Problem description

3D Ultrasound is often generated by reconstruction of freehand input, generating 3D volumes from 2D ultrasound images with tracked positions. Ultrasound imaging is a fast and flexible modality, allowing it to be used in settings where other imaging modalities would be inadequate.

The FAST framework is designed to make cross-platform and heterogeneous implementation easy and efficient. There are no previous implementations of 3D Ultrasound reconstruction in FAST.

Development of reconstruction algorithms have been performed for decades, and speed performance always increasing with faster hardware and more parallel development, for instance on GPUs.

The goal of this thesis is to implement a reconstruction algorithm which:

- Is the first to integrate into FAST, discovering and trespassing its caveats
- Is previously unpublished, and will be well-described in this thesis.
- Creates reconstructions of good quality
- Performs at top-end reconstruction speeds
- Can be extended to fit in a real-time setting

With the GPU implementation, we hope to see good quality at top-end reconstruction speeds.

Abstract

Ultrasound imaging is a versatile, portable, and low cost medical imaging modality. It produces real-time data from a local area in the scanned person, or object, useful in use cases such as intra-operative imaging. Freehand 3D ultrasound reconstruction is a technique used to convert sets of 2D ultrasound images into a 3D volume. It is prefered over direct 3D acquisition due to increased resolution and flexibility, and lower cost. However, freehand 3D ultrasound reconstruction requires tracking, which will contain inaccuracies.

Reconstruction algorithms are supposed to approximate the volume, given the overlap and gaps between input data, while taking position uncertainties into account. There are many existing solutions, from basic one-to-one solutions like Pixel Nearest Neighbor (PNN) and Voxel Nearest Neighbour (VNN), to compounding distance weighted solutions like Varying Gaussian Distance Weighting (VGDW). With the increasing power of Graphical Processing Unit (GPU) calculations, reconstruction can run faster, allowing more powerful methods to reach demands for new situations. In this thesis we propose a previously unpublished *hybrid* approach based on pixel-based methods. For each input frame it finds the voxels within relevant distance, and for each of these voxels it accumulates the most relevant data from the current input frame. Voxel focus in accumulation, allows for a highly parallel, and computationally efficient method that can run at high performance. Variations in input distance are easily adjusted for by a flexible distance weighting function.

Multiple configurations were tested, to improve the visual quality or performance properties of the algorithm. Visual evaluation by a group of ultrasound technologists proved that the hybrid approach can score at least as good as the VGDW, as the evaluators consensually scored both *hybrid* configurations ahead of VGDW. Running on a Nvidia GTX 970 GPU high-end runtime performance was achieved, with sub-second reconstruction time on volumes with 32 million voxels, and very good scaling with bigger volumes. An alpha-blending compounding method has been implemented, accumulating frames at a rate of 2000 frames per second, with no need for normalization. Hybrid accumulation performance is on line with PNN accumulation, yet without the need for a hole-filling step it severely outperforms PNN on overall performance. The hybrid algorithm scales particularly well with volume size, and is able to adapt well to varying distance between input frames. Additional improvements have been looked into for further work, as well as exploring the real-time solutions for this algorithm, or others.

Acknowledgements

A thesis can be a big task to get into, and many people have contributed to make it happen. 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. Multiple people connected to the SINTEF Department of Medical Technology have also been involved. I would like to thank these two groups for all the tools and assistance available for me during my thesis.

I would like to thank my supervisor, Dr. Frank Lindseth (IDI, SINTEF), for invaluable feedback. He provided lots of knowledge that helped me understand the ultrasound imaging modality, and its relations to intra-operative use. The lead developer of the FAST framework, Erik Smistad (SINTEF), has done a big contribution to this work, primarily by helping me setting up FAST, and making good implementation choices for the framework, but also by assisting me with valuable insight on my thesis and 3D Ultrasound reconstruction, together with Daniel Høyer Iversen (SINTEF).

Also, I would like to thank Torgrim Lie (SINTEF) for assisting with his important knowledge of the algorithm, and its previous implementation. He provided essential feedback for the new implementation.

Finally, I would like to thank my friends, and family, who stood by as moral support until the very end.

Trondheim, Oktober 2016

Ruben Håskjold Fagerli

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

Hybrid	Hybrid reconstruction method, a hybrid-focus 3D ultrasound reconstruction method introduced in this thesis
dv	Parameter: voxel-distance. Used to represent the wanted spacing between neighbouring voxels in the output volume of the hybrid approach
df	Half width. Used to represent the current impact distance of the input pixel
Rmax	Parameter: max distance. Used to represent the maximum possible value of df, and thereby the maximum possible impact distance of the input pixel
PNN	Pixel Nearest Neighbour, a pixel-focused 3D ultrasound reconstruction method
VNN	Voxel Nearest Neighbour, a voxel-focused 3D ultrasound reconstruction method
VGDW	Varying Gaussian Distance Weighting, a voxel-focused 3D ultrasound reconstruction method with varying weighting by local variance
B-Scan	2D ultrasound image
US	Ultrasound
CT	Computed Tomography
MRI	Magnetic Resonance Imaging
FAST	Framework for Heterogeneous Medical Image Computing and Visualization, a framework for heterogeneous computing used in this thesis
CPU	Central Processing Unit
GPU	Graphics Processing Unit
OpenCL	Open Computing Language, a framework for parallel computing on heterogeneous systems maintained by the Khronos Group
PBM	Pixel-based method. Type of 3D ultrasound reconstruction focused on input pixels.
VBM	Voxel-based method. Type of 3D ultrasound reconstruction focused on output voxels.
FBM	Function-based method. Type of 3D ultrasound reconstruction with functions.

Chapter 1: Introduction

3D Ultrasound reconstruction computes a 3D volume representing all the data available with a set of 2D ultrasound images, including the pixel position, and intensity value. This reconstruction allows further segmentation and generating slices in arbitrary planes, empowering ultrasound to do task otherwise reserved for CT and MR imaging. 2D ultrasound imaging offer advantages over these two with real-time acquisition, and no radiation. To preserve the real-time speed in 3D a reconstruction implementation has to be efficient, and utilize the incremental nature of the input efficiently. This would allow the operator to continue using the ultrasound machine as when scanning normal 2D ultrasound, keeping the flexibility of 3D ultrasound at its prime. Rapid, or real-time, reconstruction enables the operator to evaluate the results right after, or during, scanning, to quickly decide if acquisition was appropriate, or if rescanning is necessary. Many times bad acquisition can produce more mistakes than any reconstruction can adjust for.

Reconstruction quality is also important, and algorithms will need to handle inaccuracies and overlapping data while preserving detail. 3D ultrasound reconstructions have no definite gold standard as they are created by approximations of the input, through smoothing and compounding effects, as complete accuracy is impossible. In the end it will matter more what differences the operator can perceive. A balance between visual quality and reconstruction swiftness should be found, to produce results good enough for the application yet swift enough to reach its constraints.

This thesis will use the massively parallel nature of GPU processing, to leverage the constraints to get a solution more on the quick side without losing much quality. The algorithm will hasten reconstruction with highly parallel and numerically simple calculations, while preserving quality with weight based accumulation methods. Flexibility will be ensured with adaptable maximum distances depending on input local densities.

1.1 Goals

The goals of this thesis are to:

1. Implement the previously unpublished algorithm in FAST, as the first 3D ultrasound reconstruction integrated into the framework.
2. Implement a reconstruction algorithm with quality result, while reaching for top-end performance.
3. Evaluate algorithm options to increase quality, or achieving performance capable of real-time results.

1.2 Contributions

The contributions of this thesis consist of:

- A thorough description of an previously unpublished hybrid algorithm, exploring what improvements this offer over other algorithms.

- Implementation of the first reconstruction algorithm in FAST, making it easier to implement new algorithms. Implementing a simple pipeline that allows both pixel-, and voxel-based methods.
- A quick accumulation technique capable of real-time accumulation. Ideas for how accumulation with real-time visualization can be implemented.

1.3 Thesis outline

The thesis chapters will be outlined below.

Chapter 2: Background presents the underlying information necessary to implement, or fully understand the reconstruction concept, as a background for the remainder of the thesis. It will introduce Ultrasound imaging, and 3D Ultrasound reconstructions, explaining the basics of the modality, and explaining existing methods of reconstruction. Basic mathematical formulas are explained, and the FAST and OpenCL are introduced.

Chapter 3: Method introduces the previously unpublished hybrid approach, and the implementations that have been done with this thesis. The algorithm is described in detail, to avoid confusion with existing methods. Parameters and OpenCL GPU implementation choices are described. Finally the evaluation processes are described.

Chapter 4: Results presents the results of the visual quality and speed performance evaluations. It includes score results from the visual evaluation, together with the evaluated images. Performance results are displayed with tables and graphs for multiple parts of the runtime and adjustments for image count and volume size.

Chapter 5: Discussion looks at the findings of Chapter 4, discussing the results of the visual evaluation, as well as the performance evaluation.

Chapter 6: Conclusions summarizes this thesis, comparing the end results to the initial goals.

Chapter 7: Further Work presents some ideas that have been discovered during the work on this thesis. These ideas are found hopeful for further work.

Chapter 2: Background

This chapter will introduce some of the background theory that serves as a basis for this thesis. Section 2.1 will introduce medical ultrasound imaging, and explain how it is created. Section 2.2 will elaborate on the topic of Freehand 3D ultrasound reconstruction, which is the main focus in this thesis. The FAST framework will be presented in Section 2.3, while in Section 2.4 the theory behind OpenCL computation will be explained. Finally, Section 2.5 describes the mathematical theory used for in this thesis, from geometry, to interpolation and weighting.

2.1 Ultrasound Imaging

This section will introduce the ultrasound imaging modality, and explain how ultrasound images are used and created. Section 2.1.1 will introduce ultrasound and compare it to CT and MRI scans. Section 2.1.2 explains how ultrasound imaging works and how the image is built up. Section 2.1.3 will talk about artefacts and problem areas that may arise, while Section 2.1.4 will introduce the concept of 3D ultrasound. The work of Sverre Holm[6] have been a great base for understanding ultrasound physics and providing reasonable numbers to describe medical ultrasound imaging.

