbf{PIXELS}_v$ is the set of pixels relevant to the voxel v, $\mathbf{I}$ represent a single image plane, ${}^{R}\mathbf{I}_v$ is the set of relevant image planes as described above, R is the radius as above, and $dist(PIXEL, v)$ denotes the distance between the pixel $PIXEL$ and the voxel v. This is illustrated in Figure 3.1a. A problem with this approach is that $|{}^{R}\mathbf{PIXELS}_v|$ becomes quite large, eating at both memory and computational resources.

Thankfully, a trade-off exists. [Coupé et al., 2005] and [Miller et al., 2012], among others, assumes that the 2D images are quite nicely defined already, so no little to no interpolation is required in the directions perpendicular to



(a) Include all pixels inside radius. (b) Include one pixel per image plane inside radius.

Figure 3.1: Illustration of pixel inclusion strategies

the plane. Under that assumption, it makes sense to only include one value from each image. There are different approaches to determining which value to take. The simplest approach is to simply take the closest pixel – this is what is done in the VNN2 algorithm summarized by Miller et.al. This approach is illustrated in Figure 3.1b. Another approach is to project the voxel orthogonally onto the image plane and perform bi-linear interpolation to determine the value – this is what is done in the variant of DW summarized by Miller et.al. Finally, Coupé et al. proposes a method called Probe Trajectory – where one attempts to estimate the position of the probe at the voxel, and perform bi-linear interpolation from there.

Out of these three method, the DW variant summarized by Miller et.al. was chosen due to the low complexity while still giving good results. A future project may investigate the use of probe trajectory estimation in the context of this algorithm. Remembering that we are discussing how to find the set ${}^{R}\mathbf{PIXELS}_v$, we can define this variant formally as

$$ {}^{R}\mathbf{PIXELS}_v = \{x' \mid x' = bilinearInterpolation({}^{\mathbf{I}}\mathbf{P}_v, \mathbf{I})\} \tag{3.3} $$

where

$$ {}^{\mathbf{I}}\mathbf{P}_v = {}^{\mathbf{I}}\mathbf{M}_V projectOrthogonallyOnto(v, \mathbf{I}) $$

and

$$ \mathbf{I} \in {}^{R}\mathbf{I}_v $$

where $bilinearInterpolation(^{\mathbf{I}}\mathbf{P}_v, \mathbf{I})$ is the value returned from bi-linear interpolation(Equation 2.18), $^{\mathbf{I}}\mathbf{M}_V$ is the transformation matrix transforming from voxel space to image space as discussed in Section 2.8.1, and $projectOrthogonallyOnto(v, \mathbf{I})$ is the orthogonal projection of v onto the image plane $\mathbf{I}$ as described in Equation 2.9 (this implies that $^{\mathbf{I}}\mathbf{P}_v.z = 0$). This results in a small $|^R\mathbf{PIXELS}_v|$, in fact $|^R\mathbf{PIXELS}_v| = |^R\mathbf{I}_v|$.

The final step of the algorithm is to compute some voxel value based on $^R\mathbf{PIXELS}_v$. A very common approach to this is to compute some weighted average (see Section 2.5), using some weight function (see Section 2.6)

We have now seen an overview of the sub-problems, as well as hinted at some possible solutions to them. We will now proceed to describe in detail how the sub-problems are solved by the algorithm.

3.1.2 Finding the closest image planes: Restricted multi-start local search

Determining the set $^R\mathbf{PIXELS}_v$ accurately depends very strongly on determining the set $^R\mathbf{I}_v$ accurately. Determining the set $^R\mathbf{I}_v$ may seem a simple problem – one can simply perform a linear search to find the set of planes that have a distance to the voxel v smaller than R. This approach has two problems: Firstly, it has no upper limit on how many image planes are included, potentially leading to a very large set of image planes, which is a problem with regards to memory consumption. Secondly, it is very slow. In a data set with 1000 image planes, surely not all 1000 image planes are relevant to a given voxel, but in this approach they would still have to be evaluated.

[Ludvigsen, 2010] used a scheme in which one keeps track of which plane was the closest for the previous voxel computed, and searches for close planes that are close to the previous plane in the plane array. The problem with this is that it will stay inside a local minimum – this fails if multiple sweeps has been performed.

[Wein et al., 2006] proposes a sophisticated scheme where one traverses the volume in a way such that the any voxel v_{i+1} is at most some distance D_{max} away from the previous voxel, v_i, and keeps track of the planes that may be relevant for that voxel using a rotation queue. This scheme can be proved to always find the set of slices with the shortest distance, but it is quite costly in terms of computation time and memory usage.

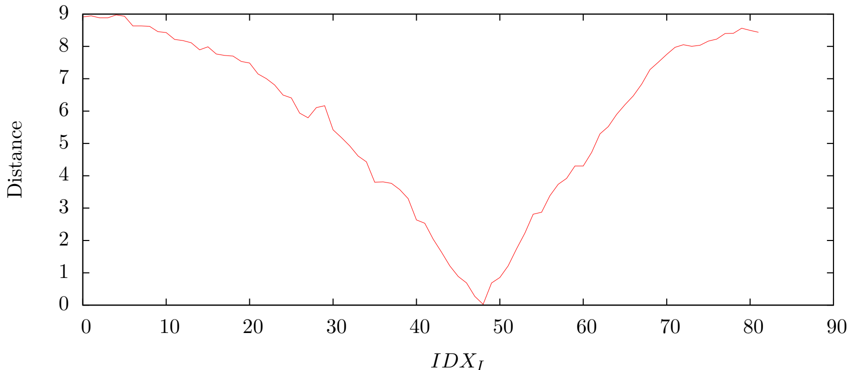


Figure 3.2: Distance function for actual data acquired by a single sweep. A voxel in the middle of the volume was selected as v

Assuming the image planes are given in the same order as they are acquired, a set of image planes acquired by a single, simple sweep can be expected to yield a distance function that looks something like Figure 3.2.

We observe from Figure 3.2 that the data is relatively nicely structured forming a 'V' with a clearly defined minimum. From this it is easy to deduce that once one has found the minimum for some voxel v, some voxel v' neighboring voxel v is going to have its minimum located close by due to the nature of the distance function, as suggested by [Ludvigsen, 2010]. This holds true for any set of image planes which has been acquired by only a simple image sweep. However, if the image planes has been acquired by a more complex sweeping pattern, the data may look more like it does in Figure 3.3. In that case, the voxel v was passed more than one time, leading to multiple minima being available for the voxel. The above mentioned method will not be able to find more than one minima.

Again we make a trade-off to gain performance, and propose using the method used by Ludvigsen et.al, but keeping multiple guesses – and thus not limiting ourselves to one local minimum. We draw some inspiration from Wein et.al as keeping locality between the voxels being traversed is a good idea.

We will now proceed to describe the employed algorithm, which is in three parts. We divide the volume into cubes, and for each cube we find a set of initial guesses. The cubes are the units of parallelization for the GPU

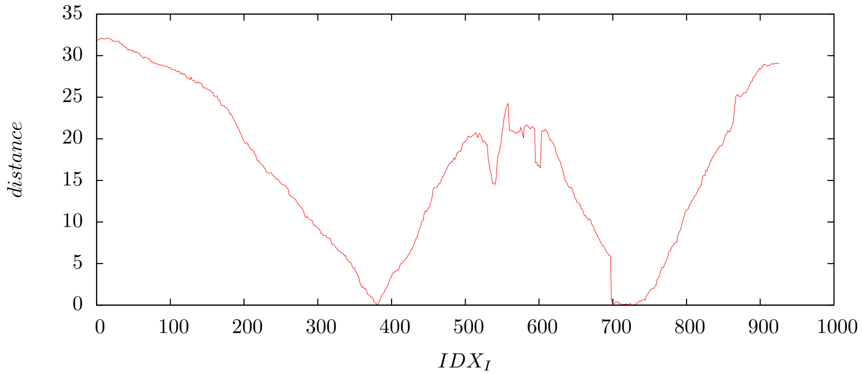


Figure 3.3: Distance function for actual data acquired by U-turn scanning. A voxel in the middle of the volume was selected as v

implementation, meaning that this scheme will be applied for all cubes in the volume.

Algorithm 1 is designed to find a suitable set of initial guesses for the initial voxel in the given cube. Algorithm 2 is designed to find the closest set of planes close to the given guess, and finally Algorithm 3 applies Algorithm 2 to each of the guesses found by Algorithm 1, and updates the guesses.

Algorithm 1 Finding the initial guesses for the multi-start local search

findStartGuesses(**v**, **IMAGES**, D_{max}):

Input:	**v**	The initial voxel
	IMAGES	The set of image planes
	D_{max}	The threshold for declaring a local minimum
	G_{max}	The maximum number of guesses
Output:	**GUESSES**	A set of guesses for multi-start local search

```
1: procedure findStartGuesses(v, IMAGES, D_max)
2:     GUESSES[0] ← 0
3:     hasHighSinceLastTaken ← true
4:     i_prev ← 0
5:     N ← 1
6:     for i ∈ [0, |IMAGES| − 1] do
7:         D_i ← |dist(v, IMAGES[i])|
8:         if D_i < D_max then
9:             if ¬hasHighSinceLastTaken then
10:                if D_i < |dist(v, IMAGES[GUESSES[i_prev]])| then
11:                    GUESSES[i_prev] ← i
12:                    hasHighSinceLastTaken ← false
13:                else if N < G_max then
14:                    GUESSES[N] ← i
15:                    i_prev ← N
16:                    hasHighSinceLastTaken ← false
17:                    N ← N + 1
18:                else
19:                    hasHighSinceLastTaken ← false
20:                    G_biggest = argmax_{j∈[0,G_max−1]}(
                        dist(v, IMAGES[GUESSES[j]]))
21:                    if |dist(v, IMAGES[G_biggest])| > D_i then
22:                        GUESSES[G_biggest] ← i
23:                        i_prev ← G_biggest
24:                    end if
25:                end if
26:            end if
27:        else
28:            hasHighSinceLastTaken ← true
29:        end if
30:    end for
31:    return GUESSES
32: end procedure
```

Algorithm 1 is an algorithm that finds a set **GUESSES** that are all located in different "valleys" in the search space. A valley in this case is detected as follows: For each distance value V_i with index i, we have

$$\begin{aligned} VALLEY &= \\ \{V_i \quad & \mid V_i < D_{max} \\ & \wedge \exists i_{start}, i_{stop} \quad : \forall i \in \mathbf{INDICES}_v : i \in [i_{start}, i_{stop}] \\ & \qquad \wedge \forall j \in [i_{start}, i_{stop}] : j \in \mathbf{INDICES}_v \} \end{aligned}$$

where $\mathbf{INDICES}_v$ is the set of indices included in the valley. In other words, an uninterrupted stretch of values in the array where all the values are smaller than D_{max}. Further, the guess that is stored for a valley will be the smallest value inside the value, i.e.

$$guess_{VALLEY} = argmin_i(V_i \in VALLEY)$$

The algorithm keeps four helper variables:

- $hasHighSinceLastTaken$ is true if there was a value $V_j > D_{max}$ after the previous time a minimum was included.
- i_{prev} is the index into the **GUESSES** array of the previously taken minimum
- N is the number of minima that has been taken, i.e. $|\mathbf{GUESSES}|$
- D_i is the distance from the current image plane to the voxel.

The algorithm works as follows: it iterates over all the image planes, and when it finds a value $V_i < D_{max}$, there are three possible cases:

1. The previous index considered was also smaller than D_{max}, i.e. $V_{i-1} < D_{max}$ (line 9) In that case, we are inside the same valley as for index $i-1$, and we simply need to determine if this guess is better than the previously stored guess. If so, we store it.
2. The previous index was bigger than D_{max}, meaning we have encountered a new valley, and we have enough room for at least one more minimum (line 13). In this case, we store i into the next position in the **GUESSES** array, set i_{prev} to N and then increment N.
3. The previous index was bigger than D_{max}, meaning we have encountered a new valley, but there is no more room in the **GUESSES** array(line 18). In this case, we toss out the worst minimum – the one with the longest distance, and include this one instead, but only if it is smaller than the one with the longest distance.

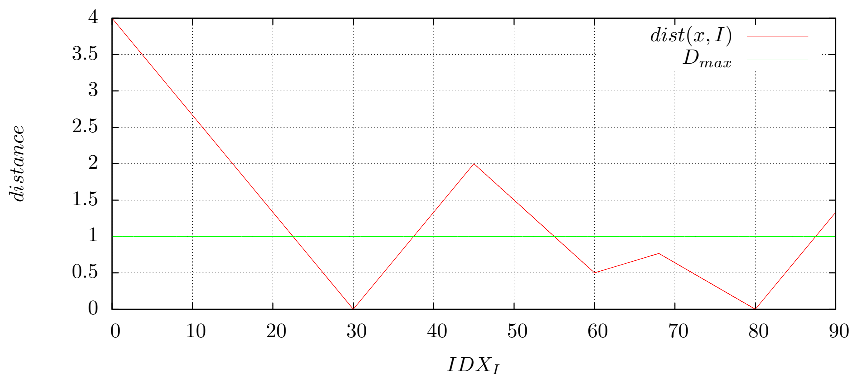


Figure 3.4: Figure to illustrate Algorithm 1. IDX_I refers to the index of image I.

The algorithm may also be explained in terms of Figure 3.4: The algorithm starts at $IDX_I = 0$ and follows the red line towards the right. Around $IDX_I = 22 - 23$, the distance function crosses the green line, D_{max}, and the algorithm starts recording a minimum. The stored minimum will be overwritten all the way until $IDX_I = 30$, where the distance function is at its lowest. The algorithm then continues to move to the right, but no minima better than the one found are found within this valley. Eventually, the distance becomes greater than D_{max} around $IDX_I = 37 - 38$. At this point, $hasHighSinceLastTaken$ becomes true, so that when the distance function at $IDX_I = 55$ and the distance function once more becomes smaller than D_{max}, a new minimum is stored. This minimum is overwritten all the way until $IDX_I = 60$, at which point the distance function starts increasing again. However, at IDX_I it starts decreasing again, before hitting D_{max}. Thus, the minimum at $IDX_I = 80$ will eventually overwrite the minimum at $IDX_I = 60$. The reason for this behavior is to consistently select the best minimum in a noisy data set.

Having a set **GUESSES** of good guesses, we may now proceed to the plane selection algorithm. We will describe the algorithm in the context of having just a single guess – observing that a variant for multiple guesses can be described in terms of an algorithm with a single guess.

Algorithm 2 Find close planes that are close to the guess, single start

FINDCLOSESTPLANESSINGLE($\mathbf{v}, \mathbf{IMAGES}, radius, N_{max}, guess, {}^{R}\mathbf{I}_v$):

Input:	$\mathbf{v}$	The voxel to find close planes for
	$\mathbf{IMAGES}$	The set of image planes definitions
	$radius$	The maximum distance between voxel and plane
	N_{max}	The maximum number of image planes to return
	$guess$	Index into **IMAGES** of the guessed image plane
	${}^{R}\mathbf{I}_v$	Set of close planes in which to store results, may already contain other close planes, or dummy planes with $dist(v, plane) = \infty$
Output:	${}^{R}\mathbf{I}_v$	The set of close planes found
	N_{found}	The number of close planes found
	$i_{smallest}$	The index of the closest found plane (i.e. the next guess)

1: **procedure** FINDCLOSESTPLANESSINGLE($\mathbf{v}$, $\mathbf{IMAGES}$, $radius$, N_{max}, $guess$, ${}^{R}\mathbf{I}_v$)
2: $found \leftarrow 0$
3: $done_{up} \leftarrow false$
4: $done_{down} \leftarrow false$
5: $i_{smallest} \leftarrow guess$
6: $N_{found} \leftarrow 0$
7: $D_{term} \leftarrow clamp(|dist(v, \mathbf{IMAGES}[guess]), radius, 3 * radius)$
8: $i_{max} \leftarrow argmax_i(|dist(v, {}^{R}\mathbf{I_v}[i])|)$
9: $D_{max} \leftarrow min(|dist(v, \mathbf{IMAGES}[i_{max}])|, radius)$
10: $i \leftarrow 0$
11: **while** $\neg done_{up} \vee \neg done_{down}$ **do**
12: $i_{up} = min(guess + i, |\mathbf{IMAGES}| - 1)$
13: $i_{down} = max(guess - i - 1, 0)$
14: **if** $\neg done_{down} \wedge |dist(v, \mathbf{IMAGES}[i_{down}])| < D_{max}$ **then**
15: ${}^{V}\mathbf{P}_v \leftarrow projectOrthotonallyOnto(v, \mathbf{IMAGES}[i_{down}])$
16: ${}^{\mathbf{I}}\mathbf{P}_v \leftarrow toImageCoord({}^{V}\mathbf{P}_v, \mathbf{IMAGES}[i_{down}])$
17: **if** $isValidPixel({}^{\mathbf{I}}\mathbf{P}_v, \mathbf{IMAGES}[i_{down}])$ **then**
18: ${}^{\mathbf{R}}\mathbf{I_v}[i_{max}] \leftarrow \mathbf{IMAGES}[i_{down}]$
19: $i_{max} \leftarrow argmax_i(|dist(v, {}^{R}\mathbf{I_v}[i])|)$
20: $D_{max} \leftarrow min(|dist(v, \mathbf{IMAGES}[i_{max}])|, radius)$
21: $N_{found} \leftarrow N_{found} + 1$
22: **if** $|dist(v, \mathbf{IMAGES}[i_{down}])| <$
23: $|dist(v, \mathbf{IMAGES}[i_{smallest}])|$ **then**
24: $i_{smallest} \leftarrow i_{down}$
25: **end if**

Algorithm 2 Find close planes that are close to the guess (continued)

26: **end if**
27: **end if**
28: **if** $\neg done_{up} \wedge |dist(v, \mathbf{IMAGES}[i_{up}])| < D_{max}$ **then**
29: ${}^{V}\mathbf{P}_v \leftarrow projectOrthotonallyOnto(v, \mathbf{IMAGES}[i_{up}])$
30: ${}^{\mathbf{I}}\mathbf{P}_v \leftarrow toImageCoord({}^{V}\mathbf{P}_v, \mathbf{IMAGES}[i_{up}])$
31: **if** $isValidPixel({}^{\mathbf{I}}\mathbf{P}_v, \mathbf{IMAGES}[i_{up}])$ **then**
32: ${}^{\mathbf{R}}\mathbf{I_v}[i_{max}] \leftarrow \mathbf{IMAGES}[i_{up}]$
33: $i_{max} \leftarrow argmax_i(|dist(v, {}^{R}\mathbf{I_v}[i])|)$
34: $D_{max} \leftarrow min(|dist(v, \mathbf{IMAGES}[i_{max}])|, radius)$
35: $N_{found} \leftarrow N_{found} + 1$
36: **if** $|dist(v, \mathbf{IMAGES}[i_{up}])| <$
37: $|dist(v, \mathbf{IMAGES}[i_{smallest}])|$ **then**
38: $i_{smallest} \leftarrow i_{up}$
39: **end if**
40: **end if**
41: **end if**
42: $done_{up} \leftarrow done_{up} \vee |dist(v, \mathbf{IMAGES}[i_{up}])| > D_{term} \vee i_{up} = |\mathbf{IMAGES}|$
43: $done_{down} \leftarrow done_{down} \vee |dist(v, \mathbf{IMAGES}[i_{down}])| > D_{term} \vee i_{down} = 0$
44: **end while**
45: **return** ${}^{R}\mathbf{I}_v, N_{found}, i_{smallest}$
46: **end procedure**

Algorithm 2 works by starting at the guess and searching the array in both directions for a closer plane. If one is found, it is inserted into ${}^{R}\mathbf{I}_v$ in the place of the plane in ${}^{R}\mathbf{I}_v$ with the largest distance to the voxel. Note that this works even if ${}^{R}\mathbf{I}_v$ already contains a set of planes – that set will just be improved upon. The algorithm also keeps track of the index of the closest plane overall – this is in order to give feedback so that the guess may be updated for the next voxel.

There are two things that can cause the algorithm to stop searching in a specific direction:

1. It hits the end or beginning of the plane array.
2. It hits a plane that is too far away to be of further interest, i.e. the distance from the voxel to the plane is greater than D_{term}

In the interest of performance, it is vital to keep D_{term} as low as possible. Though, it cannot be too small. We therefore use the distance of the guess to the voxel to determine it, but we clamp it to $[radius, 3 * radius]$ so that it never gets too small to find any nearby planes that are inside the radius. The upper limit, $3 * radius$ is there to avoid searching for a long time from a bad guess, but must be high enough to accommodate for noisy data.

We also make sure that the voxel maps to a pixel actually on the image plane in question – the mathematical definition of a plane is infinitely stretched, but an image plane has limits.

Having seen the single start search variant, we will now define the multi-start search, Algorithm 3 in terms of Algorithm 2.

Algorithm 3 Find close planes that are close to guesses, multi-start

FINDCLOSESTPLANESMULTI($\mathbf{v}, \mathbf{IMAGES}, radius, N_{max}, \mathbf{GUESSES}_v$):

Input:	$\mathbf{v}$	The voxel to find close planes for
	IMAGES	The set of image planes definitions
	radius	The maximum distance between voxel and plane
	N_{max}	The maximum number of image planes to return
	$\mathbf{GUESSES}_v$	Indices into **IMAGES** of the guessed image planes close planes.
Output:	${}^R\mathbf{I}_v$	The set of close planes found
	N_{found}	The number of close planes found

1: **procedure** FINDCLOSESTPLANESMULTI($\mathbf{v}$, **IMAGES**, *radius*, N_{max}, $\mathbf{GUESSES}_v$):
2: ${}^R\mathbf{I}_v \leftarrow N_{max}$ dummy planes with $dist(v, dummy) = \infty$
3: $N_{found} \leftarrow 0$
4: **for** $i \in [0, |\mathbf{GUESSES}_v| - 1]$ **do**
5: ${}^R\mathbf{I}_v, N_i, i_{smallest} = findClosestPlanesSingle(\mathbf{v}, \mathbf{IMAGES}, radius, N_{max}, \mathbf{GUESSES_v}[i], {}^R\mathbf{I}_v)$
6: **if** $N_{found} > 0$ **then**
7: $\mathbf{GUESSES_v}[i] \leftarrow i_{smallest}$
8: **end if**
9: $N_{found} \leftarrow N_{found} + N_i$
10: **end for**
11: **return** ${}^R\mathbf{I}_v, min(N_{found}, N_{max})$
12: **end procedure**

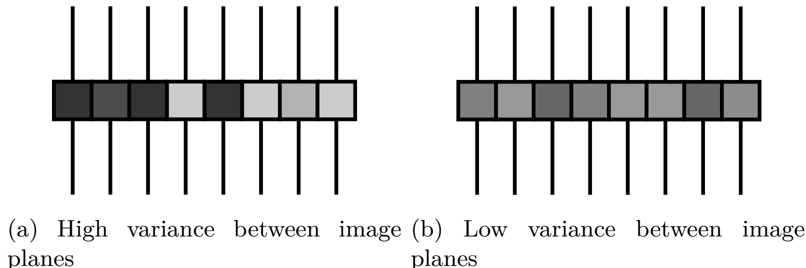
As we see, Algorithm 3 simply applies Algorithm 2 once for each guess, keeping the same ${}^R\mathbf{I}_v$ between the applications, while updating the guesses. Thus, at the end, ${}^R\mathbf{I}_v$ contains the closest N_{max} image planes found from all the guesses, provided that there exists at least N_{max} planes with distance smaller than *radius* that is reachable from the guesses.

We have now seen an efficient way to select the image planes without sacrificing too much flexibility in terms of probe movement patterns. We now proceed to describe how the algorithm determines the voxel value given ${}^R\mathbf{I}_v$.

3.1.3 Determining the voxel value: VGDW

This section describes the process of determining a voxel value from the image planes located by the algorithms described in the previous section. The algorithm described here is what constitutes the VGDW method; the adaptive reconstruction method that adjusts its smoothness according to the variance of the input data, drawing inspiration from AGDW[Huang et al., 2009] and anisotropic diffusion.

We must first find the ${}^{R}\mathbf{PIXELS}_v$, recalling that this is the set of pixels relevant to the voxel v from the set ${}^{R}\mathbf{I}_v$. We do this using bi-linear equation, which we have already defined in Equation 3.3.



(a) High variance between image planes

(b) Low variance between image planes

Figure 3.5: High and low variance between image planes

Now, once ${}^{R}\mathbf{PIXELS}_v$ is determined, we want to find some appropriate voxel value. We do this by computing a weighted average as described in Section 2.5. Remembering that it is of interest to keep important detail but blur noise, we use a Gaussian weight function with respect to the distance of the pixel to the voxel, as described as Section 2.6, and vary the parameter σ with the variance of ${}^{R}\mathbf{PIXELS}_v$. With high variance, we want a narrow Gaussian to preserve detail, illustrated in Figure 3.5a, meaning we want a small σ. With low variance as illustrated in Figure 3.5b, we want a wide Gaussian to smooth out noise. The weight function then becomes

$$W_{dist}(x) = \frac{1}{\sigma_G\sqrt{2\pi}} e^{-\frac{dist(v,x)^2}{2\sigma_G^2}} \tag{3.4}$$

with $\sigma_G = clamp(K/\sqrt{Var({}^{R}\mathbf{PIXELS}_v)}, \sigma_{min}, \sigma_{max})$, and K is some experimentally determined constant – higher K giving more smoothing. Through-

out this thesis, K was set to $K = 32$ – exactly how this parameter behaves remains to be investigated.

We also desire good behavior in the case of value conflicts from multiple sweeps. Taking inspiration from earlier approaches for dealing with value collisions in PNN(see Section 2.1.3); we expand our weight function to consider two more properties: how "late" the ultrasound image was taken (i.e. its position in the **IMAGES** array, and the brightness of the pixel. Equation 3.5 shows the brightness term, and Equation 3.6 shows the lateness term.

$$W_{brightness}(x) = \begin{cases} B & \text{if } intensity(x) > E(intensity({}^{R}\mathbf{PIXELS}_v)) \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

where B is a parameter determining how heavy to weigh intensity, x is a pixel, $intensity(x)$ is the intensity value of pixel x, and $E(intensity({}^{R}\mathbf{PIXELS}_v)$ is the average intensity of the pixels in ${}^{R}\mathbf{PIXELS}_v$.

$$W_{lateness}(x) = \begin{cases} L & \text{if } index(x) > E(index({}^{R}\mathbf{PIXELS}_v)) \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

where L is a parameter determining how heavy to weigh "lateness", x is a pixel, $index(x)$ is the index of the image plane containing the pixel x in **IMAGES** , and $E(index({}^{R}\mathbf{PIXELS}_v))$ is the average index into **IMAGES** of the pixels in ${}^{R}\mathbf{PIXELS}_v$.

The complete weight function then becomes

$$\begin{aligned} W(x) &= W_{dist}(x) + W_{brightness}(x) + W_{lateness}(x) \\ &= \frac{1}{\sigma_G\sqrt{2\pi}} e^{-\frac{dist(v,x)^2}{2\sigma_G^2}} \\ &+ \begin{cases} B & \text{if } intensity(x) > E(intensity({}^{R}\mathbf{PIXELS}_v)) \\ 0 & \text{otherwise} \end{cases} \\ &+ \begin{cases} L & \text{if } index(x) > E(index({}^{R}\mathbf{PIXELS}_v)) \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.7}$$

and finally, the voxel value becomes

$$V = \Big(\sum_{x \in {}^{R}\mathbf{PIXELS}_v} W(x) \Big)^{-1} \sum_{x \in {}^{R}\mathbf{PIXELS}_v} W(x) intensity(x) \tag{3.8}$$

In total, this leaves us with 5 parameters concerning the computation of a voxel value: K, σ_{min}, σ_{max}, B and L.

3.1.4 OpenCL Implementation

We have seen the techniques used to compute a single voxel value, but we are of course required to compute an entire volume. This is where the parallelism is exposed. As mentioned, the OpenCL implementation uses cubes in the target volume as units of parallelization. The choice of using cubes is to ensure an upper bound on the distance between any two voxels computed by the same thread, meaning we can find a small value for D_{max} for Algorithm 1. A set of guesses is computed for each thread. The threads iterate over the voxels in the cube in a zig-zag-scheme, ensuring that the next voxel always is a neighbor of the previous voxel, as suggested by [Wein et al., 2006].

We keep plane equations in local memory, as well as the planes found by Algorithm 3, as these are data that are accessed frequently throughout the algorithm. We also use image objects for the image data, allowing us to leverage bi-linear interpolation in hardware, as well as the texture cache.

3.2 Evaluation

In this section we will investigate the evaluation techniques employed in this thesis. Since the VGDW algorithm is new, we need a thorough evaluation of its reconstruction quality and computational speed to see how it compares to VNN, VNN2, DW, and in some cases PNN. VNN, VNN2 and DW were implemented in OpenCL similarly to VGDW, sharing the plane search algorithm.

This section is organized as follows: First we will describe the data sets we have used in the evaluation, since they will be referred to during the description of the evaluation methods themselves. Then we will proceed to describe the evaluation techniques employed – 4 image quality evaluations, and a computational performance evaluation.

3.2.1 The data sets

Five data sets of varying character was used for the evaluation. We will proceed to describe them here.

Phantom The "Phantom" data set is a small data set obtained by linearly scanning a phantom filled with water. The phantom consists of a single rod. This data set has a total of 85 recorded slices. The data set was obtained using a NDI Polaris Spectra tracking system, and an Ultrasonix L14-5 probe.

Angio The "Angio" data set is a medium sized data set obtained by a linear scan on a real patient. The data set contains both angio- and B-mode data. The visual quality of this data set is quite low, so the data set has been used only for performance measurements. This data set has a total of 623 recorded slices. The data set was obtained using a NDI Polaris Spectra tracking system, and a GE Vingmed 11L probe.

Tumor The "Tumor" data set is a large data set depicting a tumor from a real patient. The images were obtained using an U-turn scan pattern, such that many slices overlap. This data set has a total of 932 recorded slices. The data set was obtained using a NDI Polaris Spectra tracking system, and a GE Vingmed 11L probe.

Simulated from ultrasound This data set was obtained by simulating an ultrasound acquisition over an existing ultrasound volume. Real tracking data was used to generate the positions of the image slices, and the slices were obtained by tri-linear interpolation from the volume. The tracking system used was NDI Polaris Spectra. The volume itself was generated by tracked freehand ultrasound acquisition and reconstructed using PNN.

Simulated from MRI The data set simulated from ultrasound was generated exactly in the same way as the "Simulated from ultrasound" data set mentioned above, but the original data was in this case an MRI volume. Since this volume was obtained from MRI data, there is very little noise present in the volume. The tracking system used was NDI Polaris Spectra.

3.2.2 Reconstruction from existing volume

An US acquisition was simulated using real tracking data from an existing volume, and reconstruction was performed on this simulation. The existing volume was then cropped and aligned to the reconstructed volume using tri-linear interpolation, and RMS error was computed across the whole volume. Figure 3.6 depicts this process. Aligned slices were extracted from the volumes and visualized. This was done using two data sets; one ultrasound volume and one MRI volume, i.e. the two "simulated" volumes mentioned above. The parameters used may be seen in Table 3.1 and Table 3.2.

This is a novel evaluation technique, and the idea behind it is that a good reconstruction algorithm should be able to approximate the original volume as closely as possible, and the error can be measured since the original volume the slices are sampled from is known. An important component in this technique is that positions of the slices come from real tracking data – an algorithm that handles the problem of tracking noise well gets rewarded in this evaluation technique. Still, one important thing to note is that the tracking noise behaves slightly differently in this evaluation than in usual contexts: since the tracking data determines the position of the slices to sample, there is no error between the tracking data and the slices – the position of the sampled slices match the tracking data perfectly, which is not the case in actual acquisition.

Parameter	**Value**
Input pixel spacing	0.0478
Output voxel spacing	0.123
Volume dimensions	374x320x278
Radius (All methods except PNN)	1.0
Distance in voxels (PNN Only)	5
Number of input slices	228
G_{max}	1
σ_{max} (VGDW only)	32.0
σ_{min} (VGDW only)	$3.2 \cdot 10^{-6}$
K (VGDW only)	32.0
B (VGDW only)	0.0
L (VGDW only)	0.0

Table 3.1: Parameters for reconstruction from existing ultrasound volume evaluation.

Parameter	**Value**
Input pixel spacing	0.0810
Output voxel spacing	0.154
Volume dimensions	303x390x282
Radius (All methods except PNN)	1.0
Distance in voxels (PNN Only)	3
Number of input slices	272
G_{max}	1
σ_{max} (VGDW only)	32.0
σ_{min} (VGDW only)	$3.2 \cdot 10^{-6}$
K (VGDW only)	32.0
B (VGDW only)	0.0
L (VGDW only)	0.0

Table 3.2: Parameters for reconstruction from existing MRI volume evaluation.

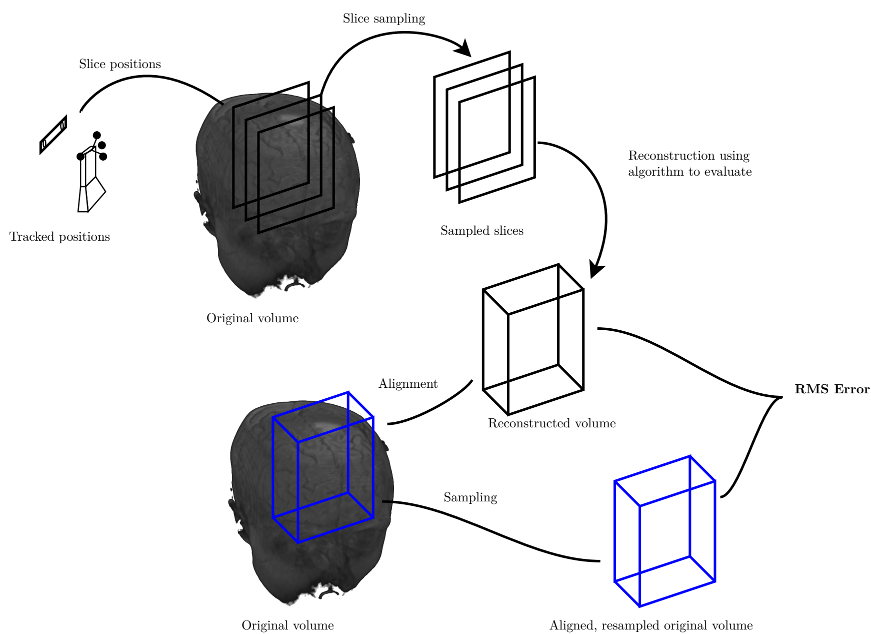


Figure 3.6: An illustration of the process used in the "simulated ultrasound from existing volume" evaluation

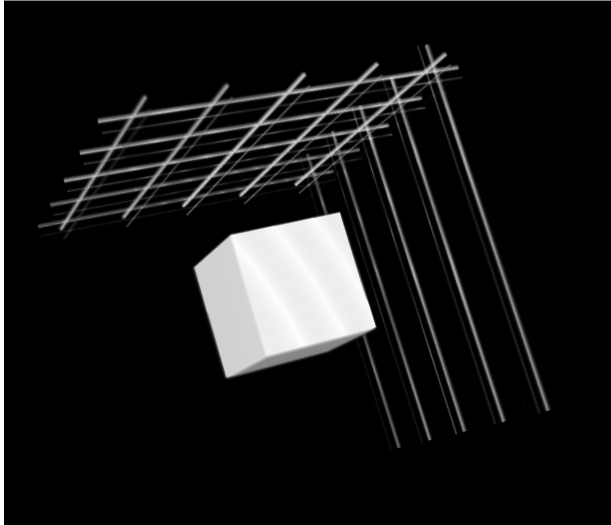


Figure 3.7: 3D rendering of the synthetic volume. Rendered by CustusX.

3.2.3 Synthetic volume

A volume was defined programatically. US acquisition was simulated with a linear scan pattern, tilted scan pattern with a total of 30 degrees of rotation through the scan, and linear scan pattern with additive Gaussian noise with $\sigma = 5$ and $\mu = 0$. The voxel spacing was twice that of the sample spacing. RMS error was computed across the whole volumes, and slices were extracted for visual inspection. The volume has a background color value of `10`, and has lines in X, Y and Z dimensions with value `255` with three thicknesses. It also has a large cube in the middle of the volume. Figure 3.7 shows a 3D rendering of the volume.

The idea of this novel evaluation technique is similar to the idea of the simulated ultrasound from existing volume evaluation technique – extracting slices from a known volume, and comparing the resulting reconstruction with the original volume. This technique, however, is entirely synthetic, and allows the volume to be designed in such a way that it demonstrates different aspects of the different algorithms. Especially the scan with additive noise is designed to demonstrate the noise suppression capabilities of VGDW.

The reconstruction parameters may be seen in Table 3.3.

Parameter	**Value**
Input pixel spacing	0.05
Output voxel spacing	0.10
Volume dimensions	300x300x300
Radius (All methods except PNN)	1.0
Distance in voxels (PNN Only)	3
Number of input slices	100
G_{max}	1
σ_{max} (VGDW only)	32.0
σ_{min} (VGDW only)	$3.2 \cdot 10^{-6}$
K (VGDW only)	32.0
B (VGDW only)	0.0
L (VGDW only)	0.0

Table 3.3: Parameters for Synthetic volume evaluation.

3.2.4 Slice removal

The data set "Phantom" was used for this evaluation. The volume was reconstructed with 0, 1, 3 and 5 input slices removed in the middle of the data, with the voxel grid aligned such that the middle slice falls exactly onto voxels. The RMS Error of the reconstructed middle slice and the middle slice of the input was computed. The middle slice was extracted for visualization (no interpolation required since the voxels are aligned to the pixels of that particular input frame). The reconstruction parameters used for this is summarized in Table 3.4.

This is a classical evaluation technique, and the idea is that a good reconstruction algorithm should be good at approximating data that's missing. When a slice is removed, the gold standard is known at the position of that slice, making it possible to compute the error. We choose to also see what happens with zero slices removed – how the algorithms perform when the data is actually in place is also interesting, as it may tell us something about smoothing or noise removal.

Parameter	**Value**
Input pixel spacing	0.064
Output voxel spacing	0.064
Volume dimensions	638x638x328
Radius (All methods except PNN)	3.0
Distance in voxels (PNN Only)	3
Number of input slices	85
G_{max}	1
σ_{max} (VGDW only)	32.0
σ_{min} (VGDW only)	$3.2 \cdot 10^{-6}$
K (VGDW only)	32.0
B (VGDW only)	0.0
L (VGDW only)	0.0

Table 3.4: Parameters for slice skipping evaluation.

3.2.5 Visual evaluation on real data

The data set “Tumor” was used for this evaluation. Slices from all three axes were presented to 3 technologists and 3 medical practitioners working with ultrasound, and they were asked to rank the reconstructions according to diagnostic value. The participants were blind to the method used to reconstruct any given slice. The reconstruction parameters are given in Table 3.5.

This evaluation technique may be the most important one, especially following the definition of the reconstruction problem in Section 1. In many cases[1], humans are the ones who will be looking at the reconstructions and making decisions based on them. The idea of this evaluation is therefore to evaluate exactly that – good reconstructions make it easy for practitioners to see what is in the volume.

[1]If one wants to perform some automated segmentation algorithm on the volume, a computer algorithm will appreciate other aspects than humans will.

Parameter	**Value**
Input pixel spacing	0.089
Output voxel spacing	0.123
Volume dimensions	532x429x584
Radius (All methods except PNN)	1.0
Distance in voxels (PNN Only)	3
Number of input slices	932
G_{max}	8
σ_{max} (VGDW only)	32.0
σ_{min} (VGDW only)	$3.2 \cdot 10^{-6}$
K (VGDW only)	32.0
B (VGDW only)	0 and 5(indicated in results)
L (VGDW only)	0 and 5(indicated in results)

Table 3.5: Parameters for visual evaluation.

3.2.6 Performance evaluation

Performance evaluation was performed for the data sets “Phantom”, “Angio”, and “Tumor”. Two different volume sizes were also used – 128M voxels and 32M voxels. For each data set and each method and each volume size, the following steps were taken:

1. Reboot the computer
2. Start CustusX with an empty patient
3. Reconstruct the volume using the given method
4. Record the “Reconstruction core time” reported by CustusX – i.e. the time spent performing the reconstruction algorithm, excluding any preprocessing.
5. Delete the reconstructed volume
6. Perform steps 2-5 a total of 10 times per data set per method.

For each of the 10 iterations, the lowest value was used. The specification of the computer used for performance evaluation is listed in Table 3.6.

Model	HP EliteBook 8460p
CPU	Intel(R) Core(TM) i7-2620M CPU @ 2.70GHz
Memory	4 GiB DDR3 1333 MHz
GPU	AMD Radeon 6470M w/1GiB DDR3
O/S	Ubuntu Linux 13.04 64-bit, Linux version 3.13.0-rc2
GPU Driver version	AMD Catalyst 13.11-beta-v9.4
OpenCL SDK version	AMD APP SDK v2.9

Table 3.6: Specifications of computer used for performance evaluation

The radius was set to 1.0 for all the voxel-based methods, and the maximum hole-filling distance was set to 3 voxels for the PNN algorithm. The PNN implementation was the existing single-threaded CPU implementation in CustusX. G_{max} was set to 1 for the two data sets "Phantom" and "Angio", and 8 for "Tumor".

This evaluation technique is related to the second aspect of the definition in Section 1 – that the reconstruction should not take too long to generate. However, the results of this evaluation should be seen in light of the results of the visual evaluation – a slower algorithm may be worth waiting for if the image quality is significantly better.

Chapter 4

Results

In this chapter we will present the results of the 4 image quality evaluations and the performance evaluation. The results will be presented with numerical values and graphs for RMS scores, as well as selected, aligned slices from the different reconstructions side-by-side.

The performance evaluation will be presented as time consumed in seconds, both numerically and graphically.

Before we present the results, we have to clarify what the X, Y and Z axis refers to in the image slices. The slices will be labeled as coming from the X, Y and Z axis. What we mean by this, is that the dumped slice number N in the X direction is the N-th slice in the X direction, in other words the YZ plane at X=N. This notation stems from the X, Y and Z axes in volume space of the volume they were extracted from. *CustusX* aligns the volume so that the middle slice of the input data is aligned to the volume. For a perfect, linear scan, this means that the ultrasound slice will fall perfectly into the XY plane. For this reason, we will typically see much higher quality in the Z direction, since they are (approximately) aligned with the input data.

Without further ado, we present the results.

4.1 Reconstruction from existing volume

In this section we will present the results of the evaluation technique "Reconstruction from existing volume", described in Section 3.2.2.

4.1.1 Reconstruction from existing ultrasound volume

We will now present the results of the "reconstruction from existing volume" with the data set "Simulated from ultrasound".