2.1.1 Ultrasound Intro

Ultrasound (US) is one of the three major medical imaging modalities. Commonly known as a tool to check the fetus in the womb of the mother, it also has a lot of other use cases, for instance cardiology and imaging organs in the abdominal region. It can sometimes replace the use of a Computed tomography (CT) scan or Magnetic resonance imaging (MRI). Ultrasound imaging might not be as accurate as these other two, but it is a lot more versatile. With its low cost and portable size it is a lot more accessible than the other two. Fast results make it easy to use, and the lack of radiation makes it a safe choice. However it is dependant on fluid contact for the sound waves to pass though, and certain artefacts may occur if air bubbles or bone exists. US imaging also have to make a trade-off between spatial resolution and penetration depth, something that often leaves the modality best at portraying objects relatively close to the surface.

2.1.2 Ultrasound image creation

The ultrasound probe is built up from multiple piezo-elements, small crystals that change size and shape depending on the applied voltage. The voltage is alternated by a AC current, which in turn creates sound waves into neighbouring materials. If the voltage is alternated at a frequency above 20 kHz, it is technically a ultrasound wave. In practice a frequency between 2 and 10 MHz is commonly used in medical ultrasound.

The ultrasound sound waves are reflected in the target material, returning the waves back to the piezo crystal. The crystal has the ability to receive this wave and turn it back into an AC current, allowing us to essentially receive feedback in the pulse-echo format. Whenever the sound wave transitions from one tissue to the next the wave is refracted, and some of the signal is reflected. The bigger the difference between the tissues, the higher the reflection. Moving into bone or air causes almost total reflection, strongly diminishing the results behind it. The delay and power of the returned reflections are used to calculate the distance and intensity.

Multiple piezo-elements are lined up to create an array of elements, making a 2D ultrasound transducer or 2D probe. Each piezo-element reads a 1D scan line of data, and together they can form a 2D image. These 2D images, B-scans, are captured in B-mode, brightness mode, and these images will be the input data of this thesis.

2.1.3 3D ultrasound

Ultrasound imaging can create 3D volumes, and for doing so there exist multiple approaches. For instance, transducers with a 2D array of piezo-elements can directly capture a 3D volume. However the probe footprint can be large, and the resolution is subpar to its 1D array counterparts. Another approach is to use a 1D array of piezo-elements mounted on a motorized unit, that will move the array in a simple, repetitive translation, tilt, or rotation. This will allow the 1D array to cover a 3D volume over a limited time period. As this volume is not acquired in an instant, the scans at each timeframe still need to be run through some kind of reconstruction algorithm to adjust for any movement. Both of these approaches suffer due to the fact that they require new specialized equipment, and the size of the probe might limit its use cases.

Freehand 3D ultrasound reconstruction uses any 1D array transducer and uses tracking data to position the B-scans in world space and calculate values for each output voxel depending on the position of each input image. Freehand 3D ultrasound has the advantage of using existing probes, with a tracking device, allowing for a cost efficient, diverse, and flexible execution. Freehand 3D ultrasound will be discussed more in Section 2.2.

2.2 Freehand 3D Ultrasound reconstruction

This section will follow up on Section 2.1 and dive into the topic of freehand 3D ultrasound. In Section 2.2.1 the topic of freehand 3D ultrasound will be introduced, requiring the tracking described in Section 2.2.2. Section 2.2.3 will showcase common reconstruction algorithms with their most important traits. Section 2.2.4 explains some of the general optimization- and parallelization-traits of the main reconstruction types. Section 2.2.5 talks about existing real-time solutions, and discusses why it matters. Section 2.2 is based on the work of Solberg et al.[13], as well as a few other sources[4][9][10].

2.2.1 Freehand 3D ultrasound reconstruction intro

Freehand 3D Ultrasound reconstruction uses a series of 2D B-scan images to construct a 3D volume. The probe is tracked to determine the position of the images. Tracking will be discussed in Section 2.2.2. A freehand approach can utilize any existing 2D ultrasound probes, making it adaptable to any clinical situation.

With current computation power available in desktop computers there has opened for new opportunities in 3D ultrasound reconstruction. New complex algorithms are becoming usable[10], and efficient, parallelizable algorithms are available in real-time[2][4].

2.2.2 Freehand tracking

Tracking is done to determine the spatial position of an object over time. In 3D ultrasound, tracking is used to find the position of the probe and its data, relative to the patient. This is commonly achieved by visual tracking system, tracking markers placed on, or next to, the patient, and on the ultrasound probe.

The probe data position has to be spatially calibrated relative to the probe markers, while the probe output B-scan timestamps has to be temporally calibrated to the timestamps of the tracking system. When properly calibrated the tracking system will be able to relate a spatial position to each B-scan image, giving us a transformation from image space to world space for each pixel. Getting the calibration perfect is difficult, and the tracking system may introduce small errors.

2.2.3 Algorithms for 3D reconstruction

There are numerous ways to approach the 3D reconstruction problem, each with a different advantage and complexity. Selecting an algorithm with the right accuracy, while maintaining an appropriate computation time, is essential for making the best out of the use of 3D ultrasound. A thorough summary has been done by a team at SINTEF and NTNU here in Trondheim[13]. They define three groups of reconstruction algorithms: Voxel-based methods (VBM), Pixel-based methods (PBM), and Function-based methods (FBM). Their classification of algorithms are used throughout this thesis. I will present some of the key algorithms presented by Solberg et al. in addition to some newer approaches.

2.2.3.1 VBM: Voxel Nearest Neighbour (VNN) and interpolation (VBMI)

A voxel-based method like the VNN, iterates over output voxels and fetches data from image planes nearby. The VNN, which is the most basic of the voxel-based methods, uses the value of the closest pixel in the input planes within a certain distance. Other implementations use 1D interpolation of beam data or 2D interpolation of the B-Scan image, to decide a more accurate representation of the voxel value. More advanced interpolations, using data from multiple input frames, are common.

2.2.3.2 VBM: Varying Gaussian Distance Weighted (VGDW)

VGDW[9] is a distance weighted (DW) method, based on the Adaptive Gaussian Distance Weighted (AGDW) method[7]. It is a voxel-based method, averaging one pixel from each image plane within a given radius, with weights depending primarily on the distance. VGDW is different in that it adjusts for local variance between input images, giving a higher smoothing at low variance than at high variance. High variance usually reflects details, such as edges, which VGDW attempts to leave intact. VGDW produces high quality reconstructions in a reasonable reconstruction time, and has been used as a reference volume in this thesis.

2.2.3.3 PBM: Pixel Nearest Neighbour (PNN)

The PNN algorithm consists of two steps: distribution step (DS), and hole-filling step (HFS). The PNN is the most basic of the pixel-based methods, filling the closest voxel from each input pixel. This process is very fast and efficient, but unless the voxel size is significantly larger than the pixel size holes will occur. A hole-filling step is performed to fill the gap voxels between the distributed slices, to ensure a continuous volume. HFS usually calculate an average of neighbours in a 3x3x3 neighbourhood, that can possibly progress to larger sizes when necessary.

2.2.3.4 PBM: Pixel trilinear interpolation (PTL)

As one way of applying a 3D kernel in the distribution step, PTL applies a 2x2x2 kernel on voxel points around the pixel location, effectively interpolating it in three directions. Being applied to two voxels in thickness, the need for a hole-filling step is highly reduced as long as the volume resolution is not set too high and the operator manage to move the probe slowly and evenly[4].

2.2.3.5 PBM: Using a 3D kernel in the distribution step

3D kernels can be applied in many shapes and sizes for the distribution of pixel-based methods. A square kernel can be increased, up from the 2x2x2 of the PTL, to increase accuracy at the cost of highly escalating computational cost. Where a 2x2x2 kernel operates on 8 voxels a 5x5x5 kernel will operate on 125 voxels, for each input pixel. Circular or elliptical kernels can also be applied for greater accuracy. These methods weigh the input by a distance weighted (DW) scheme, decreasing its influence the further away it gets. These weights, be it inverse distance or gaussian, will be discussed in Section 2.5.5.

2.2.3.6 FBM: Radial basis function interpolation (RBF)

Function-based methods, like the RBF, determine multiple functions so that each input pixel is represented at least once. RBF can interpolate and approximate, and can work at different tension levels, allowing it to easily scale with increased volume sizes. RBF is known to create high quality reconstructions. Function-based methods are extremely computationally expensive, but work have been done by Rohling et al.[10] to make it a little bit faster. The runtime is however still in the order of hours, as opposed to seconds for many other methods[13].

2.2.4 Optimizations and parallelization

Optimization of the algorithms are important to ensure efficient execution. While RBF ensure efficiency by splitting up the problem size to more manageable blocks, the VBMs try to split up the amount of input images it has to search through in order to find the closest frame[9]. Optimizing for smaller work groups and quicker search can make a huge difference in how fast these problems are solved, as they have bad computation scaling properties.

Parallelization is also substantial for achieving a quick running time, as run times can be divided by up towards the number of parallel units, essential to reach the next level of performance. Nowadays with GPU processing, parallel computation is more available than ever before. Voxel-based methods can often separate the work on each voxel, allowing it to queue millions of parallel tasks at once. An overlapping calculation is finding the closest frames, which can be shared within a neighbourhood cube[9]. Pixel-based methods can also be parallelized, first of all each input image can be processed separately. Also, each pixel in each input image can also be processed independently.

2.2.5 Real-time 3D ultrasound reconstruction

Real-time 3D reconstruction is the process of incrementally acquiring input, reconstructing, and visualizing the volume, for each input frame as they are captured. This is quite useful as it allows the operator to get visual feedback during acquisition, guiding position and displaying artefacts, allowing him to rescan where needed.

Gobbi & Peters[4], and Dai. et al.[2] described PNN- and PTL-based real-time reconstruction and visualization, with 3-slices view, and volume visualization, respectively. Their work gave inspiration for much of the real-time reconstruction work in this thesis.

Real-time reconstruction requires multiple steps to be executed in fast succession. In addition to minimizing the runtime of each step, the latency delay from the time of acquisition to rendering should be minimized to ensure a real-time response in visualization. To ensure this, storage is prepared on device before starting incremental reconstruction and visualization described below.

Firstly, the digital images and tracking data should be acquired and matched on a stack, ready to be processed.

Secondly, the image is accumulated into the reconstruction volume by the reconstruction algorithm. This is called incremental reconstruction. The input-based methods pixel nearest neighbor (PNN) and pixel trilinear interpolation (PTL), are popular methods for real-time reconstruction. They utilize some form of blending to require no normalization, while achieving very low runtimes. An algorithm that requires no hole-filling is prefered, as this can be a difficult step to include incrementally.