Results

Method	#P	RMS Error	% of lowest
PNN	N/A	28.10	100.00%
VNN	N/A	28.78	102.43%
VNN2	4	28.33	100.81%
VNN2	8	28.15	100.16%
DW	4	28.30	100.70%
DW	8	28.12	100.06%
VGDW	4	28.18	100.28%
VGDW	8	28.22	100.44%

Table 4.1: Results from the reconstruction from existing ultrasound volume evaluation described in Section 3.2.2

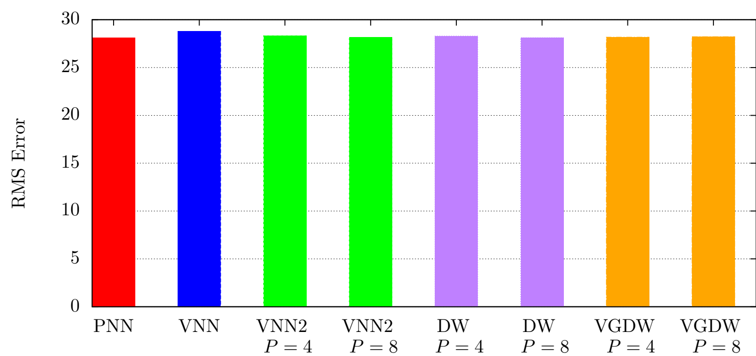


Figure 4.1: The RMS Errors for the reconstruction from existing ultrasound volume evaluation

Images

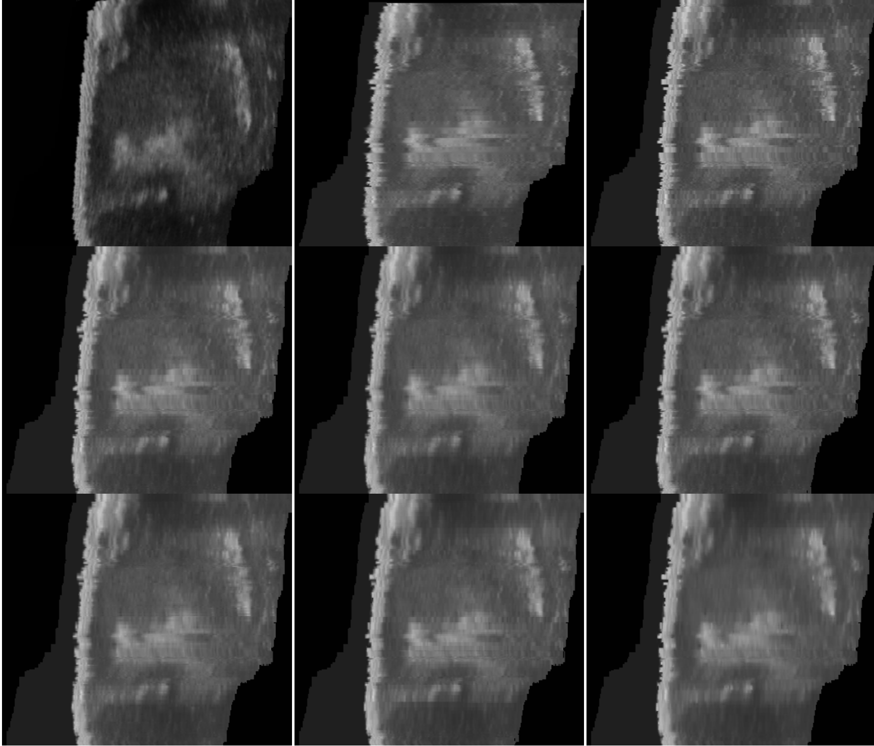


Figure 4.2: Images from the reconstruction from existing ultrasound volume evaluation in the X direction
Top row: Original US, PNN, VNN
Middle row: VNN2 with 4 planes, VNN2 with 8 planes, DW with 4 planes
Bottom row: DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

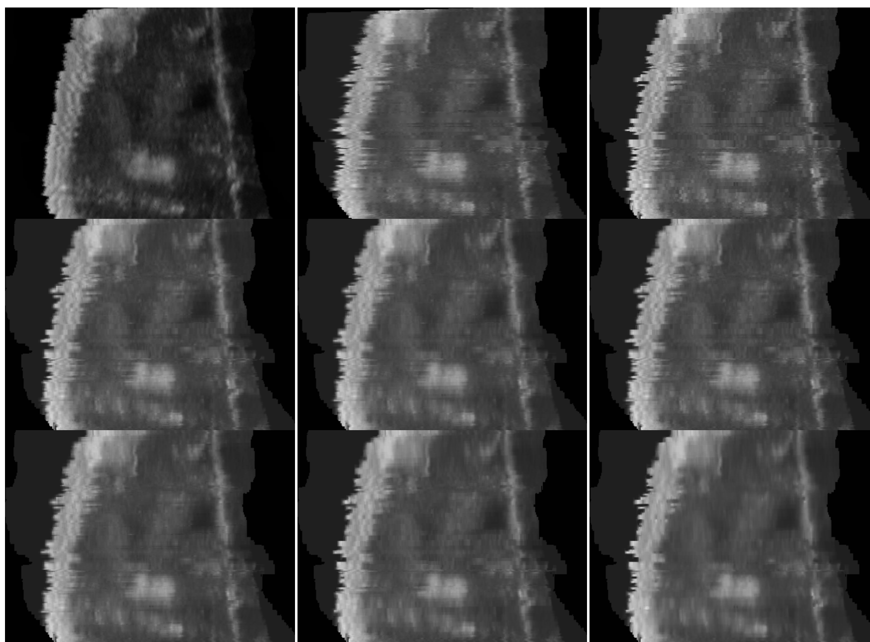


Figure 4.3: Images from the reconstruction from existing ultrasound volume evaluation in the Y direction
Top row: Original US, PNN, VNN
Middle row: VNN2 with 4 planes, VNN2 with 8 planes, DW with 4 planes
Bottom row: DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

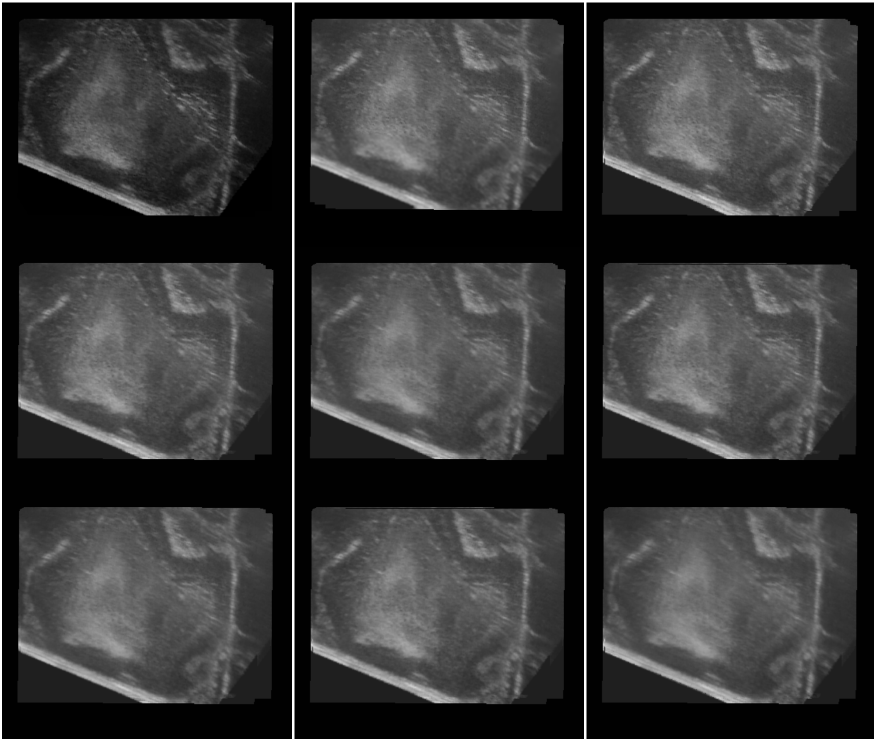


Figure 4.4: Images from the reconstruction from existing ultrasound volume evaluation in the Z direction
Top row: Original US, PNN, VNN
Middle row: VNN2 with 4 planes, VNN2 with 8 planes, DW with 4 planes
Bottom row: DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

4.1.2 Reconstruction from existing MRI volume

We will now present the results of the “reconstruction from existing volume” evaluation with the data set “Simulated from MRI”.

Results

Method	#P	RMS Error	% of lowest
PNN	N/A	9.05	114%
VNN	N/A	9.26	116%
VNN2	4	8.64	108%
VNN2	8	8.19	103%
DW	4	8.63	108%
DW	8	8.17	103%
VGDW	4	8.35	105%
VGDW	8	7.97	100%

Table 4.2: Results from the reconstruction from existing MRI volume evaluation test described in Section 3.2.2

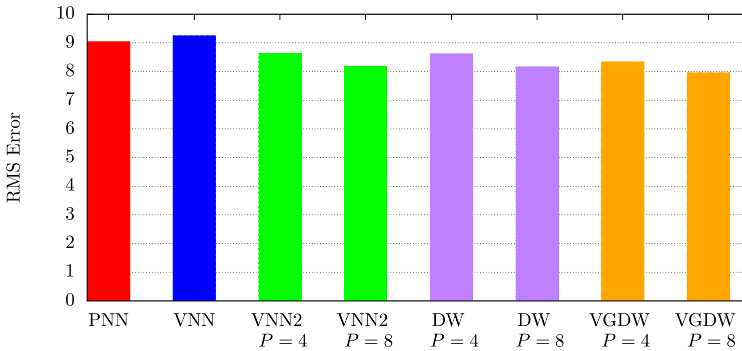


Figure 4.5: The RMS Errors for the reconstruction from existing MRI volume evaluation

Images

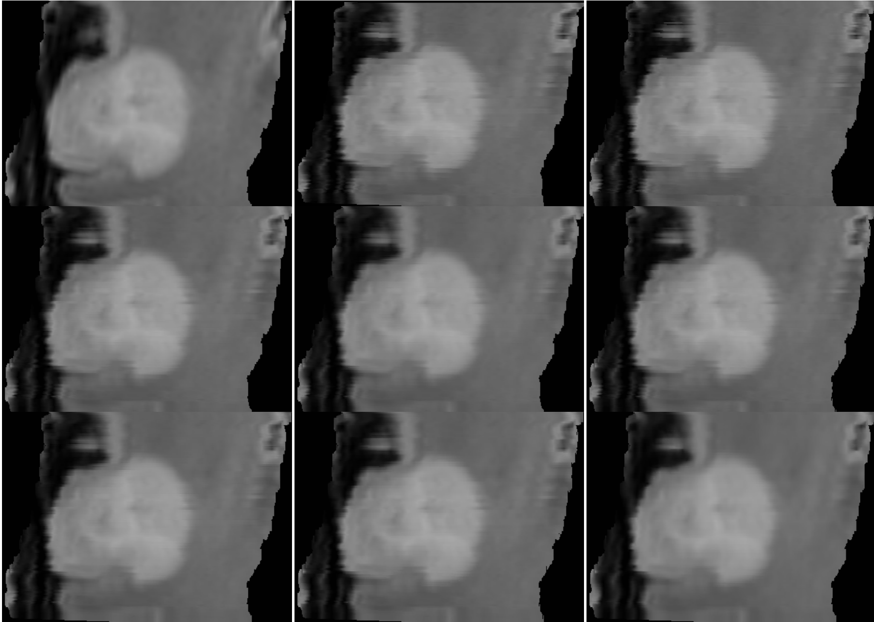


Figure 4.6: Images from the Reconstruction from existing MRI volume evaluation in the X direction
Top row: Original MRI, PNN, VNN
Middle row: VNN2 with 4 planes, VNN2 with 8 planes, DW with 4 planes
Bottom row: DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

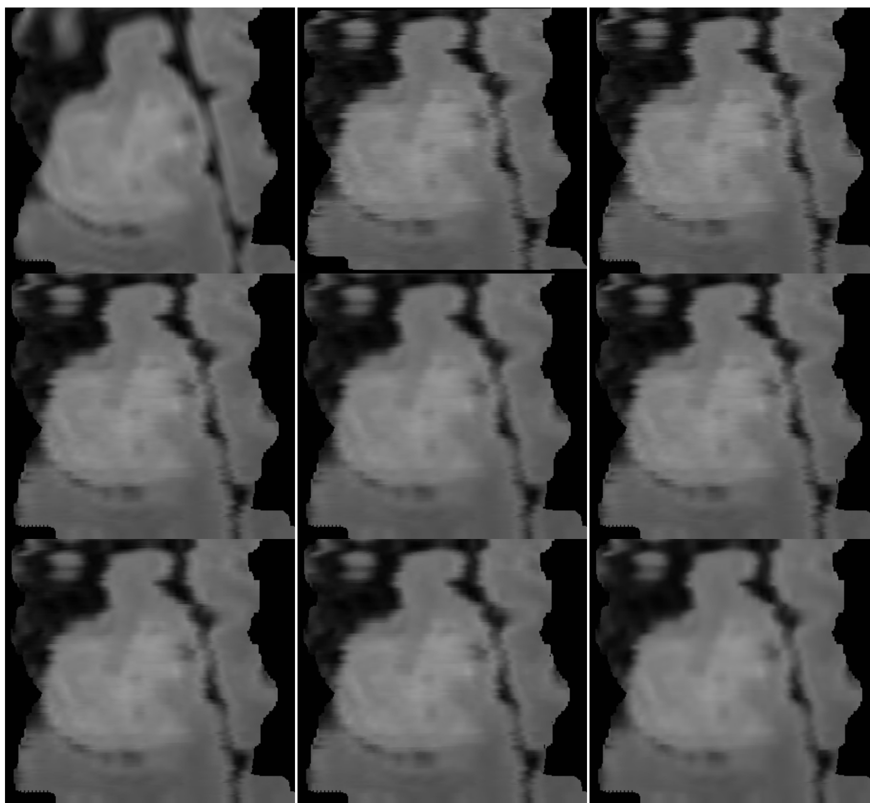


Figure 4.7: Images from the reconstruction from existing MRI volume evaluation in the Y direction
Top row: Original MRI, PNN, VNN
Middle row: VNN2 with 4 planes, VNN2 with 8 planes, DW with 4 planes
Bottom row: DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

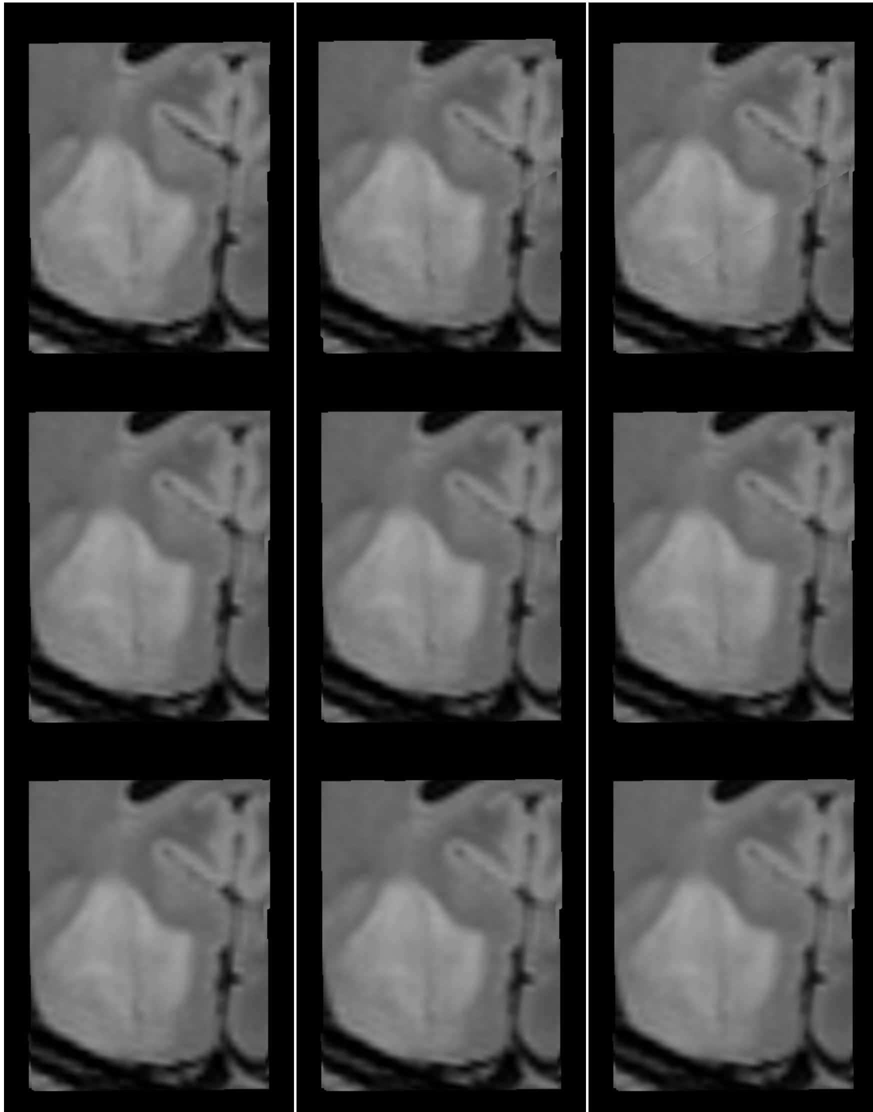


Figure 4.8: Images from the reconstruction from existing MRI volume evaluation in the Z direction
Top row: Original MR, PNN, VNN
Middle row: VNN2 with 4 planes, VNN2 with 8 planes, DW with 4 planes
Bottom row: DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

4.2 Synthetic volume

We will now present the results of the synthetic volume evaluation, described in Section 3.2.3.

4.2.1 Results

Method	# P	RMS Error			% of lowest		
		Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
VNN	N/A	8.45	13.18	9.69	106%	114%	115%
VNN2	4	8.61	11.60	9.39	108%	100%	112 %
VNN2	8	9.04	11.68	9.72	114%	101%	116%
DW	4	8.61	11.58	9.39	108%	100%	112%
DW	8	9.04	11.66	9.72	114%	101%	116%
VGDW	4	7.96	11.72	8.40	100%	101%	100%
VGDW	8	8.23	12.36	8.49	103%	107%	101%

Table 4.3: Results from the Synthetic volume test described in Section 3.2.3

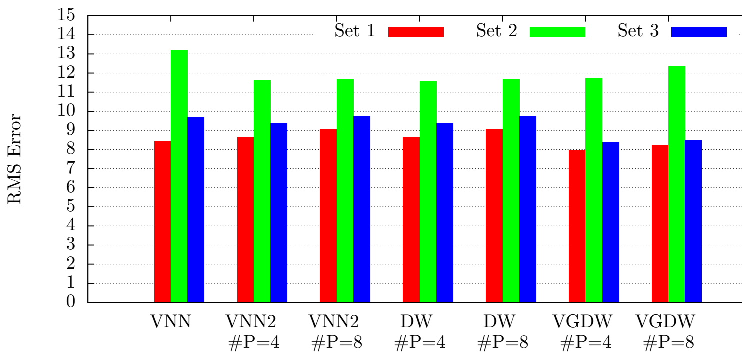


Figure 4.9: The RMS Errors for the Synthetic volume test

4.2.2 Images

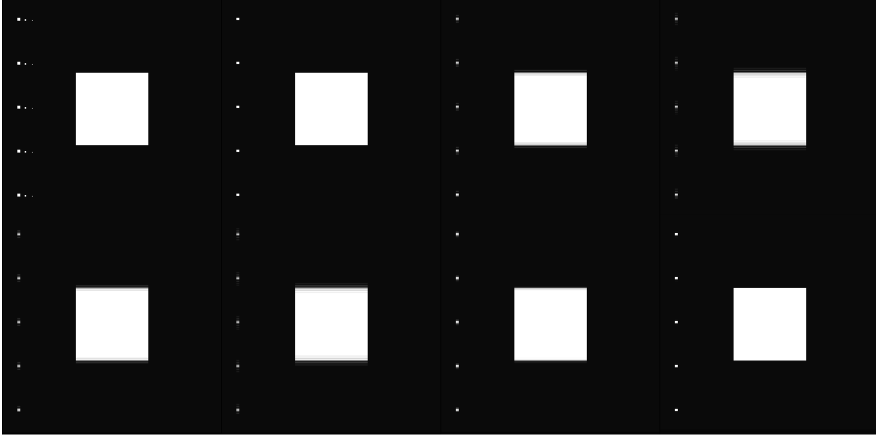


Figure 4.10: Images from set 1 of the Synthetic Volume test in the X direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

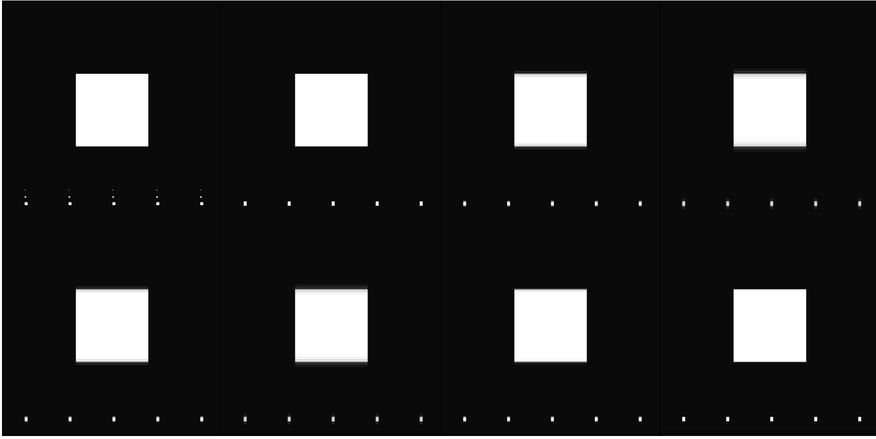


Figure 4.11: Images from set 1 of the Synthetic Volume test in the Y direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

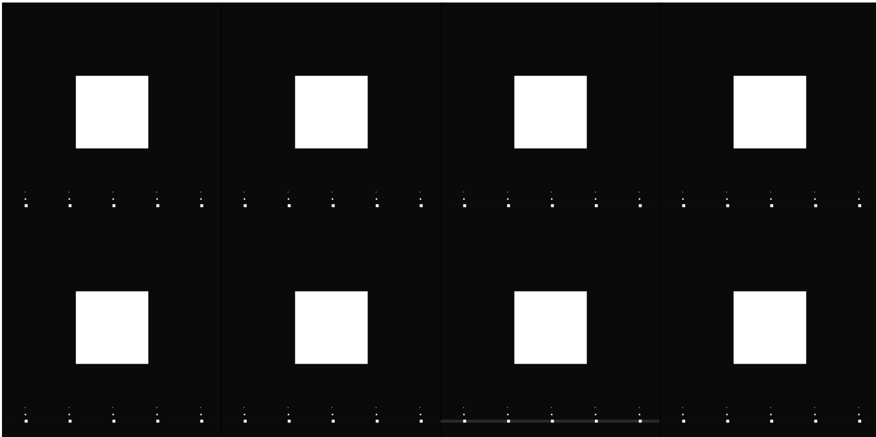


Figure 4.12: Images from set 1 of the Synthetic Volume test in the Z direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

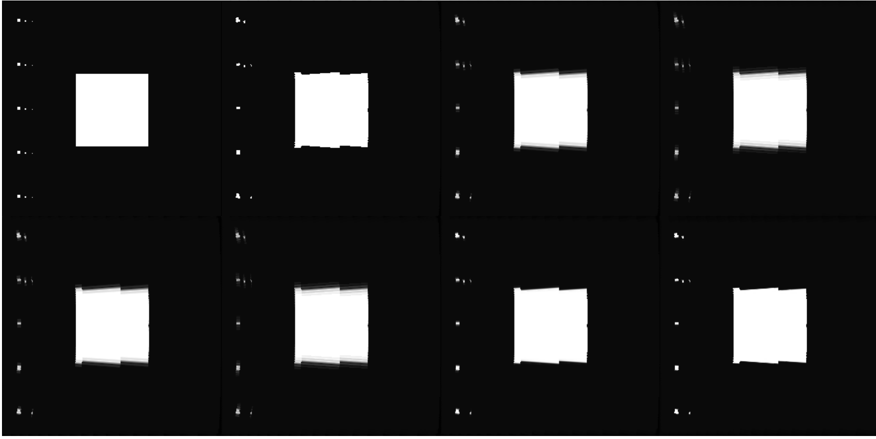


Figure 4.13: Images from set 2 of the Synthetic Volume test in the X direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

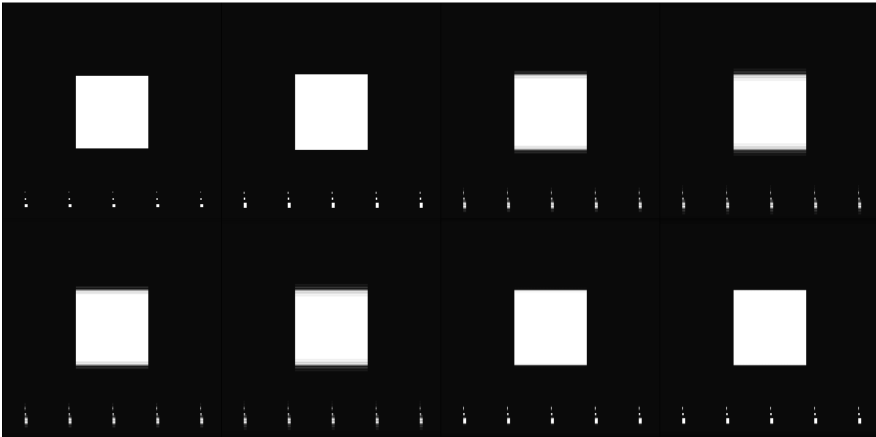


Figure 4.14: Images from set 2 of the Synthetic Volume test in the Y direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

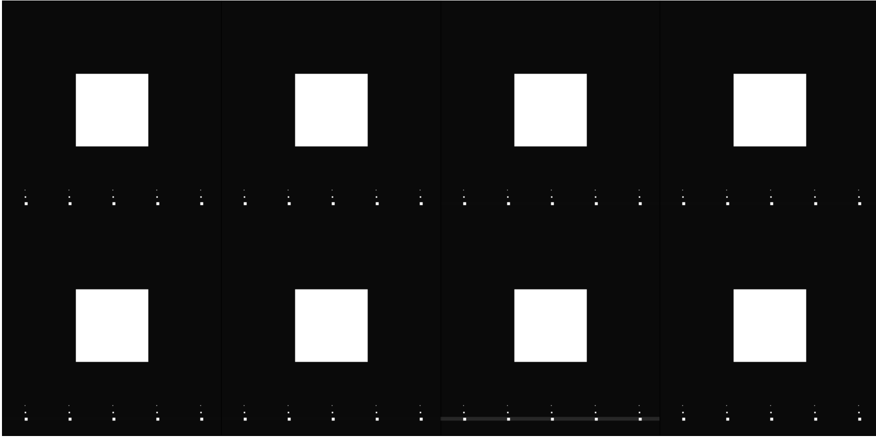


Figure 4.15: Images from set 2 of the Synthetic Volume test in the Z direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

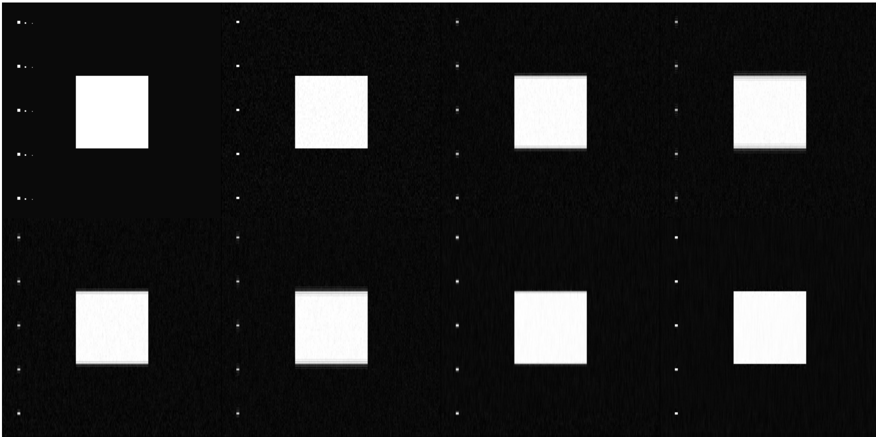


Figure 4.16: Images from set 3 of the Synthetic Volume test in the X direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

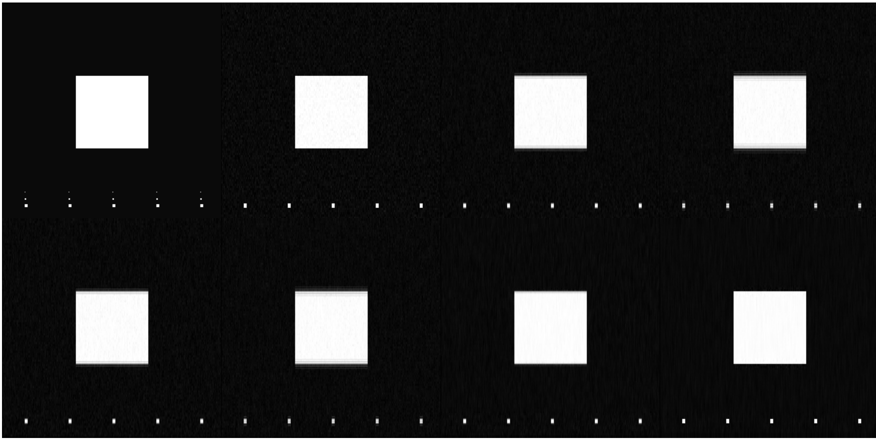


Figure 4.17: Images from set 3 of the Synthetic Volume test in the Y direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

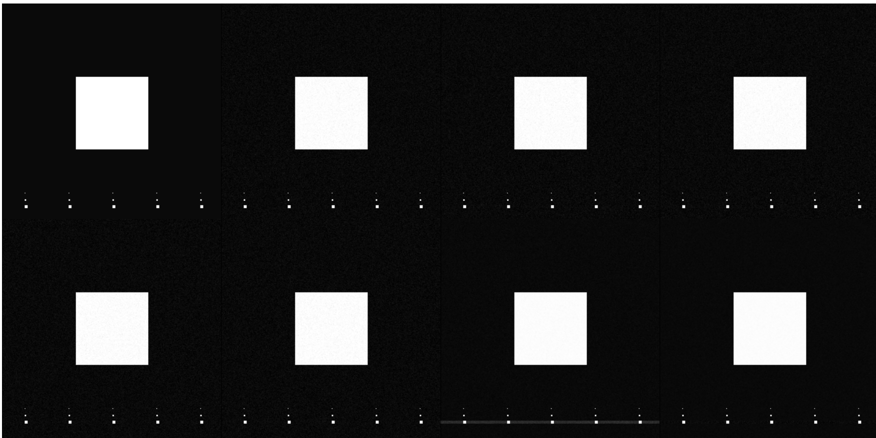


Figure 4.18: Images from set 3 of the Synthetic Volume test in the Z direction
Top row: Gold standard, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

4.3 Slice removal

We will now present the results of the slice removal evaluation, described in Section 3.2.4.

4.3.1 Results

Method	#P	RMS Error				% of lowest			
		Skip 0	Skip 1	Skip 3	Skip 5	Skip 0	Skip 1	Skip 3	Skip 5
VNN	N/A	0.19	7.60	7.14	8.05	100%	137%	124%	114%
VNN2	4	1.48	6.19	5.77	7.76	797%	112%	100%	110%
VNN2	8	1.55	6.02	6.32	7.08	836%	109%	110%	100%
DW	4	1.48	6.19	5.77	7.76	797%	112%	100%	110%
DW	8	1.56	6.02	6.32	7.08	836%	109%	110%	100%
VGDW	4	3.95	5.63	5.79	7.48	2126%	102%	100%	106%
VGDW	8	3.99	5.54	6.54	7.58	2143%	100%	113%	107%

Table 4.4: Results from the Slice skip evaluation described in Section 3.2.3

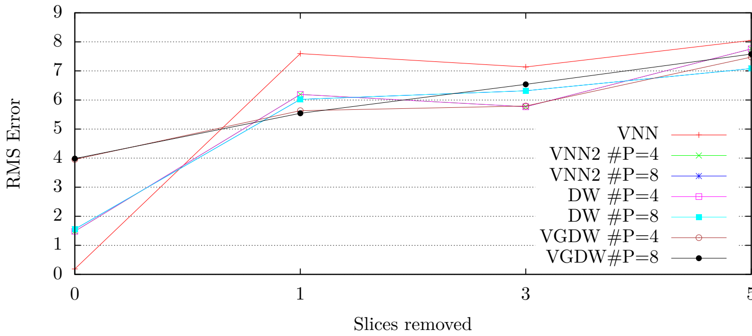


Figure 4.19: The RMS Errors for the slice removal test

4.3.2 Images

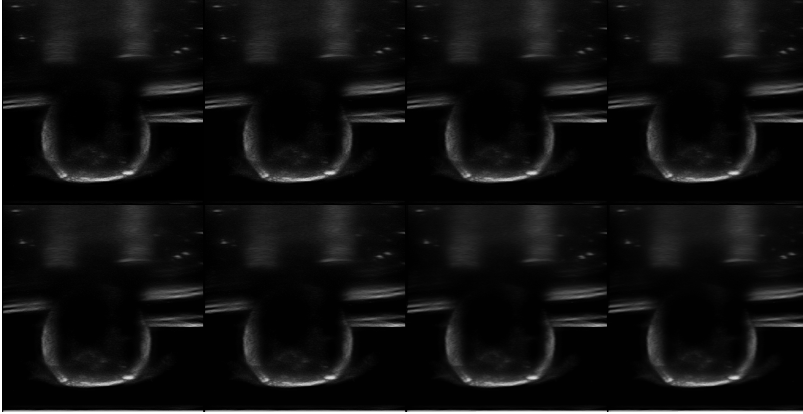


Figure 4.20: Images from the frame skip test with 0 frames skipped
Top row: Original US slice, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

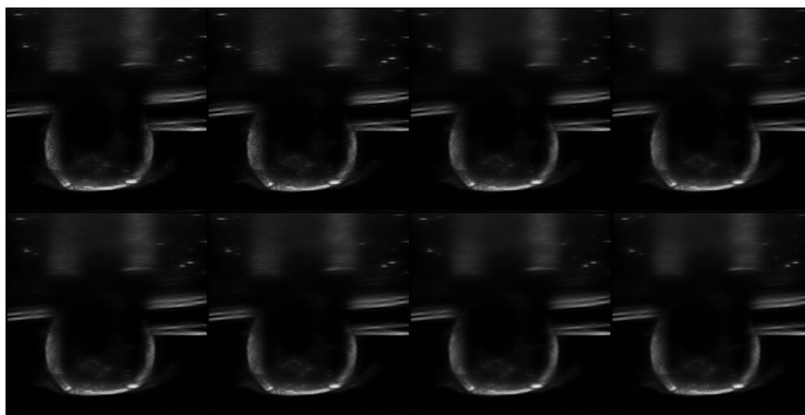


Figure 4.21: Images from the frame skip test with 1 frame skipped
Top row: Original US slice, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

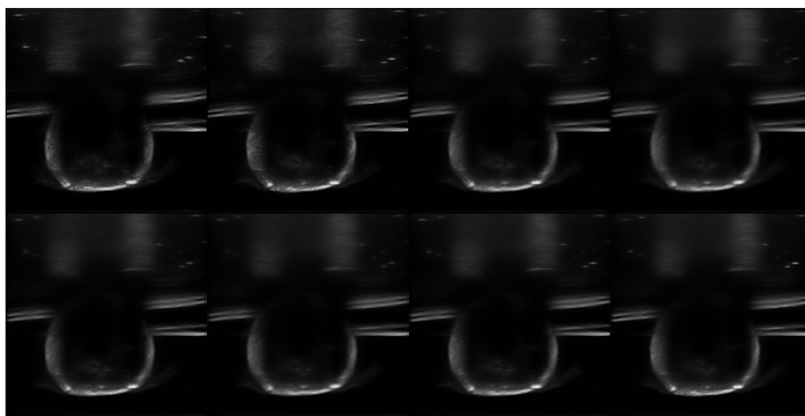


Figure 4.22: Images from the frame skip test with 3 frames skipped
Top row: Original US slice, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

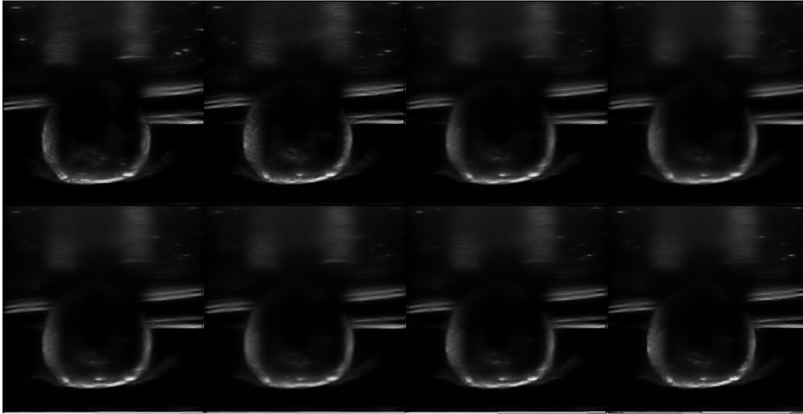


Figure 4.23: Images from the frame skip test with 5 frames skipped
Top row: Original US slice, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Bottom row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

4.4 Visual evaluation

We will now present the results of the visual evaluation, described in Section 3.2.5.

4.4.1 Results

Some notes for Table 4.5: Medical person 1 gave separate ranks for each axis, these scores have been averaged. Medical person 3 only stated that VGDW with $B = 5$ was better than the others, and did not provide further ranking. Medical person 3 was excluded from the average computation, since full ranking was not provided.

Method	**#P**	**Avg**	**T1**	**T2**	**T3**	**M1**	**M2**	**M3**
PNN	N/A	10.93	12	11	11	8.67	12	X
VNN	N/A	8.33	3	10	10	9.67	9	X
VNN2	4	6.6	5	7	9	6	6	X
VNN2	8	5.8	9	6	4	6	4	X
DW	4	5.67	4	9	5	7.33	3	X
DW	8	7.4	8	5	8	8	8	X
VGDW	4	6	6	8	3	6	7	X
VGDW	8	5.53	9	3	7	3.67	5	X
VGDW $L = 5$	4	11.53	11	12	12	11.67	11	X
VGDW $L = 5$	8	7.53	10	4	6	7.67	10	X
VGDW $B = 5$	4	1.17	1	1	2	1	1	1
VGDW $B = 5$	8	1.67	2	2	1	2	2	1

Table 4.5: Results from visual evaluation. “T” refers to technologist, “M” refers to medical personnel. Lower is better.

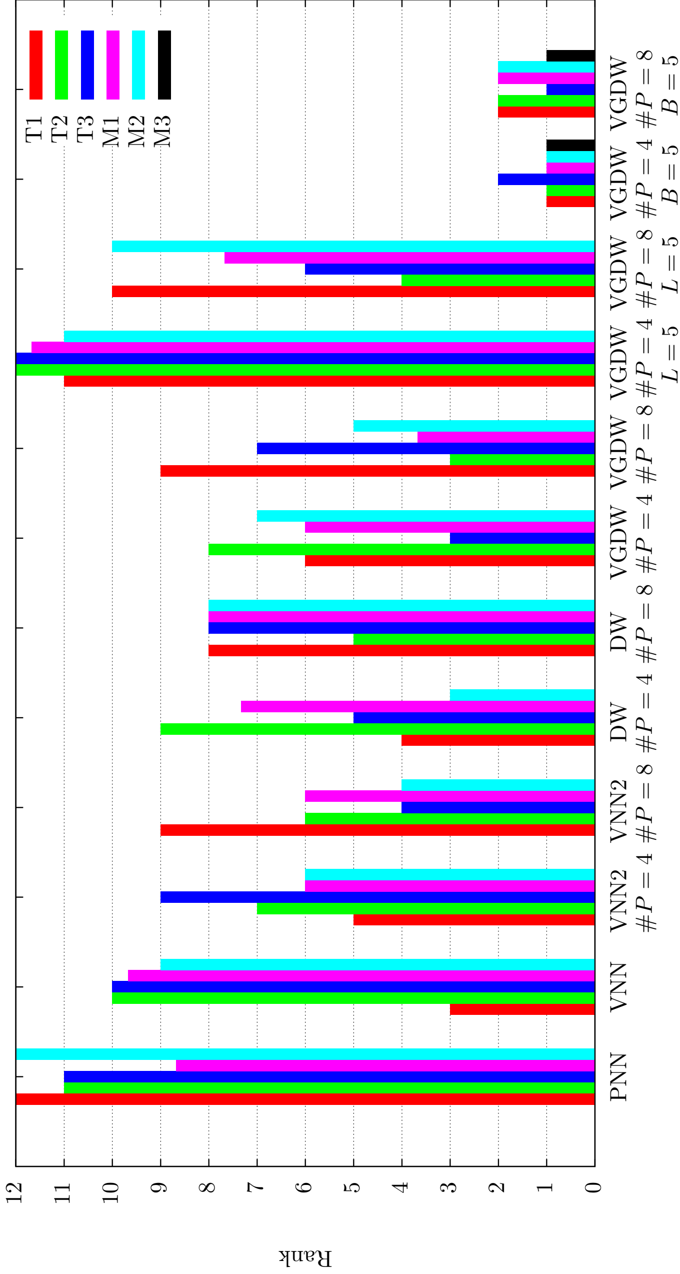


Figure 4.24: Results of the visual evaluation. Y-axis is rank, lower is better

4.4.2 Images

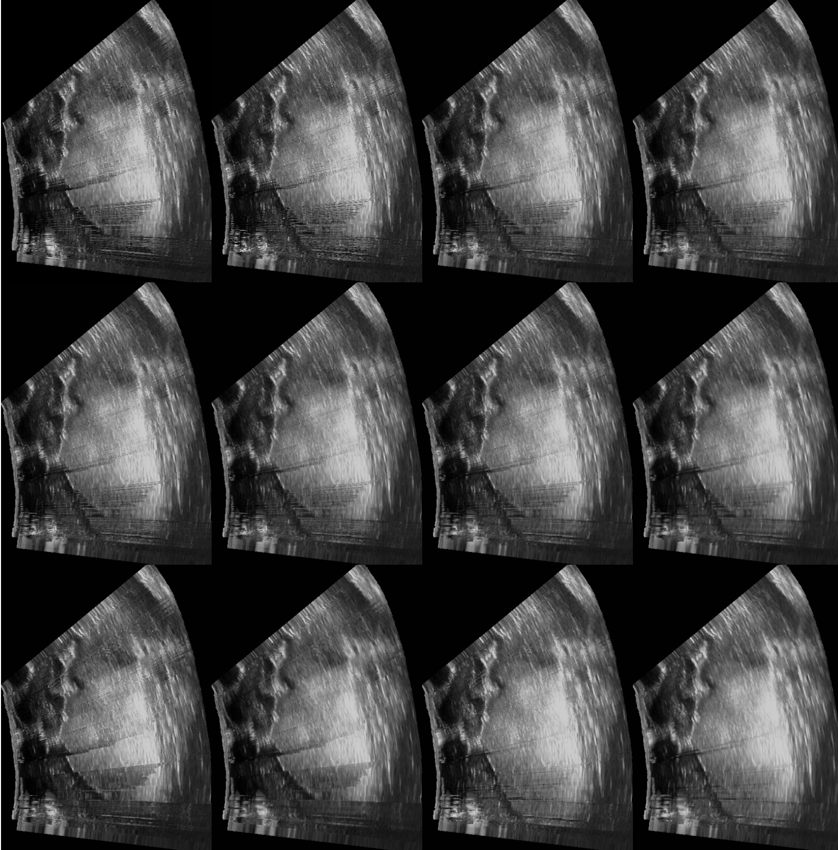


Figure 4.25: Slices from the reconstructed volumes generated from the “Tumor” data set. Slice 200 in X direction
Top row: PNN, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Middle row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes
Bottom row: VGDW with $W_{new} = 5$ with 4 planes, VGDW with $W_{new} = 5$ with 8 planes, VGDW with $W_{bright} = 5$ with 4 planes, VGDW with $W_{bright} = 5$ with 8 planes