Finally, the reconstruction volume is incrementally rendered and visualized. Normally a relatively time-consuming process, but it can be optimized to avoid processing the full volume. Gobbi & Peters[4] implemented an adjustable 3-slice view, requiring only processing of the voxels in each plane. Dai et al.[2] rendered, projected and visualized the volume incrementally, by only updating the sub-sections affected by the incremental reconstruction.

With 3-slice view, a reconstruction was achieved at up towards 30 frames/s[4], while the newer reconstruction with incremental volume rendering was performed up towards 58 frames/s[2]. Dai et al. performed incremental reconstruction at a rate of 90 frames/s, and at half of that rate with a fan scan.

In both of these papers, the volume was seemingly statically predefined, requiring no further adjustments of the volume as input is acquired. To maximize the view compared to the accumulated data, an incrementally adjusting step should be added, that can extend or resize the volume. They can also require a hole-filling step at 16M volume, that further adds to the runtime, although it can be performed as a final postprocessing step, after all incremental reconstruction. In practice, any algorithm can be run to create a final reconstruction after acquisition.

With some restrictions and simplifications, it is already possible to achieve real-time reconstruction and visualization. The next step will be to remove these restrictions, creating implementations that can run with any type of probe movement, bigger volumes, and creating superior results. This thesis presents a reconstruction that can form the basis to such a solution.

2.3 FAST background

This section will introduce FAST (Framework for Heterogeneous Medical Image Computing and Visualization)[11], with the framework design and pipeline. FAST was developed by Erik Smistad as part of his doctoral degree and is available as an open-source project available online[12].

Section 2.3.1 gives a intro to FAST, while Section 2.3.2 explains its framework design. Section 2.3.3 introduces the FAST pipeline. Finally, Section 2.3.4 specifies some points that can affect implementation of 3D ultrasound algorithms.

2.3.1 FAST intro

FAST is a cross-platform framework with a goal of easy and efficient processing and visualization of medical images. It is developed with heterogeneous systems in mind, systems with multi-core CPUs as well as GPUs. With increasing number of processing units the need for parallelizable code increases, to utilize the computational power available. This is important because many algorithms are striving to achieve real-time results. Good parallelization can give speedups of an order of magnitude or more, which can be the difference between an algorithm which is usable in a practical setting and one that is not.

2.3.2 FAST Framework Design

Programming directly on hardware is a daunting task, one that would require a lot of work for even simple problems. On top of that it will have to be reimplemented if trying to use another piece of hardware. To get around this problem we use drivers, libraries and frameworks, generalizing the underlying opportunities into simpler commands that works across different configurations. Drivers provides the functionality of the hardware to the computer in a standardized way, while frameworks and libraries further simplifying the situation of the programmer by bringing increasingly stronger and versatile tools available through simple commands, giving us high-level programming. FAST is such a framework, and it is built up from many frameworks and libraries to make its development easier and more versatile.

The FAST framework is built up of many layers, each providing operations to the layers above it. These are shown in the image below. Roughly these can be divided into 5 layers, listed from bottom up :

- Hardware
- Drivers
- Libraries
- FAST Framework Core
- Applications

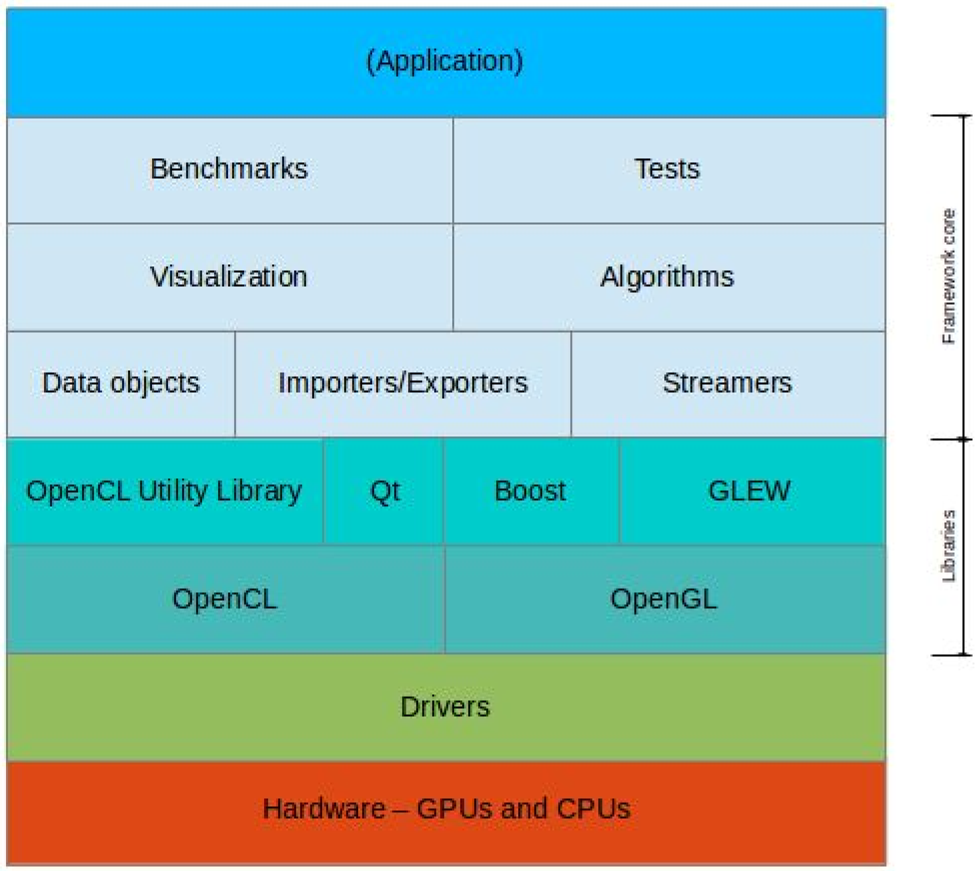


Figure 2.1: FAST framework layers. Applications run on the top of the framework core, with multiple submodules. The framework utilizes many libraries to simplify underlying processes. Finally this runs on the physical hardware with their drivers.

2.3.2.1 Libraries

The OpenCL and and OpenGL are the fundamental libraries for cross-platform programming, for parallel programming and visualization respectively. OpenCL will be talked about more in detail in Section 2.4, while visualization and OpenGL have not been central in this project. Other libraries include Qt, GLEW, Eigen, Boost and OpenIGTLink.

2.3.2.2 FAST Framework Core

The core defines the FAST framework. The basis for all processing are the Data objects, enabling synchronized processing of images and meshes, static and dynamic on many heterogeneous devices. Streamers are objects that allow streaming of data. Importers and Exporters enable us to move objects in and out of the framework. The algorithms, like the reconstruction in this thesis, process data objects from importers, or streamers, to produce results for the next algorithm, visualization, or exporters.

2.3.2.3 Applications
On top of this are the applications, which are very simple to implement, given the high-level functionality provided by the FAST algorithms and its other tools. The user just has to initialize the algorithms and importers and link them together to form a execution pipeline.

2.3.3 FAST Execution Pipeline

The structure of FAST forces applications to apply the pipes and filters design pattern, where for instance Algorithms and Importers/Exporters are filters and the user sets the connections, named pipes, between these filters. This allows for simple reuse and reordering of filters. Pipes and filters is in general a highly parallelizable design pattern, where each filter can be run in parallel if updating regularly. FAST uses a demand-driven execution pipeline, inspired by the frameworks ITK and VTK.

2.3.4 FAST Image loading and volume rendering

When loading input images into a FAST pipeline on a device with a GPU unit available FAST will automatically load these images onto the GPU, so it does not have to be performed during execution. This might increase loading times significantly, but will allow further processing to execute smoothly.

FAST was during the implementation time of this thesis without a volume renderer working with the loading methods required by 3D ultrasound reconstruction methods.

2.4 OpenCL background

This section will introduce the OpenCL standard[8] and how it is built up through kernels and memory layout, with a focus towards GPU layouts. The videos of David Gohara[5] have been a big help for understanding this topic. His foils and example code are available at his website[3]. These videos are strongly recommend, especially episode 2 through 6, for those new to OpenCL.

Section 2.4.1 gives an intro to OpenCL, while Section 2.4.2 discuss kernels and work items. Section 2.4.3 takes a deep dive into OpenCL memory, how it is grouped, and optimized. Finally, Section 2.4.4 how OpenCL kernels can read, and write, to the same image with different OpenCL versions.

2.4.1 OpenCL Intro

OpenCL allows writing programs that will execute on heterogeneous systems consisting of compute devices like CPUs and GPUs. It is made for parallel computing using task-based and data-based parallelism. OpenCL specifies a programing language and API to execute programs on compute devices. It is based on the C99 version of the C programming language, meaning OpenCL code is built up similar to C code, with native OpenCL methods as additions.

It is up for the hardware-providers to implement these specifications for their hardware. Extensions of OpenCL functionality also needs to be implemented by the providers for these to be available for their hardware. All the big providers, like AMD, Intel and Nvidia, have implemented OpenCL on their devices, meaning that they should all have the basic OpenCL functionality. This distinguishes OpenCL from for instance CUDA, allowing it to run on all hardware and not just Nvidia CUDA GPUs. The same OpenCL code can be set to run on a CPU, a Nvidia GPU or an AMD GPU.

The OpenCL host APIs are defined for C and C++, but third-party APIs for Python, Java and .NET are also available. In addition to the C/C++ API library, a OpenCL C compiler is necessary for the compute devices.

2.4.2 OpenCL kernels and work items

OpenCL kernels are the programmed functions destined to run on the compute device. Kernels are functions written in a .cl file, and the kernel can be run in multiple instances in parallel.