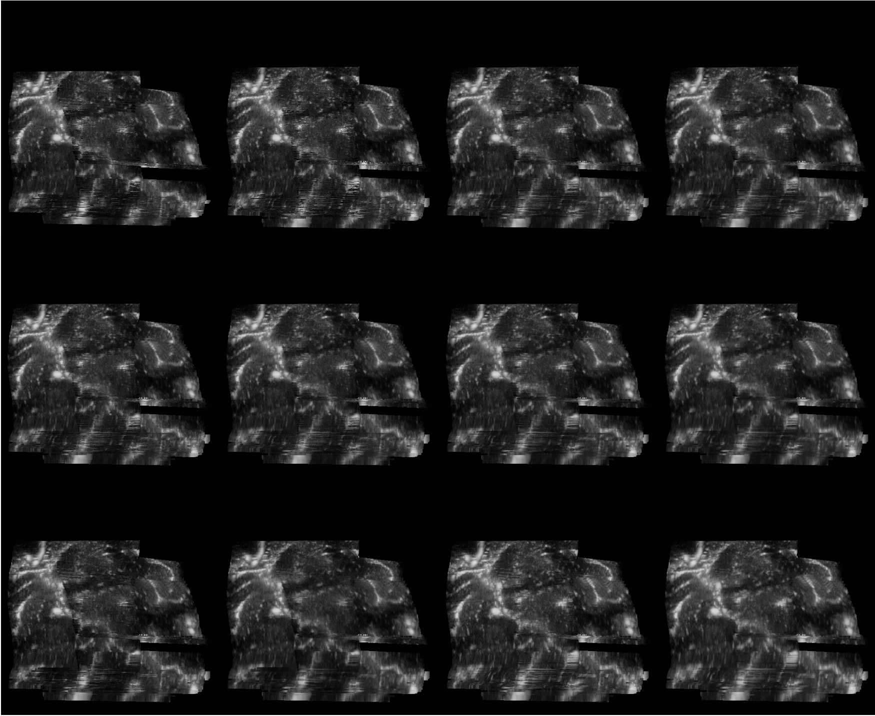


Figure 4.26: Slices from the reconstructed volumes generated from the "Tumor" data set. Slice 100 in Y direction
Top row: PNN, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Middle row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes
Bottom row: VGDW with $W_{new} = 5$ with 4 planes, VGDW with $W_{new} = 5$ with 8 planes, VGDW with $W_{bright} = 5$ with 4 planes, VGDW with $W_{bright} = 5$ with 8 planes

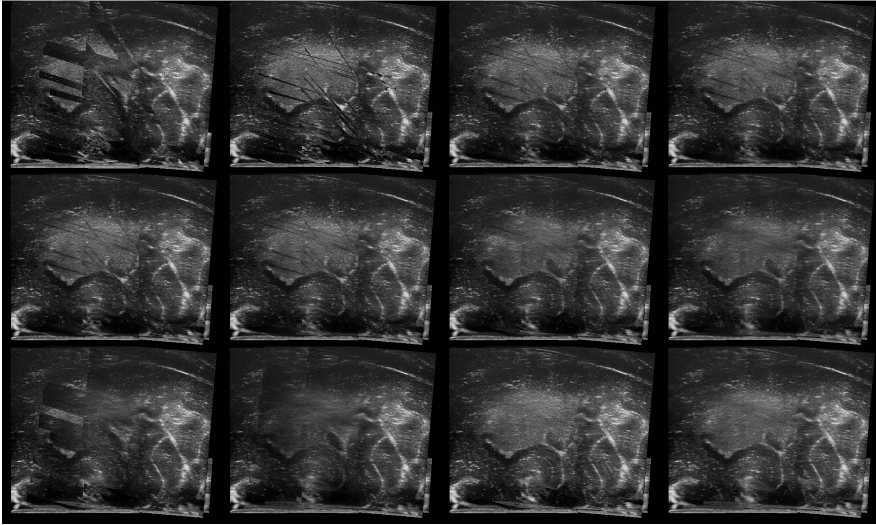


Figure 4.27: Slices from the reconstructed volumes generated from the "Tumor" data set. Slice 100 in Z direction
Top row: PNN, VNN, VNN2 with 4 planes, VNN2 with 8 planes
Middle row: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes
Bottom row: VGDW with $W_{new} = 5$ with 4 planes, VGDW with $W_{new} = 5$ with 8 planes, VGDW with $W_{bright} = 5$ with 4 planes, VGDW with $W_{bright} = 5$ with 8 planes

4.5 Performance

4.5.1 32M Volume

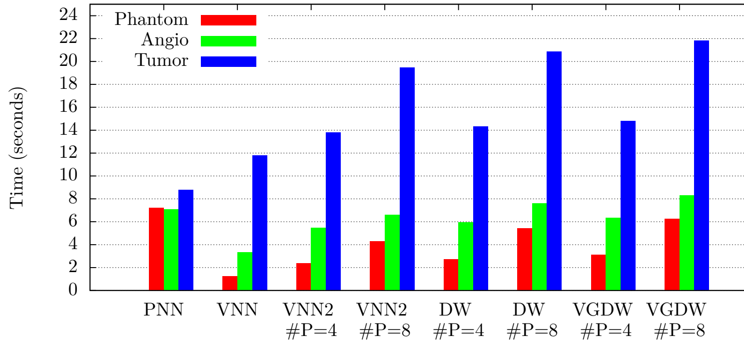


Figure 4.28: The results of the 32M performance evaluation

Method	#P	Time taken (seconds)			% of smallest		
		Phantom	Angio	Tumor	Phantom	Angio	Tumor
PNN		7.22	7.07	8.79	578%	214%	100%
VNN		1.25	3.31	11.77	100%	100%	134%
VNN2	4	2.36	5.46	13.78	189%	165%	157%
VNN2	8	4.28	6.60	19.46	342%	199%	221%
DW	4	2.73	5.95	14.30	218%	180%	163%
DW	8	5.43	7.62	20.85	434%	230%	237%
VGDW	4	3.10	6.33	14.81	248%	191%	168%
VGDW	8	6.23	8.29	21.80	498%	250%	248%

Table 4.6: The results of the 32M performance evaluation

4.5.2 128M Volume

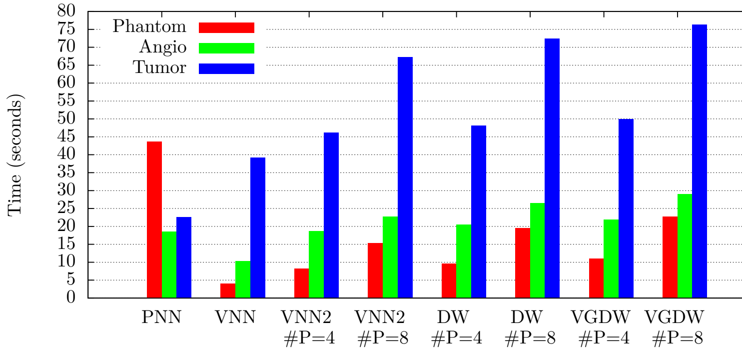


Figure 4.29: The results of the 128M performance evaluation

Method	#P	Time taken (seconds)			% of smallest		
		Phantom	Angio	Tumor	Phantom	Angio	Tumor
PNN		43.57	18.47	22.52	1095%	179%	100%
VNN		3.98	10.30	39.20	100%	100%	174%
VNN2	4	8.12	18.61	46.10	204%	181%	205%
VNN2	8	15.29	22.77	67.19	384%	221%	298%
DW	4	9.53	20.45	48.03	239%	199%	213%
DW	8	19.56	26.45	72.38	491%	257%	321%
VGDW	4	11.01	21.82	49.90	277%	212%	222%
VGDW	8	22.72	28.99	76.22	571%	281%	338%

Table 4.7: The results of the 128M performance evaluation

Chapter 5

Discussion

In this chapter we will discuss the results presented in the previous chapter. We will first discuss the results of each evaluation technique separately, trying to discern what we can learn from the results. Then we will take a step back and discuss some broader topics, and try to see how to attack some of the issues the results may have revealed.

5.1 Reconstruction from existing volume

We will now discuss the results of the evaluation technique "reconstruction from existing volume".

5.1.1 Reconstruction from existing ultrasound volume

The results of the reconstruction from existing volume evaluation with the "Simulated from Ultrasound" data set are are not very conclusive – all methods are within 2.43% of each other in the RMS comparison. The "PNN" method delivers the "best" result in regards to RMS, which may be related to the fact that the original volume was reconstructed using PNN. However, the images display seen in Figure 4.2, Figure 4.3 and Figure 4.4 show less noise with VGDW, but the images also look less crisp with VGDW. In general, the 8 plane version appear more smoothed than does the 4-plane variant. This may indicate that the smoothing is too aggressive with VGDW.

5.1.2 Reconstruction from existing MRI volume

We see that VGDW with 8 planes produces the reconstructions that are closest to the original in this test, while VNN is the worst performer. The images provide some insight into why VGDW performs well on this test – since the input data is relatively smooth, VGDW benefits from being able to use a wide Gaussian without smoothing the result much further. As a result, the "liney" appearance from the other reconstructions are tougher to spot. There is also virtually no noise in the input data, which in this theoretical measurement is beneficial for VGDW, since it will try to reduce noise. It must also be noted that the original MRI volume was not aligned originally, tri-linear interpolation had to be performed in order to compute RMS values – this may have lead to elevated RMS values in general, but all methods suffered equally from this problem.

5.2 Synthetic volume

For the synthetic volume test, we see that the 4 plane variants generally give better results than the 8 plane variants. This may be related to the

nature of the details in the synthetic volume – as long as they are visible, further smoothing them by adding more planes will generally give a result further away from the gold standard. The images from this evaluation tell an interesting story. For data set 1, if we look at Figure 4.11, we see that VNN shows hard edges, since it does not perform any smoothing when selecting what pixel value goes into what voxel. The VNN2 and DW have artifacts on the sides of the details – the "lines" over and under the big center block is a good example. Why these lines appear as lines and not a completely smooth region is currently unknown – an educated guess is that the inverse distance weight function becomes very large and causes floating point errors. Finally, we observe that the VGDW variants both have quite hard edges, and exhibit opposite behavior from the DW and VNN2 variants: with increasing number of planes, the edges become less smoothed, as opposed to DW and VNN2 where the edges become more smoothed. This may be because with more planes it is possible to demonstrate larger variance with a larger amount of planes to consider. We also observe that the 4 plane variant achieved a better RMS score than the 8 plane variant – this may be due to the fact that the lines (seen as 5 dots on the bottom of Figure 4.11) are too wide in the direction of acquisition, but are slightly smoothed when using 4 planes.

For data set 2, we clearly see the artifacts of the tilted scan pattern. Looking at Figure 4.13, we see here the trade-off of the assumption we made in Section 3.1.1: Since we only include one value from each image plane, no indication is provided to the algorithm that the side of the cube is in fact straight. As for the RMS results, DW with 4 planes performs best, and VGDW is the worst performer outside of VNN. This may be because since all the reconstructions have quite a bit of artifacts, the reconstructions that provide a bit of smoothing will generally score higher. We also observe at the edges of the images that there is a larger area that is completely black on the reconstructions with 8 planes, and especially for VGDW, contributing to an elevated RMS. Interestingly, VGDW still shows well defined edges, without the jaggedness of the VNN reconstruction and the "liney" artifacts of DW and VNN2.

For data set 3, we see the noise smoothing capabilities of VGDW clearly in Figure 4.13. With VNN, the noise is clearly visible, VNN2 and DW do smooth the noise a little bit, but it they also smooth the details of the volume, whereas VGDW smooths the noise quite drastically, but retains the well defined edges of the details of the volume. Again VGDW with 4 planes has the lowest RMS error, and VGDW in general beats the other reconstruction methods by a large margin. However, the noise added to the

slices in this test was Gaussian noise, not speckle, and the details of the volume are not very subtle - it is to be expected that VGDW will perform very well in this test.

5.3 Slice removal

We here observe that VGDW has very high RMS error even with 0 slices skipped – this may be attributable to the fact that VGDW deliberately smooths noise, as may be seen in Figure 4.20. However, it may also be attributable to over-smoothing and slight misalignment. VNN performs best, as is to be expected – theoretically the slice should be identical when using VNN. However, when 1 slice is removed, VGDW performs best, which is quite a feat considering it already has quite a bit of error due to noise smoothing. When 3 planes is removed, it is neck in neck with DW and VNN2. We observe in Figure 4.22 that the DW and VNN2 reconstructions are blurrier with less clearly defined edges around the rod. We also observe that VGDW seems to experience large variance in at the edges of the rod, and seems to have a very steep weighting function in that region, leading to much sharper edges than DW and VNN2. With 5 slices removed, DW and VNN2 are clearly better than VGDW in terms of RMS. Interestingly, VGDW with 4 planes generally perform better than the variant with 8 planes in this test. Again, this may be indicative of either successful noise reduction, or oversmoothing.

5.4 Visual evaluation

In this evaluation, we used the Tumor data set which exhibits U-turn scanning, and thus overlapping data. This can be seen by looking at Figure 4.25 – the PNN and VNN reconstructions both have very visible lines due to overlapping data having very different intensities. VNN2 and DW smooths this out somewhat, but it is still clearly visible. VGDW with the standard weight function smooths it out even further, but it is still clearly visible, albeit not as disruptive. The reconstructions with $W_{new} = 5$, which gives higher weight to the latest frames, gets rid of the lines in their original sense, but in this particular case the most recent data is also the darkest data, making it clearly visible where the "new" data is located. Finally, for the reconstructions with $W_{brightness} = 5$, the tumor can be seen relatively

unobstructed. It is difficult to try to answer the question of which weight strategy is "best" – in this case the $W_{brightness} = 5$ reconstructions provide the best visual quality, but this is situation-dependent. If a practitioner decided to do an extra sweep over a region, he might have had a reason to do so, and may then want to give extra weight to more recent samples in the reconstruction.

For the Y direction, seen in Figure 4.26, the same reasoning may be applied. Finally, in the Z-direction as seen in Figure 4.27 the effects of the different reconstruction strategies can be seen quite dramatically, with particularly PNN and VNN giving very poor results, and the VGDW reconstructions giving very good results with little obstructive artifacts.

The results from the visual evaluation performed by ultrasound technologists and medical personnel can be seen in Table 4.5. The most striking result is that the reconstructions generated by VGDW with extra weight to bright values are unanimously preferred over the other reconstructions. We also observe that VGDW with extra weight to more recent images are generally given a poor rank. This may be explained by the fact that data in the end of the "Tumor" data set was darker than the data in the beginning, leading this weight function to give extra weight to data that in this case was undesirable. This does, however, suggest that giving extra weight to bright data is a more "universally good" approach to the problem of intersecting/conflicting data, since there are fewer conditions in which it will yield poor results. We also observe that on average, there is a very large "middle field" ranging from 5.53 to 7.17, which contains VGDW with no extra weight, VGDW with 8 planes and extra weight to late planes, DW and VNN2. This tells us that subjective the reconstruction qualities of these algorithms are very similar, and it is hard to claim that one is "better" than the other. However, PNN and VGDW with L=5 and 4 planes are clear "losers" of this evaluation.

In regards to the use of 4 planes versus 8 planes, we observe that in the case of VGDW with $B = 5$, the 4-plane version is generally preferred. From this we can possibly deduce two things:

1. Spending the extra time computing the 8-plane variant is not worth it, and/or
2. VGDW smooths too aggressively, causing more data to yield poorer results.

The last point can be tied directly to the behavior of 3 of the 5 parameters mentioned in Section 3.1.3, which we discuss in Section 5.6.2.

5.5 Performance

For the 32M reconstructions, we see that the voxel-based methods in most cases are faster than PNN, with the exception of the Tumor data set. This may partly be explained by the large amount of image planes in the Tumor data set, as well as the fact that the plane search algorithm will need to keep more guesses, and thus perform more plane searches for the Tumor data set than for the others, which are simple linear scans. The PNN algorithm appears relatively robust with respect to how many image planes the data set contains.

For the 128M reconstructions, we see that PNN performs very poorly for the Phantom data set, probably due to having to do a lot of hole-filling as there is not much data available. The voxel-based methods generally perform very well for the Phantom data set, but for the Tumor data set they have rather poor performance, bordering unusable in a clinical setting.

A very interesting thing to note is that among the voxel-based methods, the most important determining factor in computational time seems to be the number of image planes to include. VNN includes only a single plane, and the others include planes as specified (4 or 8). This means that the overhead of the much more complex weight function in VGDW does not contribute significantly to computational time, at least not on the GPU used in the evaluation.. An explanation for this is that the algorithms are memory-bandwidth starved – they spend a lot of time waiting for data from the global memory. Some extra computation once the data has arrived therefore does not contribute much to the computational time.

It should however be noted that these results come from a a relatively slow AMD Radeon 6470M GPU with slow DDR3 memory, and since the algorithm is already parallelized it will benefit tremendously from a faster GPU.

5.6 General discussion

In this section we will take a step back, and discuss some general aspects of the algorithm and results as a whole.

5.6.1 What is reconstruction quality?

Most of the evaluation techniques in this thesis revolve around measuring the RMS error. For an algorithm designed to suppress noise, this is not advantageous, since RMS error will also measure the noise removed. In regards to the visual evaluation, a reconstruction with no smoothing may also appear more "feature-rich" than a slightly smoother reconstruction, yet the perceived feature-richness may in fact be noise. On the flip side, excessive smoothing may blur and mask important image features, such as tumor edges. An important argument is that it is much more important to avoid hiding true positives than avoiding false positives in the context of medical imaging. A wise strategy would therefore be to "err on the side of caution", i.e. sacrifice some noise reduction to reduce the chance of hiding true positives. This argument especially carries over to the discussion of the reconstruction parameters below.

5.6.2 Parameters

VGDW has a number of parameters that deserve to be discussed in more detail. Some of them pertain to the problem of finding the closest image planes to a voxel, which is common to the other distance-weighted algorithms. Others are specific to VGDW, and will be discussed separately.

Multi-start local search as described in Section 3.1.2 has two interesting parameters:

1. G_{max} – the maximum number of guesses for Algorithm 1
2. N_{max} – the maximum number of image planes to return for Algorithm 3

Higher G_{max} has a performance penalty in the beginning of the execution of the algorithm, but will generally give less artifacts in problematic regions in the volume (e.g. if the direction of the probe movement suddenly turns around, this will be a problematic region for a local search, due to the existence of many minima with very short distance between them). Because Algorithm 1 does not store more minima than it finds, the performance hit is a "one-time" hit per cube. If reason is found to increase G_{max}, one may thus also consider increasing the cube size to reduce the relative performance impact.

Higher N_{max} gives, as we see in Table 4.6 and Table 4.7, dramatic increase in computational time, the time consumed increasing approximately proportionally to the increase in N_{max}. Ludvigsen [Ludvigsen, 2010] concluded in his master thesis that the increase in computational speed was not worth it in terms of image quality for the methods he evaluated, and we reiterate this conclusion here based on the fact that using 8 planes over 4 planes does even always constitute an improvement, in some cases even the opposite is true.

VGDW-specific parameters are mentioned in Section 3.1.3. We reiterate them here:

1. B – the extra weight to give to bright data
2. L – the extra weight to give to "late" data
3. σ_{min} – the lowest value σ is allowed to take
4. σ_{max} – the highest value σ is allowed to take
5. K – the constant which is divided by the square root of the variance of the data to obtain σ

B and L are parameters whose behavior has been evaluated in this thesis. The parameters σ_{min} and σ_{max} can be used to set upper and lower bounds on how "hard" to smooth, and finally K can be used to tune the aggressiveness of the smoothing.

The three parameters tuning the Gaussian have not been evaluated in this thesis. However, as we saw in Section 5.4, the parameters chosen for the visual evaluation(listed in Table 3.5) may have been too aggressive, i.e. the algorithm may be smoothing too much.

Another thing to be on the lookout for is that while using B gives good results in terms of the visual evaluation, it may also cause bright structures to appear slightly wider than they are, since the extra weight given will cause bright data to have higher weight than dark data, also in the regions where the bright structure ends.

5.6.3 GPU considerations

When considering a GPU implementation of reconstruction methods, voxel-based methods are what comes to mind first due to the simple parallelization

– the same set of operations is repeated for a large set of voxels. However, pixel based methods can also be implemented on the GPU with good results, as demonstrated by Ludvigsen[Ludvigsen, 2010].

The plane-search algorithm described in Section 3.1.2 has both an unpredictable memory access pattern, and quite a few branches, properties that are not desirable for optimal performance on a GPU. A pixel-based approach would completely sidestep the problem of finding the closest planes, and could potentially be both faster and more robust when facing complex scanning patterns.

We may also notice that the reconstruction time of VGDW compared to DW and VNN2 is not extremely much longer. Those 3 algorithms all require the same amount of image data per reconstructed voxel, yet VGDW performs considerably more computation to determine the voxel value. This may be an indication that the main bottleneck of the implementation is the memory interface, since extra computation does not add significantly to the reconstruction time.

5.6.4 32M vs 128M volumes

From the tables Table 4.6 and Table 4.7 we see that the reconstruction time required is roughly proportional to the volume size for the voxel-based methods. PNN behaves slightly differently with the large volume, in that case the small input data set leads to a lot of holes that need to be filled and thus very long computational time. We have included aligned slices from 128M and 32M volumes in Appendix B. Looking at them, it is very hard to tell the difference, suggesting the conclusion that the 4x longer computational time is a waste of time.

Chapter 6

Conclusions/Further Work

Ultrasound imaging is an important imaging modality, both due to its speed, portability and flexibility. In the context of 3D ultrasound, 3D reconstruction of ultrasound is especially attractive due to its low cost, flexibility (no need for big clunky 3D probes) and superior resolution. The purpose of this thesis was to investigate the possibility of improving upon the existing voxel-based distance-weighted algorithms, and implementing an optimized version of it on a GPU. While we have not been able to obtain performance results on a recent GPU, we observe that even a dated mid-range notebook GPU shows reasonable performance.

6.1 Conclusions

We have seen a novel reconstruction algorithm, VGDW, which tries to exploit the nature of features in the volume – in feature-rich regions the algorithm becomes sharp and minimizes smoothing in that region. In regions where the volume does not have features, it smooths much more aggressively to suppress speckles and tracking system inaccuracies. The algorithm also includes a scheme for dealing with voxels that have more than one image plane hitting it, allowing increased priority for bright values, or newer values (or even both). We have seen a GPU implementation of said algorithm, and compared it to the older algorithms PNN, VNN, VNN2 and DW. We have also seen a compromise between speed and correctness to address the problem of complex scanning patterns, by performing a multi-start local search.

In terms of reconstruction quality, VGDW performs well in the synthetic tests. In real-world volumes it is difficult to establish whether it is better using computational metrics, due to the fact that it tries to suppress noise. However, visual evaluation reveals that the algorithm produces a more desirable result than the other algorithms it was compared with, significantly so with the customized weight function with increased weight to bright values. The result of the visual evaluation is strikingly clear – giving extra weight to bright values gives great results in conditions with value collisions, and was unanimously preferred over the other approaches. Conversely, giving extra weight to "late" values appears to be a bad idea, however we cannot rule out the possibility that it is beneficial in some specific conditions.

We can also conclude that the increased computation time associated with using 8 planes as opposed to 4 planes is in most cases not worth it. Finally, we can conclude that spending a lot of time computing high-resolution volumes is generally a waste of time, unless there are very good reasons to compute a high-resolution volume.

6.2 Further work

In working with this thesis, some ideas have occurred to us that was out of the scope of this thesis to pursue. We will present them here as inspiration for future projects.

6.2.1 Looking at the problem from the other side: a pixel-based approach

We have seen an elaborate multi-start local search scheme to deal with the problem of finding the image planes close enough to a voxel. This approach is still not perfect – it still depends on some loose structure on the input data, as it tries to find "valleys" in the search space defined by the distance of the planes to the voxel. However, PNN which looks at the problem from the opposite direction, "here I have an image plane, what voxels is it close to?" is a much simpler problem to solve. One can imagine a two-step algorithm where the first step iterates over the image planes and assigns them to voxels, and then computing the statistics and weighted sums in a second step. The first step may require some degree of synchronization, but it is absolutely worth it to investigate, especially considering the slow performance of the voxel-based methods on the "Tumor" data set, which contained a complex scanning pattern.

6.2.2 The weight functions

One of the subtle beauties of the use of a Gaussian weight function as a baseline is its flexibility – it has a very well-defined behavior, does not take any unmanageable high values and has a lot of tweaking potential. We saw in Equation 3.4 that there is a constant K influencing σ_G. The nature of this constant should be investigated – should it be a constant, or should it take on some other dynamic value (for instance the inverse of the mean value of the samples, giving higher degree of smoothing in dark areas). Further, the extra weight functions for brightness and "lateness", Equation 3.5 and Equation 3.6 are rather primitive, and should perhaps be more dynamic as opposed to their current binary nature. In particular, L and B should perhaps not be constants, but rather some values dependent on the current condition. Furthermore, collaboration with practitioners is important in determining the weight functions, as no-one knows better what they want to see than the practitioners themselves. Formulating weight functions constitutes a straightforward way to emphasize some things and suppress others, and should absolutely be looked into.

6.2.3 Performance on recent GPUs, APUs and even FPGAs

The performance results in this thesis are not very telling, as they were obtained on a dated notebook GPU, as opposed to a recent, powerful desktop GPU which will no doubt be a more common choice in a specialized system to perform 3D Freehand Ultrasound acquisitions. Especially the memory bandwidth of the GPU is interesting, as the reconstruction process is quite memory intensive. At the moment of writing, AMD has recently announced its Kaveri APU(Accellerated Processing Unit)[AMD, 2014], which can be described as a highly integrated CPU and GPU on the same die, and importantly sharing the same memory space – leading to no need for copies between the CPU and GPU. The performance of the algorithm (and other 3D Reconstruction Algorithms for that matter) on such a system would be interesting. Finally, it would also be interesting to investigate the performance on modern embedded GPU-s, such as ARM Mali, Qualcomm Adreno or Imagination PowerVR series, or even FPGAs. Acceptable performance on such devices may pave the way into the realm of affordable hand-held, battery-powered equipment.

6.2.4 Applying Probe Trajectory estimation

The Probe Trajectory technique[Coupé et al., 2005] has shown interesting results, and it may be interesting to see what applying this technique to our algorithm may achieve, especially in fan-based acquisition.

6.2.5 Investigating the effect of noise from the tracking system

Noise in the tracking data may cause the image slices to be slightly mispositioned, which may be the cause of the jagged lines visible in many reconstructions. Treating the tracking system as a “black box” delivering accurate positions may not be the best approach – modeling its noise and attempting to compensate for it, combined with Probe Trajectory estimation may further improve reconstruction quality.

6.3 Final thoughts

The work with thesis has been exploratory and experimental in nature, and lies at the border between rigorous Computer Science and more human considerations – the question "What is good image quality?" is a question that in this context is very relevant. That is why we chose to include the element of the practitioner in the definition of the problem, and it is also why we have left it to the actual practitioners to be the judge of the reconstruction quality. We have seen that good reconstruction quality can be achieved without very large compromises in reconstruction speed, and we have tried to explore the use of weight functions to emphasize parts of the data set. We hope this work will become as useful as it was fun and rewarding to work with.

Bibliography

[AMD, 2014] AMD (2014). AMD Revolutionizes Compute and UltraHD Entertainment with 2014 AMD A-Series Accelerated Processors. http://www.amd.com/us/press-releases/Pages/amd-revolutionizes-2014jan14.aspx. [96]

[Barry et al., 1997] Barry, C., Allott, C., John, N., Mellor, P., Arundel, P., Thomson, D., and Waterton, J. (1997). Three-dimensional freehand ultrasound: Image reconstruction and volume analysis. *Ultrasound in Medicine & Biology*, 23(8):1209 – 1224. [10, 22, 30]

[Cinquin et al., 1995] Cinquin, P., Bainville, E., Barbe, C., Bittar, E., Bouchard, V., Bricault, L., Champleboux, G., Chenin, M., Chevalier, L., Delnondedieu, Y., et al. (1995). Computer assisted medical interventions. *Engineering in Medicine and Biology Magazine, IEEE*, 14(3):254–263. [11]

[Coupé et al., 2005] Coupé, P., Hellier, P., Azzabou, N., and Barillot, C. (2005). 3D Freehand Ultrasound Reconstruction Based on Probe Trajectory. In Duncan, J. and Gerig, G., editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2005*, volume 3749 of *Lecture Notes in Computer Science*, pages 597–604. Springer Berlin Heidelberg. [10, 30, 96, 104]

[Fenster et al., 2001] Fenster, A., Downey, D. B., and Cardinal, H. N. (2001). Three-dimensional ultrasound imaging. *Physics in Medicine and Biology*, 46(5):R67. [8]

[Havlice and Taenzer, 1979] Havlice, J. and Taenzer, J. (1979). Medical ultrasonic imaging: An overview of principles and instrumentation. *Proceedings of the IEEE*, 67(4):620–641. [6]

[Hearn, 2011] Hearn, Baker, C. (2011). *Computer Graphics with OpenGL, 4th ed.* Pearson. [16]

[Huang et al., 2009] Huang, Q., Zheng, Y., Lu, M., Wang, T., and Chen, S. (2009). A new adaptive interpolation algorithm for 3d ultrasound imaging with speckle reduction and edge preservation. *Computerized Medical Imaging and Graphics*, 33(2):100 – 110. [2, 10, 42, 103]

[Huang and Zheng, 2008] Huang, Q.-H. and Zheng, Y.-P. (2008). Volume reconstruction of freehand three-dimensional ultrasound using median filters. *Ultrasonics*, 48(3):182–192. [10, 103]

[Khronos OpenCL Working Group, 2012] Khronos OpenCL Working Group (2012). The opencl specification. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. [12]

[Loupas et al., 1989] Loupas, T., McDicken, W., and Allan, P. (1989). An adaptive weighted median filter for speckle suppression in medical ultrasonic images. *Circuits and Systems, IEEE Transactions on*, 36(1):129–135. [7]

[Ludvigsen, 2010] Ludvigsen, H. (2010). Real-time GPU-based 3D ultrasound reconstruction and visualization. [10, 16, 32, 33, 90, 91, 104]

[Miller et al., 2012] Miller, D., Lippert, C., Vollmer, F., Bozinov, O., Benes, L., Schulte, D., and Sure, U. (2012). Comparison of different reconstruction algorithms for three-dimensional ultrasound imaging in a neurosurgical setting. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 8(3):348–359. [8, 10, 30, 104]

[NDI, 2014] NDI (2014). Polaris spectra and polaris vicra system configurations. http://www.ndigital.com/medical/polarisfamily-techspecs.php. [11]

[nVidia, 2009] nVidia (2009). Opencl programming guide for the cuda architecture. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf. [12]

[nVidia, 2013] nVidia (2013). Cuda c programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. [12]

[OpenAAC, 2013] OpenAAC (2013). The *openaac*^{TM} application programming interface. http://www.openacc.org/sites/default/files/OpenACC20.pdf. [12]

[Solberg et al., 2007] Solberg, O. V., Lindseth, F., Torp, H., Blake, R. E., and Hernes, T. A. N. (2007). Freehand 3D Ultrasound Reconstruction Algorithms—A Review. *Ultrasound in Medicine & Biology*, 33(7):991 – 1009. [8, 30, 103]

[Strandness Jr. et al., 1967] Strandness Jr., D., Schultz, R., Sumner, D., and Rushmer, R. (1967). Ultrasonic flow detection: A useful technic in

the evaluation of peripheral vascular disease. *The American Journal of Surgery*, 113(3):311 – 320. [6]

[Valderhaug, 2010] Valderhaug, T. K. (2010). Real-time GPU-based Freehand 3D Ultrasound Reconstruction. [16]

[Wein et al., 2006] Wein, W., Pache, F., Röper, B., and Navab, N. (2006). Backward-warping ultrasound reconstruction for improving diagnostic value and registration. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2006*, pages 750–757. Springer. [32, 44, 104]

[Wiora, 2005] Wiora, G. (2005). Principle of a sonar or radar distance measurement. http://en.wikipedia.org/wiki/File:Sonar_Principle_EN.svg. [6]

[Yu and Acton, 2002] Yu, Y. and Acton, S. (2002). Speckle reducing anisotropic diffusion. *Image Processing, IEEE Transactions on*, 11(11):1260–1270. [7]

[Øygard, 2013] Øygard, T. (2013). Aliasing- and angle correction for Doppler ultrasound measurements in 3D. [16]

Appendix A

Annotated Bibliography

In this chapter we will give a brief summary of selected entries from the bibliography of this thesis. References to the bibliography entries is included in the paragraph headers.

Freehand 3D Ultrasound Reconstruction Algorithms—a review [Solberg et al., 2007] This paper by O.V. Solberg et.al gives a comprehensive summary of the state of the art of Freehand 3D Ultrasound Reconstruction in 2007. Solberg also introduced the classification into "Voxel-based", "Pixel-based" and "Function-based" methods.

A new adaptive interpolation algorithm for 3D ultrasound imaging with speckle reduction and edge preservation[Huang et al., 2009] Huang et.al. in this paper presents a novel reconstruction algorithm similar to the one presented in this thesis. The core idea of adjusting the filter based on the data present came from this approach. However, the implementation details were scarce, as were the results in terms of computational speed – but they delivered very good results in terms of image quality.

Volume reconstruction of freehand three-dimensional ultrasound using median filters[Huang and Zheng, 2008] The approach to use median filters was also a huge inspiration in this thesis. In this paper, Huang et.al. presents an algorithm using median filters on all values falling inside

the radius to select the value to go into a voxel. This method had excellent speckle suppression, but was very computationally demanding.

3D Freehand Ultrasound Reconstruction Based on Probe Trajectory[Coupé et al., 2005] Coupé et. al. presented a novel 3D Ultrasound Reconstruction which takes the Probe Trajectory into account – instead of performing orthogonal projection onto the image plane, they estimate the movement of the ultrasound probe and use that movement estimation to find the position on the image plane. They show impressive reconstruction quality, especially considering the low computational complexity of the approach.

Backward-warping ultrasound reconstruction for improving diagnostic value and registration[Wein et al., 2006] Wein et.al. in this paper presents a fast voxel-based method for 3D Ultrasound Reconstruction implemented on a CPU. This paper, among other things, presents a very efficient and sound way to locate the image planes close to a voxel.

Comparison of different reconstruction algorithms for three-dimensional ultrasound imaging in a neurosurgical setting[Miller et al., 2012] This paper evaluates the algorithms MPR, PNN, VNN, VNN2 and DW, 4 of which are also evaluated in this thesis. After both a subjective evaluation similar to the visual evaluation in this thesis, and a slice-skip evaluation also similar to the evaluation in this thesis, Miller et.al. finds that VNN2 and DW produce the best results.

Real-time GPU-based 3D ultrasound reconstruction and visualization[Ludvigsen, 2010] This Master's thesis could be regarded as a direct predecessor of this thesis. Ludvigsen paved the way into the realm of GPU-based 3D ultrasound reconstruction, presenting several algorithms implemented on a GPU, with tremendous speedup compared to CPU implementations.

Appendix B

Additional Results

B.1 32M images vs 128M images

In this section we will display slices from volumes with 32M voxels side-by-side with aligned slices from volumes with 128M voxels.

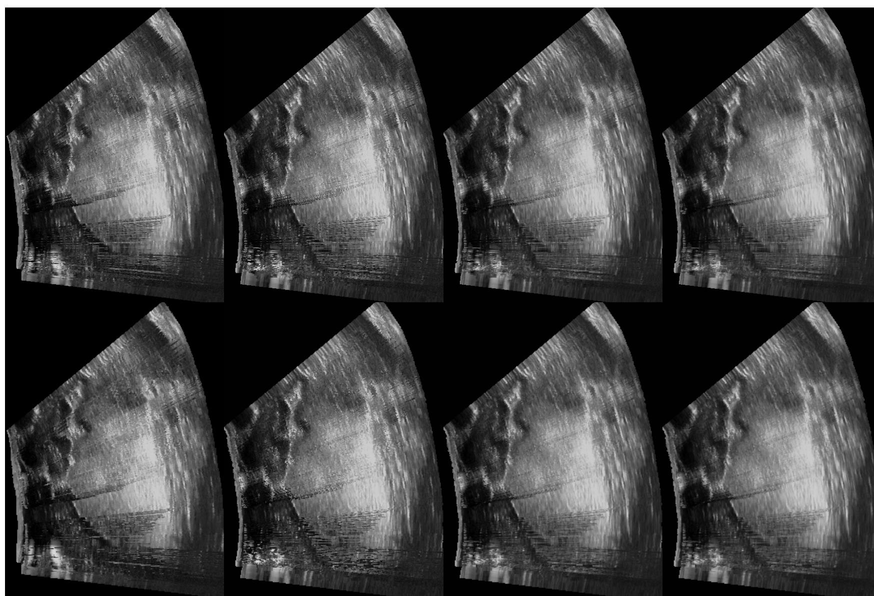


Figure B.1: Images comparing 128M volumes to 32M volumes, X direction
Top row: 128M volumes. Bottom row: 32M volumes
Method, from left: PNN, VNN, VNN2 with 4 planes, VNN2 with 8 planes

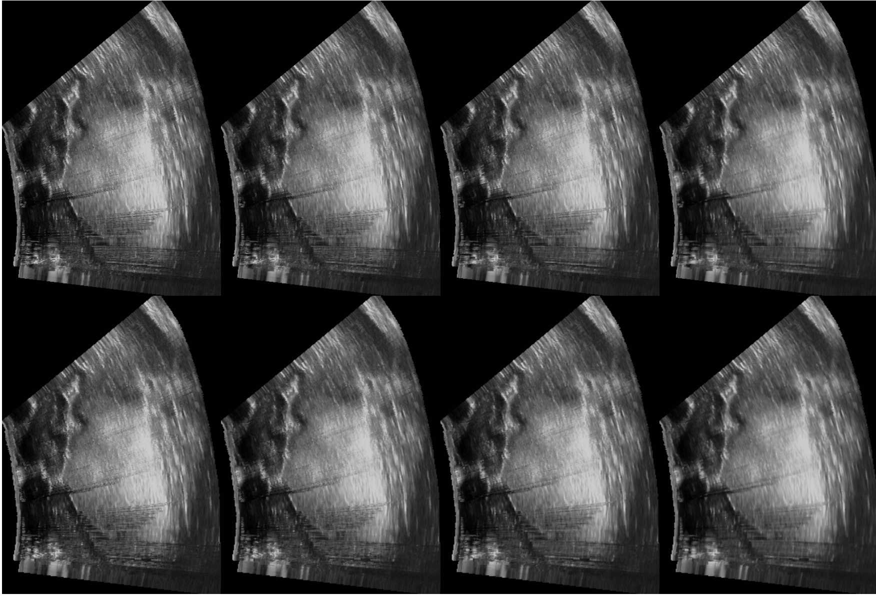


Figure B.2: Images comparing 128M volumes to 32M volumes, X direction
Top row: 128M volumes. Bottom row: 32M volumes
Method, from left: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

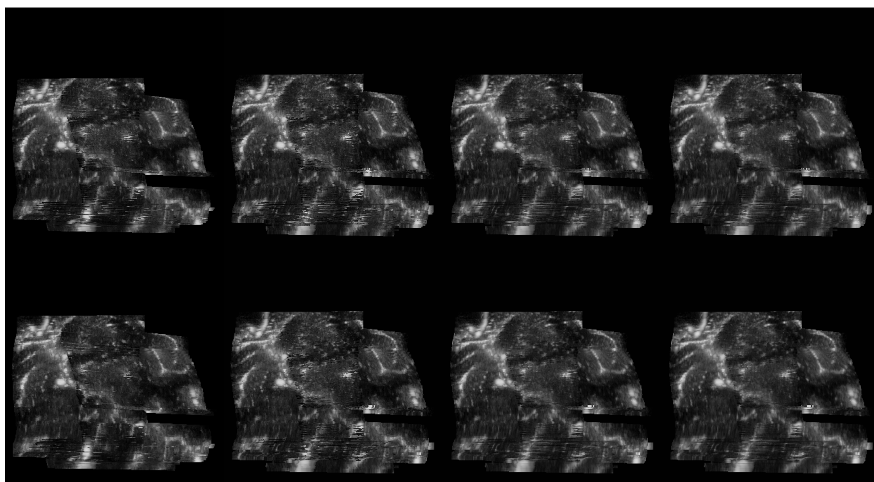


Figure B.3: Images comparing 128M volumes to 32M volumes, Y direction
Top row: 128M volumes. Bottom row: 32M volumes
Method, from left: PNN, VNN, VNN2 with 4 planes, VNN2 with 8 planes

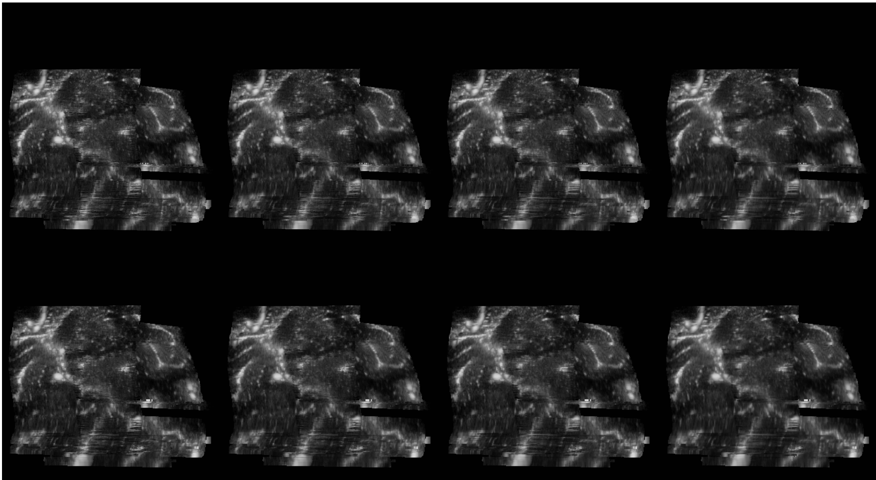


Figure B.4: Images comparing 128M volumes to 32M volumes, Y direction
Top row: 128M volumes. Bottom row: 32M volumes
Method, from left: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

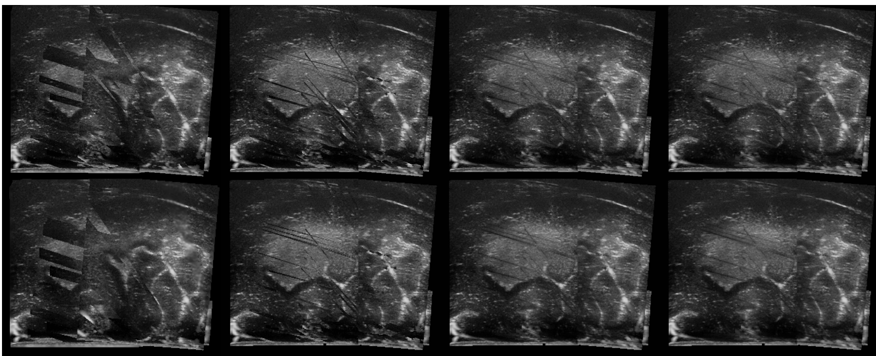


Figure B.5: Images comparing 128M volumes to 32M volumes, Z direction
Top row: 128M volumes. Bottom row: 32M volumes
Method, from left: PNN, VNN, VNN2 with 4 planes, VNN2 with 8 planes

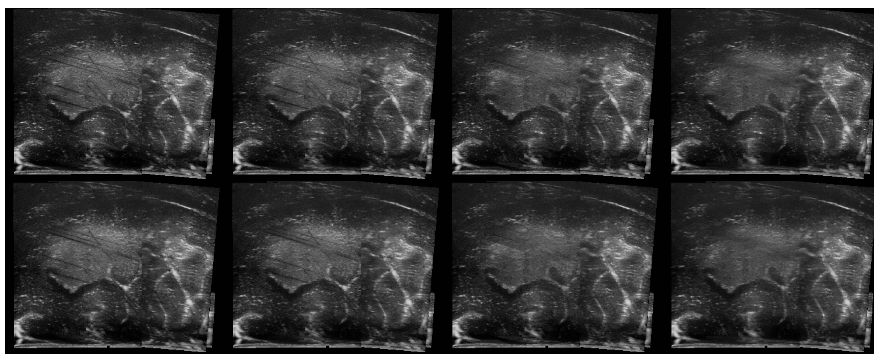


Figure B.6: Images comparing 128M volumes to 32M volumes, Z direction
Top row: 128M volumes. Bottom row: 32M volumes
Method, from left: DW with 4 planes, DW with 8 planes, VGDW with 4 planes, VGDW with 8 planes