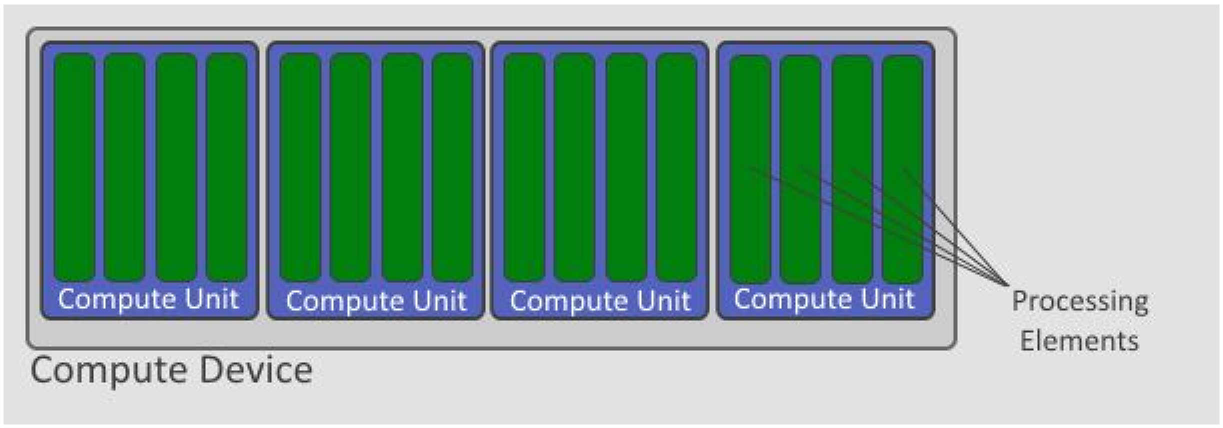


Figure 2.2: OpenCL - Compute Units. The compute device consists of multiple compute units, each of which contains multiple processing elements (PE). PEs runs each work thread, and the PEs within the same Compute Unit can cooperate.

Figure 2.2 shows a typical compute device unit breakdown:

- Compute Device: The device consists of multiple compute units.
- Compute Unit: This is the cores that consists of multiple Processing Elements.
- Processing Element (PE): This is the threads that execute the OpenCL work items.

It is however important to note that the concept of cores are very different for CPUs and GPUs. The kernel is run on each processing element(PEs) with each their own work item. A work item is usually a single element in the data set, like one element in an array or a pixel in an image. In the simplest programs the kernel will load one input element, process it and store the corresponding output element. This allows for massive parallelization, potentially letting all elements being processed simultaneously by different PEs.

OpenCL kernels uses work item focused programming. It might seem like a lot of overhead doing this work for each pixel, but with smart loading and writing this can lead to big speedups due to the massive parallelization, especially in GPU devices. In the next subsection I will talk about how this can be achieved.

2.4.3 OpenCL Memory

OpenCL defines multiple types of memory, with each their different benefits and limitations. This section focus on the GPU.

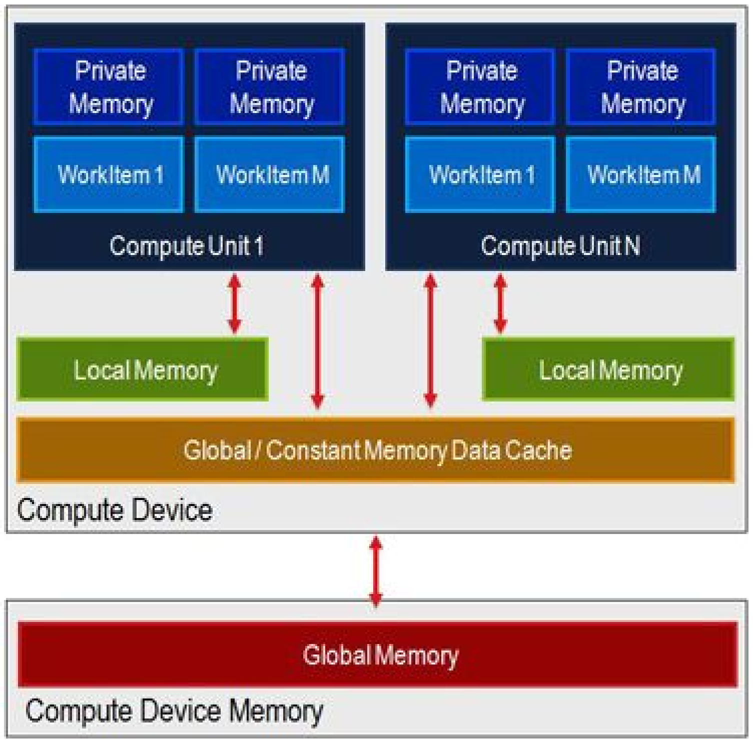


Figure 2.3: OpenCL - memory structure. There are multiple levels of memory, from the global memory shared across the whole device to multiple compute device specific levels of memory, where the later is much smaller, but also significantly faster.

We have 4 types of memory on GPUs as shown in the image above:

- Global: The main memory of the GPU. Slow access, but capacity order of MBs or more.
- Constant: Constant memory available for all units on the compute device or thread cluster. Typically 64KB, shared among all units. Slightly faster than global memory.
- Local: The most important kind of memory for OpenCL optimization. Order of 16KB size. Shared among all PEs, or work item, in a Compute Unit, allowing for access close to registry speed if used properly.
- Private: Small and fast memory private for each work item.

The key to OpenCL optimization is to load data that is to be reused within the work items of a Compute Unit into local memory, instead of taking the costly trip to global memory and back. To specify the work items sharing the same local memory we define a work group. These consist of a set of work items that will share data. A work group can for instance be neighbouring elements in an array, or pixels of a region of an image. The maximum number of work items per work group is however limited, usually to the order of 512 or 1024 elements, which limits our work groups to the likes of 16x16 or 32x32 in a 2D setting.

So we have two main limiting numbers:

1. The maximum number of work items that a work group can consist of.
2. The local memory storage capacity.

Another important factor to account for is how the memory is accessed. The key concepts here are memory banks, warps and half-warps. At the end I will also talk about the concept of efficient coalesced loading of data into local memory.

Local memory is divided into memory banks, typically 16 of them. Each bank can access one element at the time, otherwise it will spark a bank conflict and the access has to be serialized. So we want all the simultaneously running elements to access different banks. This is where warps come in, or more specifically half-warps. A half-warp is the number of work items that are executed at the same atomic time. It is usually of a size of 16 or half of the total warp size.

One element is typically at the size 32 bit, which equals to 4 Byte. All of the 16 memory banks store 4 Byte, until 64 Byte is reached, then it starts over from the first bank again. This means that the first bank will store among other Byte 0, Byte 64, Byte 128 and so on. So what we have to avoid is that work items in the same half-warp will access the Bytes corresponding to the same memory bank. If this is a problem, the solution is usually to add padding of one between each row or groups of memory. This padding will change the indexes and might fix this issue. There is one exception that will not cause a bank conflict, and that is if we have a broadcast. A broadcast occurs when all work items in a half-warp access the same address. Then the value will only be read once and broadcasted to all work items.

It is also important how memory is loaded into local memory from global memory. We can do one call to global memory for each element we load into local memory. This is the case when we do:

- Misaligned load: the 16 work items of the half-warp does not fetch the elements of a 64 Byte block, but starts somewhere in the middle of one and crosses over to the next one.
- Permuted load: Work items will not load elements of the block in order, but work item 1 will load element 3, while work items 2 and 3 will load element 1 and 2 for instance.
- Sparse load: When not all work items participate in loading.
- Sequential load: One work item is set to load all elements into local memory.

All of these load the data just fine, but will not utilize optimized loading from global memory. Coalesced loading occurs when all work items of a half-warp simultaneously load data aligned with a 64 Byte block in a non permuted way. This greatly reduces the overhead of loading data into local memory.

2.4.4 OpenCL Image Read-Writes and atomic operations

When reading and writing data to the same image array, it is necessary to have both reading and writing rights to the image. OpenCL does not acquire this until OpenCL version 2.0, and Nvidia graphics cards are limited to OpenCL version 1.2. This requires finding some workarounds to be able to read and write to the same buffer at the same time, from multiple sources. One workaround is to use a buffer of integer types, and accumulate them by atomic operations. Another is to use a separate semaphore array, and lock the indexes currently in use. Both methods are used in this thesis, where semaphores are required by the alpha-blending approach, and other solutions use the simpler atomic operations.

2.5 Mathematical theory

This section will present some of the mathematical theories that work at the core of a 3D Ultrasound Reconstruction algorithm. These are grouped in three main types: geometry, acquisition, and accumulation.

The geometry sections will explain the basic geometrical mathematics required for point and plane representation and distance. Section 2.5.1 starts off explaining how planes can be represented in 3D space, Section 2.5.2 introduces the general point-to-point distance along a vector, which can be used to find the point-to-plane distance on certain conditions. Section 2.5.3 explains how to calculate the point-to-plane distance with less restrictions.

The acquisition Section 2.5.4 demonstrates how data can be approximated at any given point in the input frame, by using bilinear interpolation to fetch data from neighbouring pixels.

The accumulation sections will explain how to handle multiple pixel values that affect the same output voxel. Section 2.5.5 introduces weighting, as a method of giving emphasis to more relevant acquisitions data, and presents a couple of weighting schemes. Section 2.5.6 shows how these weights can be added together to create an accurate representation of all input.

2.5.1 Geometry: Describing a plane in 3D space

It is crucial to have an appropriate description of the plane when calculating its position relative to other objects of interest. In mathematical terms a plane is described as follows:

$$ax + by + cz + d = 0$$

Where *x*, *y*, and *z* are variable 3D space coordinates, and *a*, *b*, *c*, and *d* are constants defining the plane. Any set of variables that make the left side of the equation zero, fulfills the equation, and will reside in the plane. To calculate the constants in practice it is possible to use the plane normal for the values of the *a*, *b*, and *c* constants, where the normal vector equals (*a, b, c*). The *d* value of the plane can be calculated from the normal vector, and a point residing in the plane. In this case the plane can be described as:

$$normal_x \cdot x + normal_y \cdot y + normal_z \cdot z + d = 0$$

2.5.2 Geometry: Point-to-point along vector

The distance between two points, P_A and P_B, along a vector, *N*, can be expressed as:

$$movement_{BA} = (\vec{P_A} - \vec{P_B})\ dot\ \vec{N}$$

$$distance = abs(movement_{BA})$$

Where $movement_{BA}$ is the distance moved from P_B to P_A in positive or negative direction along the vector, and *abs* a function to find absolute value.

If the vector, *N*, is the normal vector to a plane in which P_A resides, the abovementioned equation will find the distance from P_B to the plane of P_A, as shown in Figure 2.4.

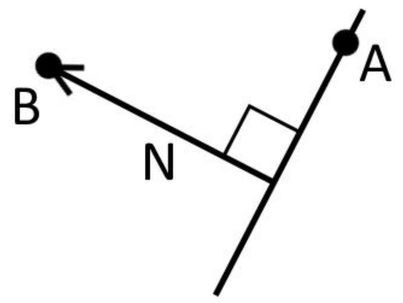


Figure 2.4: Point-to-plane distance along plane normal. Given two points, A and B, where A is residing in a plane whose normal vector is N. The point-to-point distance formula can be used to find the point-to-plane distance between B and the plane of A, as the normal is by definition perpendicular to the plane.