Appendix C

Code Listings

This appendix contains the full OpenCL code produced during the course of the work. A few notes about the code to make it more understandable: There is a single kernel, `voxel_methods`, which is the entry point of all methods. Which method is in use, and the parameters is selected using defines at compile time.

```
/******************/
/* Begin constants */
/******************/

#define CUBE_SIZE 4

// Reconstruction methods
#define METHOD_VNN 0
#define METHOD_VNN2 1
#define METHOD_DW 2
#define METHOD_VGDW 3

// Plane searching methods
#define PLANE_HEURISTIC 0
#define PLANE_CLOSEST 1

/****************/
/* End constants */
/****************/


/***************/
/* Begin macros */
```

```
/****************/

//#define DEBUG
#define CHECK_PLANE_INDICES

#ifdef DEBUG
#define DEBUG_PRINTF(...) if((get_global_id(0) % 5000) == 0)
    printf(##__VA_ARGS__)
//#define DEBUG_PRINTF(...) printf(##__VA_ARGS__)
//#define BOUNDS_CHECK(x, min, max) if(x < min || x >= max)
    printf("Line %d: %s out of range: %d min: %d max: %d\n",
    __LINE__, #x, x, min, max)
#define BOUNDS_CHECK(x, min, max)

#else
#define DEBUG_PRINTF(...)
#define BOUNDS_CHECK(x, min, max)
#endif


#define plane_dist(voxel, matrix) (dot(matrix.s26AE,voxel) - dot
    (matrix.s26AE, matrix.s37BF))

#define euclid_dist(a, b, c) sqrt((a)*(a) + (b)*(b) + (c)*(c))

#define projectOntoPlane(voxel, matrix, dist) (voxel - dist*(
    matrix.s26AE))

#define projectOntoPlaneEq(voxel, eq, dist) (voxel - dist*(eq))

#define isInside(x, size) ((x) >= 0 && (x) < (size))
#define isNotMasked(x, y, mask, xsize) ((mask)[(x) + (y)*(xsize)
    ] > 0)
//#define isNotMasked(x, y, mask, xsize) true

#define VOXEL(v,x,y,z) v[x + y*volume_xsize + z*volume_ysize*
    volume_xsize]

#define WEIGHT_INV(x) (1.0f/fabs(x))

#define WEIGHT_TERNARY(val, mean, factor)
                                        \
  ((val) >= (mean) ? (factor) : 0.0f)

#define WEIGHT_GAUSS_SQRT_2PI 2.506628275f

#define WEIGHT_GAUSS_NONEXP_PART(sigma) (1.0f/((sigma)*
    WEIGHT_GAUSS_SQRT_2PI))
#define WEIGHT_GAUSS_EXP_PART(dist, sigma) exp(-((dist)*(dist))
    /(2*(sigma)*(sigma)))
```

```
#define WEIGHT_GAUSS(x, sigma) (WEIGHT_GAUSS_NONEXP_PART(sigma)*
    WEIGHT_GAUSS_EXP_PART(x, sigma))

#define VGDW_GAUSS_WEIGHT(px, var, mean, mean_id, sigma)
    WEIGHT_GAUSS(px.dist, sigma)

#ifndef BRIGHTNESS_FACTOR
#define BRIGHTNESS_FACTOR 0.0f
#endif
#ifndef NEWNESS_FACTOR
#define NEWNESS_FACTOR 0.0f
#endif

#define VGDW_WEIGHT_BRIGHTNESS(px, var, mean, mean_id, sigma)
       \
  ((WEIGHT_GAUSS(px.dist, sigma)) + (WEIGHT_TERNARY(px.intensity
      , mean, BRIGHTNESS_FACTOR)))

#define VGDW_WEIGHT_LATENESS(px, var, mean, mean_id, sigma)
         \
  ((WEIGHT_GAUSS(px.dist, sigma)) + (WEIGHT_TERNARY(px.plane_id,
       mean_id, NEWNESS_FACTOR)))


#define VGDW_WEIGHT(px, var, mean, mean_id, sigma) \
  ((WEIGHT_GAUSS(px.dist, sigma)) \
   + (WEIGHT_TERNARY(px.plane_id, mean_id, NEWNESS_FACTOR)) \
   + (WEIGHT_TERNARY(px.intensity, mean, BRIGHTNESS_FACTOR)))

#define DW_WEIGHT(x) WEIGHT_INV(x)
#define VNN2_WEIGHT(x) WEIGHT_INV(x)

#define CLOSE_PLANE_IDX(p, i) p[get_local_id(0)*(MAX_PLANES+1)+(
    i)]

/**************/
/* End macros */
/**************/

/****************/
/* Begin structs */
/****************/

typedef struct _close_plane
{
  float dist;
  short plane_id;
  unsigned char intensity;
  unsigned char padding; // Align with 4
```

```
} close_plane_t;

/***************/
/* End structs */
/***************/

/********************/
/* Begin prototypes */
/********************/

// Declare all the functions, as Apple seems to need that

int isValidPixel(int x,
                 int y,
                 const __global unsigned char* mask,
                 int in_xsize,
                 int in_ysize);

int
findHighestIdx(__local close_plane_t *planes,
               int n);
int2
findClosestPlanes_heuristic(__local close_plane_t *close_planes,
                            __local float4* const plane_eqs,
                            __constant float16* const
                                plane_matrices,
                            const float4 voxel,
                            const float radius,
                            int guess,
                            bool doTermDistance,
                            __global const unsigned char* mask,
                            int in_xsize,
                            int in_ysize,
                            float in_xspacing,
                            float in_yspacing);

int2
findClosestPlanes_multistart(__local close_plane_t *close_planes
    ,
                             __local float4* const plane_eqs,
                             __constant float16* const
                                 plane_matrices,
                             const float4 voxel,
                             const float radius,
                             int *multistart_guesses,
                             int n_multistart_guesses,
                             bool doTermDistance,
                             __global const unsigned char* mask,
                             int in_xsize,
                             int in_ysize,
```

```
                                    float in_xspacing,
                                    float in_yspacing);


// I'm sorry about this..
#if PLANE_METHOD == PLANE_EXACT
#define FIND_CLOSE_PLANES(a, b, c, d, e, f, g, h, i, j, k, l)
    findClosestPlanes_multistart(a, b, c, d, e, f, g, 1, h, i, j
    , k, l)
#elif PLANE_METHOD == PLANE_CLOSEST

#ifdef MAX_MULTISTART_STARTS
#undef MAX_MULTISTART_STARTS
#define MAX_MULTISTART_STARTS 1
#endif

#define FIND_CLOSE_PLANES(a, b, c, d, e, f, g, h, i, j, k, l)
    findClosestPlanes_multistart(a, b, c, d, e, f, g, 0, h, i, j
    , k, l)
#endif

float2
transform_inv_xy(float16 matrix, float4 voxel);

void
toImgCoord_int(int* x,
               int* y,
               float4 voxel,
               float16 plane_matrix,
               float in_xspacing,
               float in_yspacing);

void
toImgCoord_float(float* x,
                 float* y,
                 float4 voxel,
                 float16 plane_matrix,
                 float in_xspacing,
                 float in_yspacing);


unsigned char vgdwFilter(__local const close_plane_t *pixels,
                                int n_planes);

#if METHOD == METHOD_VNN

__constant const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
     | CLK_ADDRESS_CLAMP_TO_EDGE| CLK_FILTER_NEAREST;
unsigned char
performInterpolation_vnn(__local close_plane_t *close_planes,
```

```
                                   int n_close_planes,
                                   __constant const float16  *
                                       plane_matrices,
                                   __local const float4 *plane_eqs,
                                   __read_only image2d_array_t in_bscans,
                                   int in_xsize,
                                   int in_ysize,
                                   float in_xspacing,
                                   float in_yspacing,
                                   __global const unsigned char* mask,
                                   float4 voxel);
#endif

#if METHOD == METHOD_VNN2
__constant const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
      | CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;
unsigned char
performInterpolation_vnn2(__local close_plane_t *close_planes,
                                    int n_close_planes,
                                    __constant const float16  *
                                        plane_matrices,
                                    __local const float4 *plane_eqs,
                                    __read_only image2d_array_t in_bscans,
                                    int in_xsize,
                                    int in_ysize,
                                    float in_xspacing,
                                    float in_yspacing,
                                    __global const unsigned char* mask,
                                    float4 voxel);
#endif

#if METHOD == METHOD_DW
__constant const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
      | CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_LINEAR;
unsigned char
performInterpolation_dw(__local close_plane_t *close_planes,
                                 int n_close_planes,
                                 __constant const float16  *
                                     plane_matrices,
                                 __local const float4 *plane_eqs,
                               __read_only image2d_array_t in_bscans,
                                 int in_xsize,
                                 int in_ysize,
                                 float in_xspacing,
                                 float in_yspacing,
                                 __global const unsigned char* mask,
                                 float4 voxel);
#endif

#if METHOD == METHOD_VGDW
```

```
__constant const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
     | CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_LINEAR;
unsigned char
performInterpolation_vgdw(__local close_plane_t *close_planes,
                            int n_close_planes,
                            __constant const float16  *
                                plane_matrices,
                            __local const float4 *plane_eqs,
                                 __read_only image2d_array_t
                                     in_bscans,
                            int in_xsize,
                            int in_ysize,
                            float in_xspacing,
                            float in_yspacing,
                            __global const unsigned char* mask,
                            float4 voxel);
#endif

void
prepare_plane_eqs(__constant float16 *plane_matrices,
                  __local float4 *plane_eqs);


int
findLocalMinimas(int *guesses,
                 __local const float4 *plane_eqs,
                 float radius,
                 float4 voxel,
                 float out_xspacing,
                 float out_yspacing,
                 float out_zspacing,
                 float in_xspacing,
                 float in_yspacing,
                 __constant const float16 *plane_matrices,
                 __global const unsigned char *mask,
                 int in_xsize,
                 int in_ysize);


__kernel void
voxel_methods(int volume_xsize,
              int volume_ysize,
              int volume_zsize,
              float volume_xspacing,
              float volume_yspacing,
              float volume_zspacing,
              int in_xsize,
              int in_ysize,
              float in_xspacing,
              float in_yspacing,
```

```
                      __read_only image2d_array_t in_bscans,
                      __global unsigned char* out_volume,
                      __constant float16 *plane_matrices,
                      __global unsigned char* mask,
                      __local float4 *plane_eqs,
                      __local close_plane_t *planes,
                      float radius);


/*****************/
/* End prototypes */
/*****************/


int isValidPixel(int x,
                 int y,
                 const __global unsigned char* mask,
                 int in_xsize,
                 int in_ysize)
{
#ifndef DEBUG
  return (isInside(x, in_xsize)
          && isInside(y, in_ysize)
          && isNotMasked(x, y, mask, in_xsize));
#else
  if((isInside(x, in_xsize)
      && isInside(y, in_ysize)
      && isNotMasked(x, y, mask, in_xsize)))
  {
    return 1;
  }
  else {
    //     DEBUG_PRINTF("Pixel %d, %d is not valid! Sizes: %d, %d
        \n",
    //            x, y, in_xsize, in_ysize);
    return 0;
  }
#endif

}


/**
 * Find the plane with the highest distance to the voxel
 * i.e. the plane with the highest absolute value of dist.
 * Return the index of that plane
 * @param *planes Pointer to first element of plane array
 * @param n size of array pointed to *planes
```

```
 */
int
findHighestIdx(__local close_plane_t *planes,
               int n)
{
  int maxidx = 0;
  float maxval = -1.0f;
  planes = &CLOSE_PLANE_IDX(planes, 0);

  for(int i = 0; i < n; i++)
  {
    float abs = fabs(planes->dist);
    if(abs > maxval)
    {
      maxidx = i;
      maxval = abs;
    }
    planes++;
  }
  //  DEBUG_PRINTF("New maxidx: %d maxdist = %f\n", maxidx,
      maxval);
  BOUNDS_CHECK(maxidx, 0, MAX_PLANES);
  return maxidx;
}

int2
findClosestPlanes_multistart(__local close_plane_t *close_planes
    ,
                             __local float4* const plane_eqs,
                             __constant float16* const
                                 plane_matrices,
                             const float4 voxel,
                             const float radius,
                             int *multistart_guesses,
                              int n_multistart_guesses,
                              bool doTermDistance,
                              __global const unsigned char* mask,
                             int in_xsize,
                             int in_ysize,
                             float in_xspacing,
                             float in_yspacing)
{
  close_plane_t  tmp;
  tmp.dist = INFINITY;
  tmp.plane_id = -1;
  for(int i = 0; i < MAX_PLANES; i++)
  {
    CLOSE_PLANE_IDX(close_planes, i) = tmp;
  }
```

```
  int2 ret;
  int found = 0;
  for(int i = 0; i < n_multistart_guesses; i++)
  {
    ret = findClosestPlanes_heuristic(close_planes,
                                      plane_eqs,
                                      plane_matrices,
                                      voxel,
                                      radius,
                                      multistart_guesses[i],
                                      doTermDistance,
                                      mask,
                                      in_xsize,
                                      in_ysize,
                                      in_xspacing,
                                      in_yspacing );
    if(ret.x > 0)
    {
      multistart_guesses[i] = ret.y;
    }
    found += ret.x;

  }

  ret.x = min(found, MAX_PLANES);
  ret.y = 0;

#ifdef DEBUG
#ifdef CHECK_PLANE_INDICES
  for(int i = 0; i < min(found, MAX_PLANES); i++)
  {
    BOUNDS_CHECK(CLOSE_PLANE_IDX(close_planes, i).plane_id, 0,
        N_PLANES);
  }
#endif
#endif

  return ret;
}

/**
 * Find planes that are within radius of voxel.
 * Search in both directions in the plane array, starting at
    guess
 * The assumption is that as you move away from the guess,
 * the distance to this voxel will increase. That assumption may
     not always be true, for instance
 * if the US probe was swept back and forth.
 * Finds the closest MAX_PLANES planes within radius,
 * provided no plane with distance greater than
```

```
 * 2x radius is found before any of the MAX_PLANES closest
     planes.
 */
int2
findClosestPlanes_heuristic(__local close_plane_t *close_planes,
                            __local float4* const plane_eqs,
                            __constant float16* const
                                plane_matrices,
                            const float4 voxel,
                            const float radius,
                            int guess,
                            bool doTermDistance,
                            __global const unsigned char* mask,
                            int in_xsize,
                            int in_ysize,
                            float in_xspacing,
                            float in_yspacing)
{

  // Number of planes found so far
  int found = 0;

  // Done condition. .x  = up, .y = down
  int2 done = {0,0};

  // The index of the plane with the smallest distance found so
      far
  int smallest_idx = guess;

  float term_condition = clamp(fabs(dot(voxel, plane_eqs[guess])
      ), radius, 3*radius);

  // The smallest distance found so far
  float smallest_dist = 99999.9f;

  // The index of the plane with the biggest index so far
  int max_idx = findHighestIdx(close_planes, MAX_PLANES);

  // The biggest distance found so far
  float max_dist = min(fabs(CLOSE_PLANE_IDX(close_planes,
      max_idx).dist), radius);

  close_plane_t tmp;
  tmp.intensity = 0;

  // If guess is 0, we will try to access data for plane id -1,
      which does not exist.
  // Assume plane 1 is close enough in that case
  if(guess == 0) guess = 1;
```

```
479 // We won't be changing the guess, but the compiler wouldn't
        know that
    const int tmp_guess = guess;
481 BOUNDS_CHECK(tmp_guess, 1, N_PLANES);
    float2 dists = {dot(voxel, plane_eqs[guess]), dot(voxel,
        plane_eqs[guess-1])};
483 float2 abs_dists = {fabs(dists.x), fabs(dists.y)}; // .x =
        abs_dist_up, .y = abs_dist_down,

485 for(int i = 0; !done.x || !done.y ; i++)
    {
487   // Compute the indices of the planes we want to look at.
      int2 idx = {tmp_guess + i, tmp_guess - i - 1};
489
      if(idx.y <=0)
491     idx.y = 0;
      if(idx.x >= N_PLANES-1)
493     idx.x = N_PLANES-1;

495   //float2 prev_abs_dists = abs_dists;

497   // Compute the distances to those planes
      dists.x = dot(voxel, plane_eqs[idx.x]);
499   dists.y = dot(voxel, plane_eqs[idx.y]);
      // Compute the absolute distances to those planes
501   abs_dists.x = fabs(dists.x);
      abs_dists.y = fabs(dists.y); // .x = abs_dist_up, .y =
          abs_dist_down,
503

505   //float2 diff_dists = prev_abs_dists - abs_dists;

507   // Check if the plane is closer than the one farthest away
          we have included so far
      if(!done.x && abs_dists.x < max_dist)
509   {
        BOUNDS_CHECK(idx.x, 0, N_PLANES);
511     BOUNDS_CHECK(max_idx, 0, MAX_PLANES);
        int px, py;
513     float4 translated_voxel = projectOntoPlaneEq(voxel,
                                                      plane_eqs[idx.x
                                                          ],
515                                                   dists.x);
        toImgCoord_int(&px,
517                    &py,
                       translated_voxel,
519                    plane_matrices[idx.x],
                       in_xspacing,
521                    in_yspacing);
```

```
      if(isValidPixel(px, py, mask, in_xsize, in_ysize))
      {

        // If yes, swap out the one with the longest distance
            for this plane
        tmp.dist = dists.x;
        tmp.plane_id = idx.x;
        CLOSE_PLANE_IDX(close_planes, max_idx) = tmp;
        found++;

        // We have found MAX_PLANES planes, but we don't know
            they're the closest ones.
        // Find the next candidate for eviction -
        // the plane with the longest distance to the voxel
        max_idx = findHighestIdx(close_planes, MAX_PLANES);
        max_dist = min(fabs(CLOSE_PLANE_IDX(close_planes,
            max_idx).dist), radius);

        if(smallest_dist > abs_dists.x)
        {
          // Update next guess
          smallest_dist = abs_dists.x;
          smallest_idx = idx.x;
        }
      }
    }

    // And the same in the down direction
    // Check if the plane is closer than the one farthest away
        we have included so far
    if(!done.y && abs_dists.y < max_dist)
    {
      BOUNDS_CHECK(idx.y, 0, N_PLANES);
      BOUNDS_CHECK(max_idx, 0, MAX_PLANES);
      // If yes, swap out the one with the longest distance for
          this plane
      int px, py;
      float4 translated_voxel = projectOntoPlaneEq(voxel,
                                                   plane_eqs[idx.y
                                                       ],
                                                   dists.y);

      toImgCoord_int(&px,
                     &py,
                     translated_voxel,
                     plane_matrices[idx.y],
                     in_xspacing,
                     in_yspacing);
      if(isValidPixel(px, py, mask, in_xsize, in_ysize))
```

```
      {
567     tmp.dist = dists.y;
        tmp.plane_id = idx.y;
569     CLOSE_PLANE_IDX(close_planes, max_idx) = tmp;

571     found++;
        max_idx = findHighestIdx(close_planes, MAX_PLANES);
573     max_dist = min(fabs(CLOSE_PLANE_IDX(close_planes,
            max_idx).dist), radius);
        if(smallest_dist > abs_dists.y)
575     {
          // Update next guess
577       smallest_dist = abs_dists.y;
          smallest_idx = idx.y;
579     }
      }
581   }

583
      int2 term_dists = {(abs_dists.x > term_condition)*
          doTermDistance, (abs_dists.y > term_condition)*
          doTermDistance };
585
      //int2 term_radius_jump = {fabs(diff_dists.x) > radius, fabs
          (diff_dists.y) > radius};
587   int2 term_boundaries = {idx.x == N_PLANES-1, idx.y == 0};

589   done = done + term_dists + term_boundaries;// +
          term_radius_jump;
    }
591
    int2 ret;
593 ret.x = min(found, MAX_PLANES);
    ret.y = smallest_idx;
595 return ret;
  }
597

599 /**
     * Perform an inverse transformation of voxel, but only
         transform the x and y coordinates. This is useful
601  * when finding image coordinates.
     */
603 float2 transform_inv_xy(float16 matrix, float4 voxel)
  {
605   float2 ret;
      float4 col0 = matrix.s048C;
607   float4 col1 = matrix.s159D;
      float4 col3 = matrix.s37BF;
609
```

```
  ret.x = dot(voxel,col0) - dot(col3,col0);
  ret.y = dot(voxel,col1) - dot(col3,col1);
  return ret;
}

/**
 * Transform to integer image coordinates - i.e. pixel
     coordinates
 */
void toImgCoord_int(int* x,
                    int* y,
                    float4 voxel,
                    float16 plane_matrix,
                    float in_xspacing,
                    float in_yspacing)
{

  float2 transformed_voxel = transform_inv_xy(plane_matrix,
      voxel);

  *x = ((transformed_voxel.x/in_xspacing) + 0.5f);
  *y = ((transformed_voxel.y/in_yspacing) + 0.5f);
}

/**
 * Transform to floating point image coordinates
 */
void toImgCoord_float(float* x,
                float* y,
                float4 voxel,
                float16 plane_matrix,
                float in_xspacing,
                float in_yspacing)
{

  float2 transformed_voxel = transform_inv_xy(plane_matrix,
      voxel);

  *x = ((transformed_voxel.x/in_xspacing));