2.5.3 Geometry: Point-to-plane distance and intersection along vector

This method can unlike the method in Section 2.5.2 utilize any relation between the vector and the plane normal, and does not require them to be equal, as seen in Figure 2.5 below.

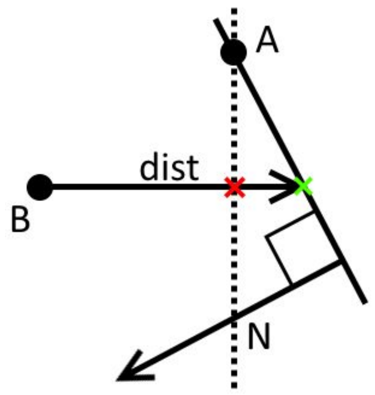


Figure 2.5: Point-to-plane distance along non-normal vector. Displaying the calculation of the distance from point B to the plane of point A. Distance is measured along the horizontal vector along which distance, dist, is shown. The dotted line and red cross indicates the intersection point that would have been found with the standard point-to-plane equation of Section 2.5.2, assuming the vector to be the plane normal. The green cross indicates the actual intersection point, to which the distance is measured.

The problem can be reformed to the problem of finding the intersection between a line and a plane. The formula for a line is described below, where *P* is any point on the line, *L* is the directional vector of the line, and L_0 is an arbitrary point on the line:

$$P = d\,\vec{L} + \vec{L_0}$$

The directional vector can be set as the vector along which the distance is measured. The point on the line is the point where the distance is measure from, here B.

The plane formula, like defined in Section 2.5.1, can also be written as below, where *P* is any point in the plane, P_0 is a reference point in the plane, and *N* is the normal vector:

$$(\vec{P} - \vec{P_0})\ dot\ \vec{N} = 0$$

As the reference point A is used, while N is the normal vector. The plane is defined by a point, A, and the plane normal, N. By putting the right side of the line formula into the plane formula, in place of P, we get;

$$((d\,\vec{L} + \vec{L_0}) - \vec{P_0})\ dot\ \vec{N} = 0$$

Which can be reevaluated to solve for distance, *d*, giving us the equation:

$$d = \frac{((\vec{L_0} - \vec{P_0})\ dot\ \vec{N})}{(\vec{L}\ \ dot\ \vec{N})}$$

To resemble Figure 2.5, where L is the vector along which distance is measured, it can be simplified to:

$$d = \frac{((\vec{B} - \vec{A})\ dot\ \vec{N})}{(\vec{L}\ \ dot\ \vec{N})}$$

The intersecting point between the line and plane can then be found by the line equation, at the top of this section, with the newfound distance, *d*.

2.5.4 Acquisition: Bilinear Interpolation

When acquiring data from the input frames, the data will often be requested outside of the exact integer locations of the actual input data. A popular method for acquiring these datasets is bilinear interpolation[14]. It will take the values of the 4 pixels around the intersection point and interpolate them for an efficient estimation of the value that would take place in the exact place of the point.

The 4 pixels around the point are found by taking the floor and ceil values of the x and y coordinates.

$$\begin{aligned} P_{min,min} &= P_{floorX,floorY} \\ P_{min,max} &= P_{floorX,ceilY} \\ P_{max,min} &= P_{ceilX,floorY} \\ P_{max,max} &= P_{ceilX,ceilY} \end{aligned}$$

The distance internally in this quadrant, *u* in x direction and *v* in y direction, is given by:

$$\begin{aligned} u &= x - floorX \\ v &= y - floorY \end{aligned}$$

These distances, and the rest of the setup, are illustrated in Figure 2.6 below. In the X axis interpolation the two top and the two bottom corners are interpolated each pair on their own:

$$\begin{aligned} P_{top} &= P_{min,min} \cdot (1-u) + P_{max,min} \cdot u \\ P_{bot} &= P_{min,max} \cdot (1-u) + P_{max,max} \cdot u \end{aligned}$$

Which is further interpolated again on the Y axis to find the final pixel value P_{value}:

$$P_{value} = P_{top} \cdot (1-v) + P_{bot} \cdot v$$

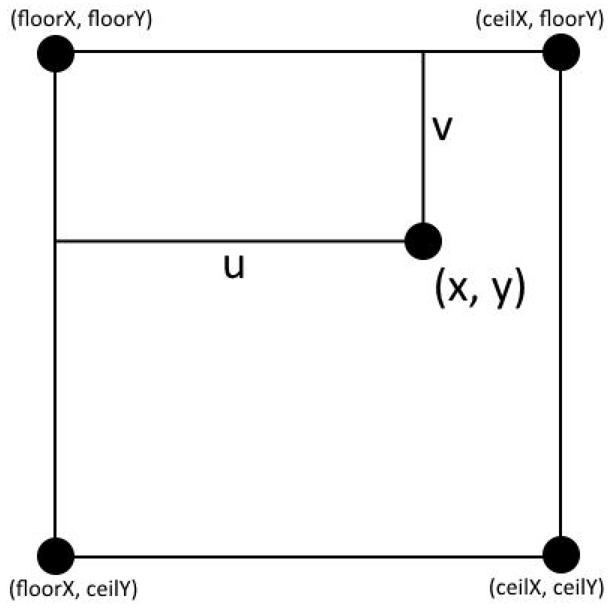


Figure 2.6: Bilinear interpolation. Around the point (x, y) with coordinates with floating point precision, are 4 pixels at integer locations with the actual pixel values. To approximate these their values are interpolated for the output result of the interpolation. The surrounding pixels are found by taking the floor and ceil values of the x and y coordinates. The distances u and v are accounted for when calculating the resulting pixel value.

2.5.5 Accumulation: Weighting

Weighting of the input data collected decides the importance of this data, and can heavily influence the resulting output volume. Many parameters can be used to decide the weight of the input data[9]. In this thesis the focus has been on distance based.

2.5.5.1 Inverse linear distance weighting

Sometimes called linear weighting for short in this thesis, it is a weighting system deriving its score from the distance and a maximum distance, as shown below:

$$W = 1 - \frac{dist}{maxDist}$$

MaxDist is the maximum value the distance, *dist*, can have, and a higher distance will give a false sub-zero weight, see Figure 2.7. This formula is simple, computationally efficient, and as the weight will scale inverse linearly with the distance, fair.

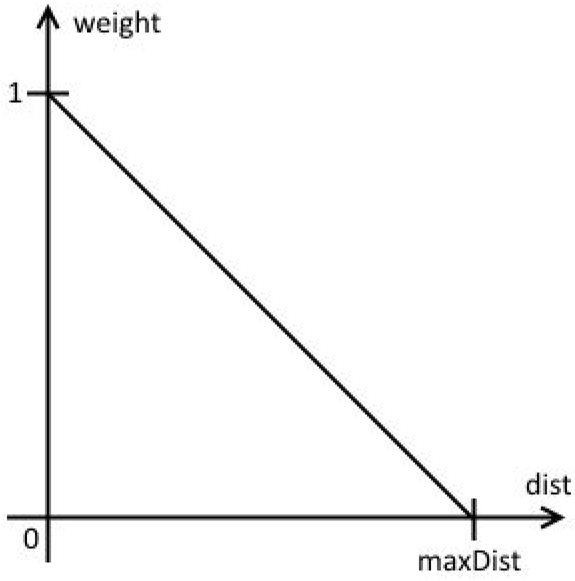


Figure 2.7: Inverse Distance Weighting - Giving a weight score for distances between 0 and maxDist, where 0 gives a weight of 1, and it scales down linearly towards a weight of 0 at maxDist. Any distance further away will receive a negative weight, which should be avoided at all cost in most situations.

2.5.5.2 Gaussian weighting

An alternative is using the gaussian curve as the basis for the weighting scheme. With its inverse S shape it gives a higher weight to points close by, and a lower weight to those further away, compared to inverse linear distance weighting. The gaussian weight is given by:

$$W = k \cdot \exp(-\frac{1}{2} \cdot (\frac{dist}{distFactor})^2)$$

But as the scaling of *k* is not necessary we can leave that constant out. The replacement for the standard deviation, *distFactor*, is a value related to maxDistance. To get the appropriate scaling *distFactor* is set as:

$$distFactor = scaleFactor \cdot \frac{\max Dist}{\pi}$$

Where *scaleFactor* is 1 by default, but can be used to scale how broad the curve will be.

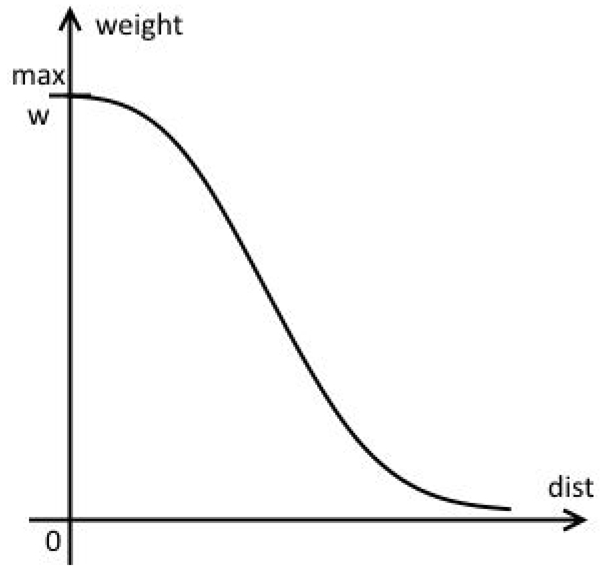


Figure 2.8: Gaussian Weight curve estimation. The curve will resemble a inverted S-curve, leaving a small plateau at the top and slowing down the change as the weight approaches 0. This puts a stronger emphasis on values relatively close to the starting point, while quickly de-escalate down towards zero, but never reaching zero.

2.5.6 Accumulation: Accumulation methods

Accumulation methods will have an effect on how the input data will be added together, and is highly correlated to the weights. They can also affect the performance, as each method will require different processing steps. Here P will reflect the pixel value buffer, W the weight value buffer, p the current pixel value, and w its corresponding weight.

2.5.6.1 General compounding

Accumulation by compounding is done by utilizing a accumulation buffer, for both pixel values and weights. In its simplest form it can be written as:

$$P = P + p \cdot w$$
$$W = W + w$$

Which is very efficient in writing, but will require a final normalization step, normalizing all voxels by:

$$P = \frac{P}{W}$$

2.5.6.2 Alpha-blending approach

An alternative method is to take inspiration from the concept of alpha-blending. Mathematically it is described as:

$$value = value \cdot (1 - \alpha) + addedValue \cdot \alpha$$

With α being the alpha-blending factor, which is generally a static predefined value. The side-effect of this is that it will be putting more emphasis on more recent values, as earlier values will diminish as more values are added. Attempt to make this more useable for 3D Ultrasound has been performed by Gobbi & Peters[4] and resulted in the formula below, rewritten for this setting:

$$P = (1 - w) \cdot P + w \cdot p$$

Despite utilizing the weight as an alpha value for individual importance, more recent additions will still have more value. The big advantages with this method is however that it will always have the scaled pixel value available, and that it requires no weight buffer, and more importantly no normalization step.

2.5.6.3 Compounding alpha-blending

To draw from the advantages of both these methods of accumulation, a middle-way can be created. As mention as 'compounding' by Gobbi & Peters[4], it is actually a blending method using accumulation buffers and historically accurate weighting, as defined below:

$$alpha = \frac{w}{W + w}$$

$$P = P \cdot (1 - alpha) + p \cdot alpha$$

$$W = W + w$$

This way P will always reflect the scaled value of the voxel value, thereby avoiding the need for a final normalization step. Yet the values of P will reflect those from the normalized results of Section 2.5.6.1.

Chapter 3: Method

In this chapter I will describe in detail the method for freehand 3D ultrasound reconstruction that has been implemented. The method is based on previously unpublished work done at SINTEF many years ago. No proper description of the method existed.

Section 3.1 will present the main implementation steps of the algorithm. Section 3.2 will introduce the pipeline that is run both on the overlying level of the FAST framework and the underlying level of the reconstruction steps. Section 3.3 will present the initialization steps to prepare the frames, volume and transformations. Section 3.4 will present detailed discussion for the implementation some of the more troublesome parts, while Section 3.5 explains the parameters of the algorithm, and other steps that can affect the output results.

The algorithm has previously been implemented just for a serial CPU execution, while our goal has been to speed it up as much as possible using a parallelized OpenCL implementation running on GPU. Section 3.6 will explain how this algorithm can be parallelized and what measures are taken for making an efficient OpenCL implementation.

Furthermore I have implemented some alternative 3D ultrasound reconstruction algorithms that will be explained in Section 3.7, including a serial and a OpenCL PNN implementation and a simple serial VNN implementation, as well as a brief description of the alpha-blending compounding implementation with the hybrid algorithm.

Finally, Section 3.8 describes the evaluation methods used to gauge visual quality and runtime performance.

3.1 Algorithm Overview

The hybrid approach is based on each input frame, then finds the voxels that are within relevant distance to it by traversing a dominating direction. These voxels are then filled with the most relevant data from the current input frame. These features gives it hybrid qualities, with both incremental computing and high parallelization capabilities.

After the initial volume initialization and preprocessing, as described in Section 3.3, the algorithm runs the accumulation loop, adding data to the volume from each input frame. This loop is described in Algorithm 1 after the textual description below.

For each frame the series of steps described below are performed. The dominating direction *domDir*, and its value *domVal*, is found by selecting the major component of the normalized normal vector for this frame in volume space. Since the algorithm operate over the *X*, *Y* and *Z* axes differently depending on the dominating direction we have chosen to use the artificial axes *A*, *B*, and *C* to iterate over. The *C* axis corresponds to the axis of dominating direction, i.e. *Z*, while *A* and *B* correspond to the two other axes, i.e. *X* and *Y*.

After acquiring the dominating direction of the frame, we calculate the ranges to iterate over on the A and B axes, see Section 3.4.1. For each combination of values in these two ranges, *a* and *b*, we calculate the *basePoint*. This is a volume space location residing on the image plane, explained in Section 3.4.2. From this point the distances, *d1* and *d2*, are calculated to the previous and the next image planes respectively in the temporal stack of frames. This distance is calculated along the image plane normal of the current frame, and is described in Section 3.4.3. Two parameters are used to limit the distances, the voxel-distance, *dv*, and the maximum distance, *Rmax*. Voxel-distance is the spacing between neighbouring voxels, and it is the lower distance limit, and it will be described in Section 3.5.1. Maximum distance is the upper distance limit, and will be described in Section 3.5.2. We calculate the half width, *df*, from the two distances, *d1* and *d2*, *dv* and *Rmax* by the formula:

$$df = \min\left(\max\left(d1, d2, dv\right), R_{Max}\right)$$

The furthest of the two temporal neighbours, between the lower limit *dv* and upper limit *Rmax*. The equivalent half width in the dominating direction, *dfDom*, is calculated by:

$$dfDom = \frac{df}{domVal}$$

The simplification to dfDom is necessary to allow us to do calculations on the dominating direction, instead of along the image plane normal, which counts for significant improvement in speed in discrete volumes. We use *dfDom* to determine how far away from the *basePoint* we are to go in either direction from the plane as shown in Figure 3.1. From the value of *basePoint* on the dominating direction axis, *baseDom*, and *dfDom* we calculate the range to iterate over on the C axis:

$$cDirRange = baseDom \pm dfDom$$

For each value in the range cDirRange the integer (x, y, z)-coordinate point *volumePoint* is calculated from *a*, *b*, *c* and *domDir*. The cDirRange and its traversal is further explained and illustrated in Section 3.4.4. The distance *d* from the *volumePoint* to the image plane is calculated along the image plane normal. This is to be used for weighting. The intersection point, *intersectionPointVolume*, is calculated where the image plane normal intersects the image plane in volume location. Further this is transformed into local frame space, *intersectionPointFrame*, to find its location among the pixels of the current image. The calculation of distance and the intersections are interrelated, and explained further in Section 3.4.5.

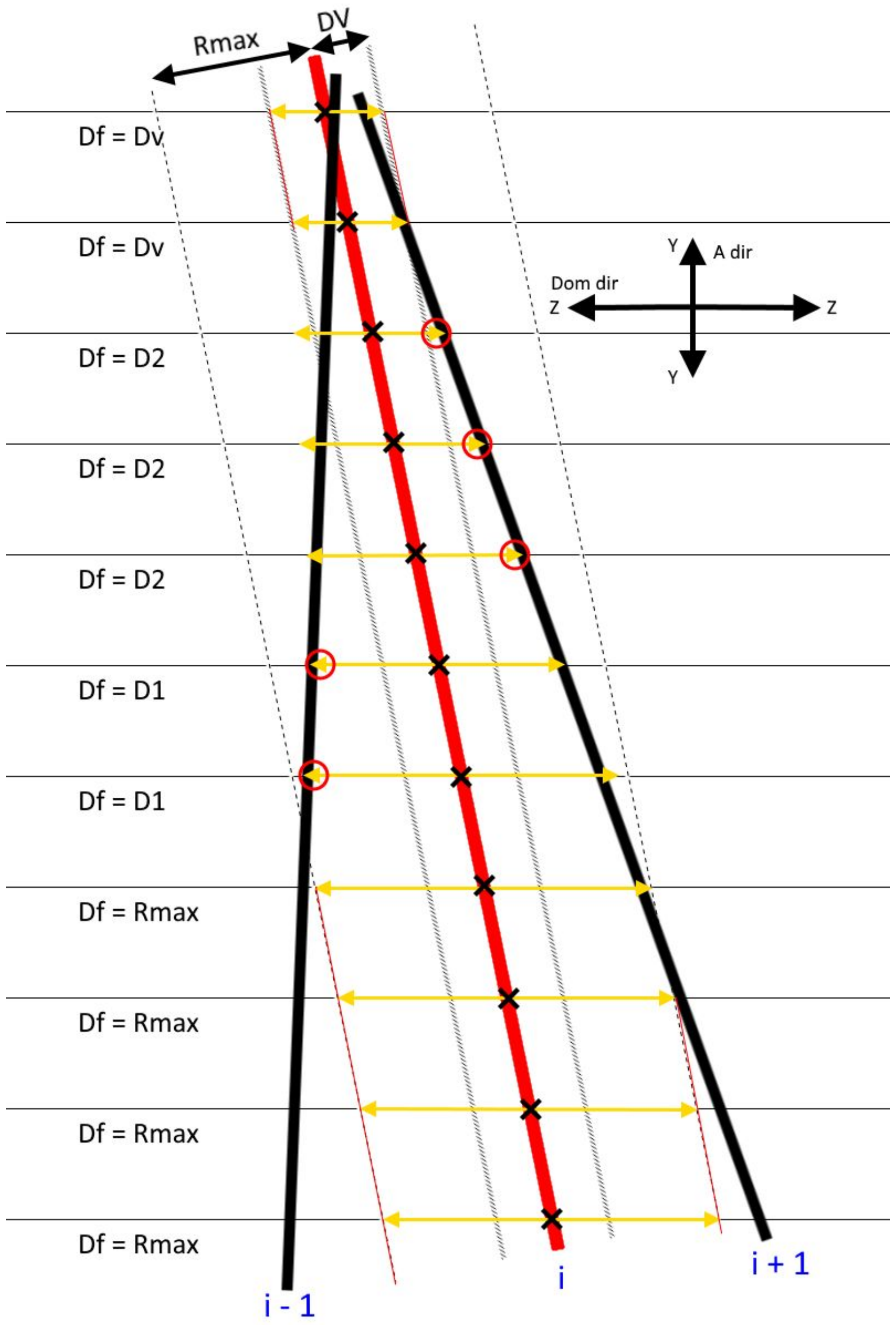


Figure 3.1: Calculation of half width in dominating direction. The thick red line is the current image plane, the thick black line on the left is the previous image plane, and the thick black line on the right is the next image plane. From each a and b coordinate on the current plane we find a basePoint, shown with cross mark, and calculate the distance to the previous and next image planes. If these distances are both within the voxel-distance, dv, its value is used instead. If one plane is further than the value of the max distance, Rmax, its value is used instead. Dv and Rmax are described further in Section 3.5. If distances stay between these two values the bigger distance will be chosen, as shown with red circles. The final value shows how far the algorithm will travel in both directions along the dominating direction axis, as shown with yellow arrows.