  *y = ((transformed_voxel.y/in_yspacing));
}



#if METHOD == METHOD_VNN
#define PERFORM_INTERPOLATION(a, b, c, d, e, f, g, h, i , j, k)
      \
  performInterpolation_vnn(a, b, c, d, e, f, g, h, i, j, k)

/**
```

```
 * Perform interpolation using the Voxel Nearest Neighbour
     method.
657 * This works by taking finding the plane closest to the voxel,
 * projecting the voxel orthogonally onto the image plane to
     find pixel coordinates
659 * and taking the pixel value
 */
661 unsigned char
performInterpolation_vnn(__local close_plane_t *close_planes,
663                          int n_close_planes,
                         __constant const float16 *
                             plane_matrices,
665                          __local const float4 *plane_eqs,
                         __read_only image2d_array_t in_bscans,
667                          int in_xsize,
                         int in_ysize,
669                          float in_xspacing,
                         float in_yspacing,
671                          __global const unsigned char* mask,
                         float4 voxel)
673 {
  if(n_close_planes == 0) return 1;
675
  int plane_id = 0;
677   float lowest_dist = 10.0f;
  int close_plane_id = 0;
679   close_plane_t plane;
  // Find the closest plane
681   for(int i = 0; i < n_close_planes;  i++)
  {
683     plane = CLOSE_PLANE_IDX(close_planes, i);
    float fabs_dist = fabs(plane.dist);
685     if(fabs_dist < lowest_dist)
    {
687       lowest_dist = fabs_dist;
      plane_id = plane.plane_id;
689       close_plane_id = i;
    }
691   }
  BOUNDS_CHECK(plane_id, 0, N_PLANES);
693

695   // Now we project the voxel onto the plane by translating the
      voxel along the
  // normal vector of the plane.
697   float4 translated_voxel = projectOntoPlane(voxel,
                                             plane_matrices[
                                                 plane_id],
699                                              CLOSE_PLANE_IDX(
                                                 close_planes ,
```

```
                                                  close_plane_id)
                                                  .dist);
  translated_voxel.w = 1.0f;

  // And then we get the pixel space coordinates
  int x, y;
  toImgCoord_int(&x,
                 &y,
                 translated_voxel,
                 plane_matrices[plane_id],
                 in_xspacing,
                 in_yspacing);
  int4 coord = {x, y, plane_id, 0};
  if(!isValidPixel(x,y, mask, in_xsize, in_ysize))
  {
    return 1;
  }
  BOUNDS_CHECK(x, 0, in_xsize);
  BOUNDS_CHECK(y, 0, in_ysize);
  return max((unsigned int)1, read_imageui(in_bscans, sampler,
      coord).x);

}
#endif

#if METHOD == METHOD_VNN2
#define PERFORM_INTERPOLATION(a, b, c, d, e, f, g, h, i ,j, k)
      \
  performInterpolation_vnn2(a, b, c, d, e, f, g, h, i, j, k)

/**
 * Perform interpolation using the VNN2 method. For each close
     plane, add (1/dist)*closest_pixel_value to the sum.
 * In the end, divide sum by sum(1/dist), and you have your
     voxel value.
 */
unsigned char
performInterpolation_vnn2(__local close_plane_t *close_planes,
                          int n_close_planes,
                          __constant const float16 *
                              plane_matrices,
                          __local const float4 *plane_eqs,
                          __read_only image2d_array_t in_bscans,
                          int in_xsize,
                          int in_ysize,
                          float in_xspacing,
                          float in_yspacing,
                          __global const unsigned char* mask,
                          float4 voxel)
```

```
743 {
      if(n_close_planes == 0) return 1;
745

747   float scale = 0.0f;

749   float val = 0;
      for(int i = 0; i < n_close_planes; i++)
751   {
        close_plane_t plane = CLOSE_PLANE_IDX(close_planes, i);
753     int plane_id = plane.plane_id;

755     // Now we project the voxel onto the plane by translating
            the voxel along the
        // normal vector of the plane.
757     voxel.w = 1.0f;
        float4 translated_voxel = projectOntoPlaneEq(voxel,
759                                                  plane_eqs[
                                                         plane_id],
                                                     plane.dist);
761
        translated_voxel.w = 1.0f;
763     // And then we get the pixel space coordinates
        int x, y;
765     toImgCoord_int(&x,
                       &y,
767                    translated_voxel,
                       plane_matrices[plane_id],
769                    in_xspacing,
                       in_yspacing);
771
        if(!isValidPixel(x,y, mask, in_xsize, in_ysize))
773     {
          continue;
775     }
        float dist = fabs(plane.dist);
777
        if(dist < 0.001f)
779       dist = 0.001f;
        float weight = VNN2_WEIGHT(dist);
781     int4 coord = {x, y, plane_id, 0};
        scale += weight;
783     val += (read_imageui(in_bscans, sampler, coord).x * weight);
      }
785

787   return max((unsigned char)1, (unsigned char) ((val / scale) +
          0.5f));

789 }
```

```
#endif
791
#if METHOD == METHOD_DW
793 #define PERFORM_INTERPOLATION(a, b, c, d, e, f, g, h, i ,j, k)
      \
  performInterpolation_dw(a, b, c, d, e, f, g, h, i, j, k)
795
/**
797  * Perform interpolation using the DW method. Works the same as
     VNN2, but instead of taking the closest pixel on each image
      plane,
 * the value from each plane is a bilinearly interpolated from
     that plane.
799  */
unsigned char
801 performInterpolation_dw(__local close_plane_t *close_planes,
                         int n_close_planes,
803                      __constant const float16  *
                             plane_matrices,
                         __local const float4 *plane_eqs,
805                      __read_only image2d_array_t in_bscans,
                         int in_xsize,
807                      int in_ysize,
                         float in_xspacing,
809                      float in_yspacing,
                         __global const unsigned char* mask,
811                      float4 voxel)
{
813

815   if(n_close_planes == 0) return 1;

817
  float scale = 0.0f;
819
  float val = 0;
821   for(int i = 0; i < n_close_planes; i++)
  {
823     close_plane_t plane = CLOSE_PLANE_IDX(close_planes, i);
    int plane_id = plane.plane_id;
825

827     // Now we project the voxel onto the plane by translating
        the voxel along the
    // normal vector of the plane.
829     voxel.w = 1.0f;
    float4 translated_voxel = projectOntoPlaneEq(voxel,
831                                                  plane_eqs[
                                                     plane_id],
                                                 plane.dist);
```

```
833
      translated_voxel.w = 1.0f;
835   // And then we get the pixel space coordinates
      float x, y;
837   toImgCoord_float(&x,
                 &y,
839              translated_voxel,
                 plane_matrices[plane_id],
841              in_xspacing,
                 in_yspacing);
843

845   // The OpenCL spec defines the linear filtering to be shited
          by 0.5f for some reason
      x += 0.5f;
847   y += 0.5f;
      int ix, iy;
849   ix = x;
      iy = y;
851   if(!isValidPixel(ix,iy, mask, in_xsize, in_ysize)
         || !isValidPixel(ix+1, iy, mask, in_xsize, in_ysize)
853      || !isValidPixel(ix+1, iy+1, mask, in_xsize, in_ysize)
         || !isValidPixel(ix, iy+1, mask, in_xsize, in_ysize)
855     )
      {
857     continue;
      }
859   float4 coord = {x, y, plane_id, 0};

861   float interpolated_value = read_imageui(in_bscans, sampler,
          coord).x;

863   float dist = fabs(plane.dist);
      if(dist < 0.001f) dist = 0.001f;
865   float weight = DW_WEIGHT(dist);
      scale += weight;
867   val += (interpolated_value * weight);
    }
869

871 return max((unsigned char)1, (unsigned char) ((val / scale) +
        0.5f));

873 }
    #endif
875
    #if METHOD == METHOD_VGDW
877 #define PERFORM_INTERPOLATION(a, b, c, d, e, f, g, h, i ,j, k)
        \
      performInterpolation_vgdw(a, b, c, d, e, f, g, h, i, j, k)
```

```
/**
 * Perform interpolation using VGDW filter
 */
unsigned char
performInterpolation_vgdw(__local close_plane_t *close_planes,
                          int n_close_planes,
                          __constant const float16  *
                              plane_matrices,
                          __local const float4 *plane_eqs
                              ,
                          __read_only image2d_array_t
                              in_bscans,
                          int in_xsize,
                          int in_ysize,
                          float in_xspacing,
                          float in_yspacing,
                          __global const unsigned char*
                              mask,
                          float4 voxel)
{

  if(n_close_planes == 0)
  {
    return 1;
  }
  int found_planes = 0;
  for(int i = 0; i < n_close_planes; i++)
  {
    close_plane_t plane = CLOSE_PLANE_IDX(close_planes ,i);
    const int plane_id = plane.plane_id;

    // Project onto plane
    voxel.w = 1.0f;
    float4 translated_voxel = projectOntoPlaneEq(voxel,
                                                 plane_eqs[
                                                     plane_id],
                                                 plane.dist);
    translated_voxel.w = 1.0f;

    float x, y;
    toImgCoord_float(&x,
                     &y,
                     translated_voxel,
                     plane_matrices[plane_id],
                     in_xspacing,
                     in_yspacing);

    // The OpenCL spec defines the linear filtering to be shited
```

```
         by 0.5f for some reason
    x += 0.5f;
925   y += 0.5f;

927   int ix, iy;
    ix = x;
929   iy = y;
    if(!isValidPixel(ix,iy, mask, in_xsize, in_ysize)
931       || !isValidPixel(ix+1, iy, mask, in_xsize, in_ysize)
       || !isValidPixel(ix+1, iy+1, mask, in_xsize, in_ysize)
933       || !isValidPixel(ix, iy+1, mask, in_xsize, in_ysize)
        )
935     {
      continue;
937   }
    float4 coord = {x, y, plane_id, 0};
939   CLOSE_PLANE_IDX(close_planes, found_planes).intensity =
        read_imageui(in_bscans, sampler, coord).x;
    found_planes++;
941 }
  return max((unsigned char)1, vgdwFilter(close_planes,
      found_planes));
943
}
945

947
unsigned char vgdwFilter(__local const close_plane_t *pixels,
949                                 int n_planes)
{
951   // Calculate the variance

953   float mean_value = 0.0f;
  int sum_ids = 0.0f;
955   close_plane_t tmp;
  for(int i = 0; i < n_planes; i++)
957   {
    tmp = CLOSE_PLANE_IDX(pixels, i);
959     mean_value += tmp.intensity;
    sum_ids += tmp.plane_id;
961   }
  float mean_id = (float)sum_ids / n_planes;
963   mean_value = mean_value / n_planes;

965   float variance = 0.0f;
  for(int i = 0; i < n_planes; i++)
967   {
    float tmp = CLOSE_PLANE_IDX(pixels, i).intensity -
        mean_value;
969     variance += mad(tmp, tmp, variance);
```

```
  }

  // We want high variance regions to have a sharp weight
      function
  // and small variance regions to have a smooth weight function
      .

  variance = clamp(variance/(n_planes-1), 1.0f, 10000000.0f);
  float gauss_sigma = 32.0f/sqrt(variance);

#ifdef DEBUG
  if(variance > 0.1f && mean_value> 10.0f)
    DEBUG_PRINTF("Mean: %f, variance: %f, sigma: %f\n",
        mean_value, variance, gauss_sigma);
#endif

  float sum_weights = 0.0f;
  float sum = 0.0f;
  // Use the resulting gauss sigma to calcualte weights
  for(int i = 0; i < n_planes; i++)
  {
    tmp = CLOSE_PLANE_IDX(pixels, i);
    float weight = VGDW_WEIGHT(tmp, variance, mean_value,
        mean_id, gauss_sigma);
    sum = mad(tmp.intensity,weight, sum);
    sum_weights += weight;
  }
  return (sum / sum_weights) + 0.5f;
}



#endif
/**
 * Build the plane equations from the matrices and store them in
      local memory
 */
void
prepare_plane_eqs(__constant float16 *plane_matrices,
                  __local float4 *plane_eqs)
{
  int id = get_local_id(0);
  int max_local_id = get_local_size(0);
  const int n_planes_pr_thread = (N_PLANES / max_local_id) + 1;

  for(int i = 0; i < n_planes_pr_thread; i++)
  {
    int idx = i + n_planes_pr_thread * id;
    if(idx >= N_PLANES) break;
    plane_eqs[idx].xyz = plane_matrices[idx].s26A;
```

```
    plane_eqs[idx].w = -dot(plane_matrices[idx].s26AE,
        plane_matrices[idx].s37BF);
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}

int findLocalMinimas(int *guesses,
                     __local const float4 *plane_eqs,
                     float radius,
                     float4 voxel,
                     float out_xspacing,
                     float out_yspacing,
                     float out_zspacing,
                     float in_xspacing,
                     float in_yspacing,
                     __constant const float16 *plane_matrices,
                     __global const unsigned char *mask,
                     int in_xsize,
                     int in_ysize)

{
  // Find all valleys in the search space of distances.
  // We don't need the _exact_ minima, however it should be
      inside the sweep we want.
  // However, the input data are noisy, so local minima in its
      strictest sense does not work for us.
  // But if we can find two indices a and b, such that dist(i) <
       dist(a) and dist(i) < dist(b)
  // and b - a = LOCAL_SEARCH_DISTANCE, it's a good chance it's
      a minima.

  int nMinima = 1;

  // Now with the cube-ish way of doing things, we may simply
      find all guesses that are closer than CUBE_SIZE *
      voxel_scale
  float max_dist = euclid_dist(out_xspacing * CUBE_SIZE,
      out_zspacing*CUBE_SIZE, out_yspacing*CUBE_SIZE) +  radius;
  DEBUG_PRINTF("Max dist is %f\n", max_dist);
  int prev_pos = 0;
  //float smallest_dist = fabs(dot(voxel, plane_eqs[0]));
  guesses[0] = 0;
  int hasHighSinceLastTaken = 1;
  for(int i = 0;
      i < N_PLANES;
      i++)
  {
    float dist = fabs(dot(voxel, plane_eqs[i]));
    if(dist < max_dist)
    {
```

```
1057
        // We are inside a local minima. Now, how do we know we
            haven't found this minima before?
1059    // We require that they be spaced by at least
            LOCAL_SEARCH_DISTANCE.
        // However, if this minima is better (i.e. closer) than
            the previous one
1061    // inside LOCAL_SEARCH_DISTANCE,
        // of course we want to use this one.
1063    if(!hasHighSinceLastTaken)
        {
1065      DEBUG_PRINTF("Minima %d: Found nearby minima: %d : %f\n"
              , nMinima, i, dist);
          // We have a previous minima, and it's too close.
1067      float prev_dist = fabs(dot(plane_eqs[guesses[prev_pos]],
               voxel));
          if(dist < prev_dist)
1069      {
            DEBUG_PRINTF("Taking it\n");
1071        // But this one is better, lets use it
            guesses[prev_pos] = i;
1073        hasHighSinceLastTaken = 0;
          }
1075    }
        else if(nMinima < MAX_MULTISTART_STARTS)
1077    {
          // We may simply store this minima
1079      DEBUG_PRINTF("Minima %d: Found new minima: %d : %f\n",
              nMinima, i, dist);
          guesses[nMinima] = i;
1081      prev_pos = nMinima;
          hasHighSinceLastTaken = 0;
1083      nMinima++;
        }
1085    else {
          // We already have MAX_MULTISTART_STARTS minimas, so now
               pick the "worst" minima
1087      // and toss it out for this one

1089      float biggest = -INFINITY;
          float tmp;
1091      int biggest_idx = 0;
          hasHighSinceLastTaken = 0;
1093      for(int j = 0; j < nMinima; j++)
          {
1095        tmp = fabs(dot(plane_eqs[guesses[j]], voxel));
            if(tmp > biggest)
1097        {
              biggest_idx = j;
1099          biggest = tmp;
```

```
          }
1101    }
        if(biggest > dist)
1103    {
          DEBUG_PRINTF("Switching out %d for %d: %f vs %f\n",
              biggest_idx, i, biggest, dist);
1105      guesses[biggest_idx] = i;
          prev_pos = biggest_idx;
1107    }
      }
1109  }
      else
1111  {
        hasHighSinceLastTaken = 1;
1113  }
    }
1115
    DEBUG_PRINTF("Found %d minima in total\n", nMinima);
1117
    return nMinima;
1119 }

1121 /** The entry point for this set of reconstruction methods.
      * Parameters:
1123  * @param volume_xsize Size of output volume, X direction
      * @param volume_ysize Size of output volume, Y direction
1125  * @param volume_zsize Size of output volume, Z direction
      * @param volume_xspacing Voxel size of output volume, X
          direction
1127  * @param volume_yspacing Voxel size of output volume, Y
          direction
      * @param volume_zspacing Voxel size of output volume, Z
          direction
1129  * @param in_xsize Size of each ultrasound input image in pixels
          , X direction
      * @param in_ysize Size of each ultrasound input image in pixels
          , Y direction
1131  * @param in_xspacing Size of each pixel in input ultrasound
          images, X direction
      * @param in_yspacing Size of each pixel in input ultrasound
          images, Y direction
1133  * @param in_bscans_b_ Ultrasound input images
      * @param out_volume Output volume - reconstructed volume goes
          here
1135  * @param plane_matrices One matrix per image plane specifying
          the transform from pixel space to voxel space
      * @param plane_eqs Pointer to local memory where we will store
          plane equations
1137  * @param radius The radius of the kernel - how far away to
          accept voxels from.
```

```
 */
1139 __kernel void
voxel_methods(int volume_xsize,
1141               int volume_ysize,
                  int volume_zsize,
1143               float volume_xspacing,
                  float volume_yspacing,
1145               float volume_zspacing,
                  int in_xsize,
1147               int in_ysize,
                  float in_xspacing,
1149               float in_yspacing,
                  __read_only image2d_array_t in_bscans,
1151               __global unsigned char* out_volume,
                  __constant float16 *plane_matrices,
1153               __global unsigned char *mask,
                  __local float4 *plane_eqs,
1155               __local close_plane_t *close_planes,
                  float radius
1157   )
{
1159
  int id = get_global_id(0);
1161
  int xcubes = (volume_xsize / CUBE_SIZE) + 1;
1163   int ycubes = (volume_ysize / CUBE_SIZE) + 1;

1165   int x_cube_id = id % xcubes;
  int y_cube_id = (id / xcubes) % ycubes;
1167   int z_cube_id = (id / (xcubes*ycubes));

1169   int x_origin = x_cube_id * CUBE_SIZE;
  int y_origin = y_cube_id * CUBE_SIZE;
1171   int z_origin = z_cube_id * CUBE_SIZE;

1173   #ifdef DEBUG
  if(id == 5000)
1175     BOUNDS_CHECK(id, 0, 1);
  #endif
1177
  int n_close_planes;
1179
  float4 voxel = {(x_origin) * volume_xspacing,
1181                   (y_origin) * volume_yspacing,
                  (z_origin)* volume_zspacing,
1183                   1.0f};

1185
  prepare_plane_eqs(plane_matrices, plane_eqs);
1187
```

```
1189    // Return if x/z is invalid

1191    if(z_origin >= volume_zsize) return;
        if(x_origin >= volume_xsize) return;
1193    if(y_origin >= volume_ysize) return;

1195    BOUNDS_CHECK(x_origin, 0, volume_xsize);
        BOUNDS_CHECK(y_origin, 0, volume_ysize);
1197    BOUNDS_CHECK(z_origin, 0, volume_zsize);

1199    int multistart_guesses[MAX_MULTISTART_STARTS];

1201    int nGuesses =
          findLocalMinimas(multistart_guesses, plane_eqs, radius,
              voxel, volume_xspacing, volume_yspacing, volume_zspacing
              , in_xspacing, in_yspacing, plane_matrices, mask,
              in_xsize, in_ysize);
1203
      #ifdef DEBUG
1205    for(int i = 0; i < nGuesses; i++)
        {
1207      DEBUG_PRINTF("Multistart %d: idx %d dist %f\n",i,
                        multistart_guesses[i],
1209                    fabs(dot(voxel, plane_eqs[multistart_guesses[i
                            ]])));
        }
1211  #endif

1213    int2 close_planes_ret;
        // Iterate over the axes such that the the
1215    // next voxel is always a neighbour of the previous voxel
        for(int xoffset = 0; xoffset < CUBE_SIZE; xoffset++)
1217    {
          int x = x_origin + xoffset;
1219      if(x >= volume_xsize) break;
          BOUNDS_CHECK(x, 0, volume_xsize);
1221
          int ystart, yend, ydir;
1223      if(xoffset % 2)
          {
1225        ystart = CUBE_SIZE-1;
            yend = -1;
1227        ydir = -1;
          }
1229      else {
            ystart = 0;
1231        yend = CUBE_SIZE;
            ydir = 1;
1233      }
```

```
    for(int yoffset = ystart; yoffset != yend ; yoffset+=ydir)
    {
      int y = y_origin + yoffset;
      if(y >= volume_ysize) continue;
      BOUNDS_CHECK(y, 0, volume_ysize);

      int zstart, zend, zdir;
      if(yoffset % 2 && xoffset % 2)
      {
        zstart = CUBE_SIZE-1;
        zend = -1;
        zdir = -1;
      }
      else {
        zstart = 0;
        zend = CUBE_SIZE;
        zdir = 1;
      }
      for(int zoffset = zstart; zoffset != zend ; zoffset+=zdir)
      {
        int z = z_origin + zoffset;
        if(z >= volume_zsize) continue;
        BOUNDS_CHECK(z, 0, volume_zsize);
        BOUNDS_CHECK(x, 0, volume_xsize);
        BOUNDS_CHECK(y, 0, volume_ysize);
        voxel.x = x * volume_xspacing;
        voxel.y = y * volume_yspacing;
        voxel.z = z * volume_zspacing;

        // Find all planes closer than radius
        close_planes_ret =
          FIND_CLOSE_PLANES(close_planes, plane_eqs,
              plane_matrices, voxel, radius, multistart_guesses,
               nGuesses, mask, in_xsize, in_ysize, in_xspacing,
              in_yspacing);

        n_close_planes = close_planes_ret.x;

        // Call appropriate method to determine pixel value
        VOXEL(out_volume,x,y,z) =
          PERFORM_INTERPOLATION(close_planes, n_close_planes,
              plane_matrices, plane_eqs, in_bscans, in_xsize,
              in_ysize, in_xspacing, in_yspacing, mask, voxel);
      }
    }
  }
}
```

code/kernels.ocl