A check is performed to ensure that *intersectionPointFrame* is within the limits of the input frame and not just in the infinite plane in which it resides. If this *volumePoint* has successfully passed all the tests until now it will be added to the accumulation volume. The pixel value *p* is acquired by bilinear interpolation at frame position *intersectionPointFrame*, and the weight *w* is calculated by inverse distance function or gaussian distance, as discussed in Section 2.5.5. Bilinear interpolation is explained in Section 2.5.4.

The pixel value p and weight w is then accumulated into the accumulation volume by:

$$P_{x,y,z} = P_{x,y,z} + p \cdot w$$

$$W_{x,y,z} = W_{x,y,z} + w$$

Which will ensure fair contribution from all pixel values affecting this voxel. Each voxel will just be affected once from each frame, but might be affected multiple times from different frames.

Finally, after the accumulation loop is finished, the output volume V can be generated by normalizing every voxel using the relations:

$$V_{x,y,z} = \frac{P_{x,y,z}}{W_{x,y,z}}$$

Which effectively averages all the contributions to the voxels while taking weight into account as well, leaving us with a value within the same range as the input pixels. This is general compounding as described in Section 2.5.6.1.

Algorithm 1 *Hybrid: Accumulating input frames to volume*

runHybrid3dReconstruction(FRAMES, dv, Rmax, volume)

Input:

```
FRAMES          The set of input images
dv              The distance between voxels in the relevant space
Rmax            The maximum distance to apply each frame
volume          Initialized volume
```

Output:

```
volume          The volume with accumulated values
```

```
function runHybrid3dReconstruction(FRAMES, dv, Rmax, volume){
  for frameNr from 0 to totNrOfFrames:
    frame = FRAMES[frameNr]
    frameSize = getSize(frame)
    frameInverseTransform = getInverseTransform(frameNr)
    frameRoot, frameNormal, framePlaneD = getPlane(frame)
    lastRoot, lastNormal, _ = getPlane(frameNr-1)
    nextRoot, nextNormal, _ = getPlane(frameNr+1)

    domDir, domVal = getDomDir(frameNormal)
    aDirStart, aDirEnd = getFrameRangeInVolume(frameNr, domDir, 0)
    bDirStart, bDirEnd = getFrameRangeInVolume(frameNr, domDir, 1)
    for a from aDirStart to aDirEnd:
      for b from bDirStart to bDirEnd:
        basePoint = getBasePointInPlane(frameNormal, frameRoot,
          framePlaneD, a, b, domDir)
        d1 = getDistanceAlongNormal(basePoint, frameNormal,
          lastRoot, lastNormal)
        d2 = getDistanceAlongNormal(basePoint, frameNormal,
          nextRoot, nextNormal)
        df = min( max(d1, d2, dv), Rmax)
        dfDom = df / domVal
        cDirStart, cDirEnd = getDomDirRange(basePoint, domDir,
          dfDom, volumeSize)
        for c from cDirStart to cDirEnd:
          volumePoint = getVolumePointLocation(a, b, c, domDir)
          if volumePointOutsideVolume(volumePoint, volumeSize):
            continue
          dist = getPointDistanceAlongNormal(volumePoint,
            frameRoot, frameNormal)
          if abs(dist) > df:
            continue
          intersectionPointWorld =
            getIntersectionOfPlane(volumePoint, dist, frameNormal);
          intersectionPointLocal =
            getLocalIntersectionOfPlane(intersectionPointWorld,
            frameInverseTransform);
          if isWithinFrame(intersectionPointLocal, thisFrameSize:
            float p = getPixelValueData(intersectionPointLocal);
            float w = 1 - (abs(dist) / df);
            accumulateValuesInVolumeData(volumePoint, p, w)
} //end runHybrid3dReconstruction
```

3.2 Algorithm pipeline

Implementation of a 3D Ultrasound Reconstruction algorithm requires more than just main algorithm steps. Multiple steps are performed in a series of events called a pipeline. The utilized FAST framework, as described in Section 2.3, links multiple processing objects together in a pipeline where the output of one step is input in the next. Here a *ImageFileStreamer* is used to import a series of ultrasound images to the reconstruction algorithm, *US3DReconstructor*, that will store all the images before processing them. The resulting volume, which can be seen as a 3D image, is exported using *MetaImageExporter* to export it in mhd format with volume data and spatial transformation of their locations. The reconstruction algorithm performs 3 main steps:

1. *Initialization*. Initializing the volume and doing pre calculations. See Section 3.3.
2. *Accumulation*. The main algorithm loop. See Section 3.1
3. *Normalization*. Normalizing the volume according to weights as described in Section 2.5.6.1, creating the final output volume.

The steps are all displayed in Figure 3.2. For this thesis the input and output data are static, but it can stream directly from an input source and to an output visualization if necessary. This will be necessary with real-time reconstruction in which case the pipeline would have to be optimized slightly. Real-time reconstruction will be discussed further in Section 3.7.3.

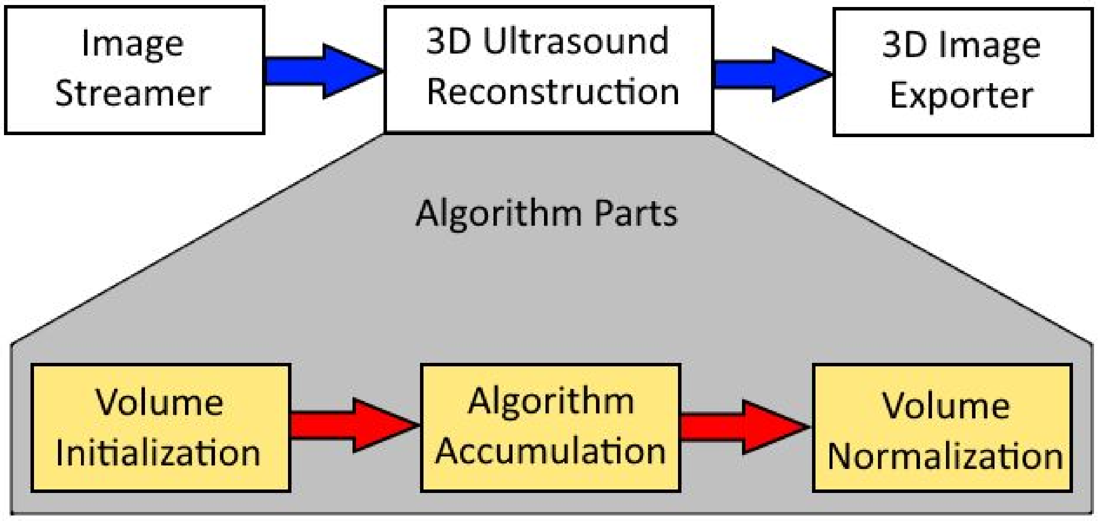


Figure 3.2: FAST and algorithm pipelines. Image Streamer streams input images that are stored by 3D Ultrasound Reconstruction algorithm. Once all images are acquired the 3 parts of the algorithm are performed. The volume is initialized from the input images, each input image is accumulated to the volume, and the volume is normalized to create the final output volume. The resulting volume is exported to a static file.

3.3 Volume initialization and pre-calculations

To reconstruct ultrasound images to a 3D volume, the volume needs to be defined in world space. It is defined by a target area, volume orientation, and spatial resolution for each axis inside the volume. These steps are a lot easier to acquire when processing all input images in bulk.

Volume orientation is decided by a reference frame plane that is set as a XY-plane in the volume and thereby deciding the initial frame-to-volume transformation.

The target area is set to the boundary in which all the image pixels of all input frames can be put, with axes decided by the volume orientation. In the case of square input images, a check on each of the corners is sufficient. A limitation on this area can also be set by letting a certain percent of the image corners be outliers, allowing for greater resolution in the final volume.

Spatial resolution is deducted from the desired output volume size, by comparing the intermediate volume size to the desired volume size and calculating a scaling factor that is applied to all three axes.

This algorithm part is responsible for finding the volume size, the volume-to-world transformation, and a frame-to-volume transformation for each input frame. In addition some variables are calculated and stored for use in the accumulation loop. The volume initialization steps are described below:

First, the image spacing is applied to all the transformations of the input images, as this is not automatically applied in the FAST framework. This ensures that the transformation connected to each input image corresponds to the frame-to-world transformation.

Second, the frame-to-world transformation of the reference frame is stored as $T_{world}^{refFrame}$, and its corresponding inverse transformation, $T_{refFrame}^{world}$, is added to each input frame transformation by left side matrix multiplication to find their positions in space relative to the reference frame. The minimum and maximum coordinate of each axis is derived from the frame corners to find the area required relative to the reference frame.

Third, the minimum coordinates are used to find the transform that will ensure that the minimum coordinate will lie at point *(0, 0, 0)* in the volume. This transform is found by doing a negative translation of the minimum coordinates, i.e. the minimum coordinates *(-17.5, -25.2, -3.8)* would translate all frames by *(17.5, 25.2, 3.8)*, with translation matrix $T_{translate}$.

Fourth, the scaling is calculated. The size of the current volume, *currentVolSize*, is found by multiplying the size of each axis, derived from the difference between its minimum and maximum value. Given the wanted amount of voxels, *volSizeMill*, in millions, as described in Section 3.5.3, and *currentVolSize*, the scaling factor is derived using the cube root as in the equation below:

$$scaleFactor = \sqrt[3]{\frac{volSizeMill \cdot 1000000}{currentVolSize}}$$

The volume is scaled by *scaleFactor* on all three axes, and a scaling transformation, T_{scale}, is created. The final transformation, T_{final}, is derived by adding T_{scale} to $T_{translate}$.

Fifth, T_{final} is added to each input frame to achieve their final frame-to-volume transforms, $T^{frame_i}_{volume}$. In this process 6 things are also calculated and stored for use in the accumulation loop:

- *minCoordsFrame, the* minimum coordinates on each axis of the frame corners. Used to limit A and B ranges.
- *maxCoordsFrame, the* maximum coordinates on each axis of the frame corners. Used to limit A and B ranges.
- *baseCorner*, the position of the plane (0, 0) point in volume space. Used as point in plane to represent the plane for calculating the distance to a plane.
- *normal*, the image plane normal. Used to represent the plane and calculate distances.
- *planeDvalue*, a calculated value d from the plane equation. Calculated from *baseCorner* and *normal*. Used to calculate the *basePoint* from the plane equation.
- $T^{volume}_{frame_i}$, the volume-to-frame transformation, derived from the inverse of the transformation currently applied to the image. Used to transform the volume location into frame space to find the local intersection point.

Then, the world-to-volume transform, T^{world}_{volume}, is created by adding $T_{translate}$ to $T^{world}_{refFrame}$ and setting the volume voxel spacing, *voxelSpacing*, to:

$$voxelSpacing = \frac{1}{scaleFactor}$$

The volume-to-world transform, T^{volume}_{world}, is set as the inverse of its counterpart transform T^{world}_{volume}.

Finally, the volume is initialized. The algorithm uses two volumes: the accumulation volume; and the output volume. The accumulation volume is a two-channel volume used by the accumulation loop to store accumulated pixel values and weights, while the output volume is a one-channel volume used to store the normalized pixel values and it is the output of the reconstruction. The accumulation volume is referred to as volume, while the output volume is explicitly referenced. It is important that the volume-to-world transformation and the volume voxel spacing is set correctly in the output volume, in order for it to be displayed correctly in volume viewing tools.

The reconstruction could alternatively have been done with one volume of pixel values used for both accumulation and output, and one volume of weights. However, the two accumulation volume approach can allow a separate unit type to be used for the intermediate step, to allow speed-ups or specific features, like the integer requirement for atomic operations, described in Section 3.6.2.

3.4 Extended information on problematic functions

This section will present some of the more difficult functions introduced in the overview in Section 3.1. Section 3.4.1 will introduce the virtual axes A and B and show how they relate to the X, Y and Z axes, while also explaining how we can calculate the ranges to traverse on the A and B axes. Section 3.4.2 shows how to find the point residing in the image plane by values *a* and *b*.

Section 3.4.3 explains how the distance to neighbouring planes are calculated, so that the half width in dominating direction, *dfDom*, can be calculated. Section 3.4.4 uses *dfDom* and introduces some details to how it is used to calculate the range to be traversed on the C axis.

Section 3.4.5 explains how to find the intersection point from which the pixel value is derived. Section 3.4.6 introduces the check that is performed to ensure that the intersection point is within the actual image area. Section 3.4.7 shows how the pixel value is calculated by bilinear interpolation, and some exception cases.

3.4.1 Calculating the A and B ranges and their relation to the X, Y and Z axes

To calculate the ranges the minimum and maximum volume space locations of the current input frame is used on each axis. For a square frame, like the output from a linear array ultrasound probe, a simple check of the 4 corners of the input frame can be performed to find a volume space minimum and maximum on each axis. For other frame shapes a more advanced or thorough check will be necessary. The algorithm calculates these values in the initialization steps, as they are already needed to find the volume size.

The minimum and maximum is acquired from the real axis corresponding to our required A or B virtual axis. The axes are defined in Table 3.1.

Table 3.1: Relations between virtual axes, A and B, and real axes X, Y, and Z.

Virtual axis:	**A**			**B**		
domDir axis:	X	Y	Z	X	Y	Z
Real axis:	**Y**	**X**	**X**	**Z**	**Z**	**Y**

Given a virtual axis and dominating direction axis the real axis can be found.

The final range is the integer values from the minimum to maximum on the real axis, so that in conjunction *aDirRange* and *bdirRange* traverses a square on the AB-plane. This is a naive approach, seeing that the AB-plane projected input frame might be tilted up to 45 degrees inside of the traversation square, putting half of the traversation square outside of the actual projected input frame, and thereby contributing nothing for the work it performs. A similar situation to this is shown in Figure 3.3.

It is possible to counter this by checking the *basePoint* for each *a* and *b* value and using the same *isWithinFrame* function which is used in the innermost loop of the algorithm. The *basePoint* is described more in Section 3.4.2, but for this section it is enough to know that on the A and B axis it would have the values of *a* and *b*.

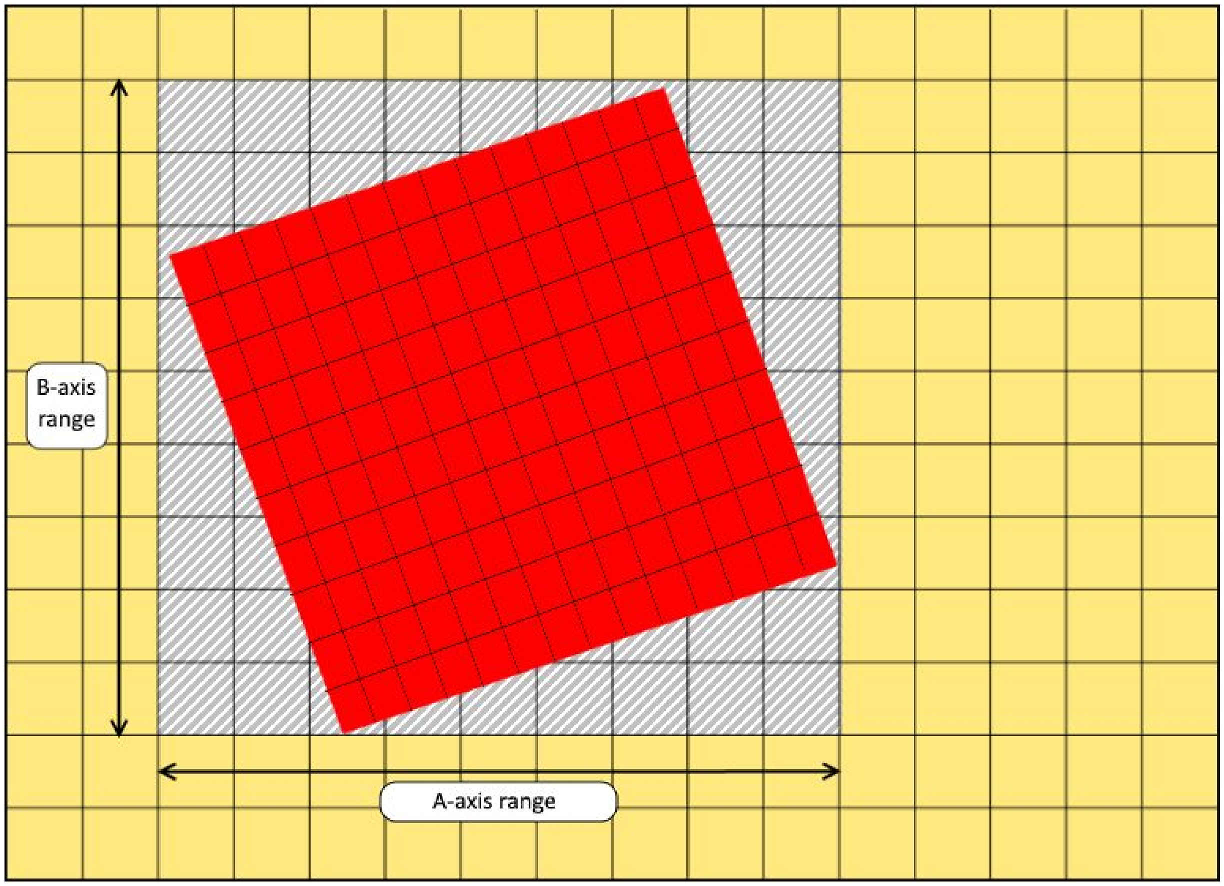


Figure 3.3: Input frame traversing ranges, A and B. Showing a 2D projection of a square image frame (red) over the volume grid (yellow) in the plane of the AB-axes. The grey region in the volume grid corresponds to the coordinate values of a and b that will be traversed. These are calculated from the 4 corners of the squared image frame. The grids are just to show that the scales of the frame grid and the volume grid can have different density, most volumes will have more than 100 voxels in each direction.

Due to the nature of the scans, very few input frames will be tilted 45 degrees. In most cases they will barely be tilted at all, as we align the volume to a root frame, and every 90 degree tilt is effectively 0 tilt. The abovementioned check on *basePoint* has been tested, but appeared to give a slight performance penalty and has therefore been left out. It can be implemented in the case where response guarantees are important and in scenarios that would create a lot of tilt relative to the root frame, which in general is put to the first acquired frame.

3.4.2 Finding the basePoint given two coordinates a and b

With the values of the two coordinates, *a* and *b*, giving the position on two of the axes in the volume, it is necessary to find the position on the third axis before further calculations can be done. This results in a (x, y, z) volume location that we name *basePoint*, representing a location residing in the current image plane.

To calculate this location the *a* and *b* coordinates, the dominating direction, *domDir*, and the plane equation are necessary. The *a* and *b* coordinates will give two of the real axis coordinates in accordance with *domDir* and Table 3.1. The algorithm represents a plane with an image plane normal, *normal*, and a pre-calculated constant *d* value, *planeDvalue*, for each image plane. The three values of *normal* can in conjunction with *planeDvalue* replace the constant values of the planar equation as described in Section 2.5.1. The new equation is:

$$normal_x \cdot x + normal_y \cdot y + normal_z \cdot z + planeDvalue = 0$$

Then we have all four constants of the equation as well of two out of three coordinates, where the dominating direction is missing.

Taken the example where dominating direction is on the X axis, the formula above can be restructured to calculate the value of *x* as shown in the formulas below:

$$normal_x \cdot x = -(normal_y \cdot y + normal_z \cdot z + planeDvalue)$$

$$x = \frac{-(normal_y \cdot y + normal_z \cdot z + planeDvalue)}{normal_x}$$

The equation can be restructured similarly to calculate *y* or *z* if either of those are the dominating direction. Due to the fact that the image plane normal in dominating direction, *domVal*, can never be zero, and there will always be just one point suitable for this solution.

3.4.3 Calculating the distances d1 and d2 from plane-to-plane along normal

This step requires the algorithm to find the distance from the current image plane to another, with some specifications. In the current plane it starts from a specified point of origin, *basePoint*, and the distance is measured along its normal, *normalA*. The other plane, *planeB*, is defined by its normal, *normalB*, and a point in the plane, *pointB*.

These specifications makes it the same as the point-to-plane distance as explained in Section 2.5.3. Here *basePoint* corresponds to L_0, *normalA* corresponds to *L*, *pointB* corresponds to P_0, and *normalB* corresponds to *N*, giving the new equation:

$$distance = abs(\frac{(pointB - basePoint)\ dot\ normalB}{normalA\ dot\ normalB})$$

If the normals are orthogonal to each other, the divisor will be zero, which gives two possible situations:

1. The dividend is also zero, implying that the *basePoint* and *normalA* is residing in *planeB*. The distance is zero.
2. Otherwise the line will be parallel to the plane. The distance is set to infinite, or in practice a value higher than any possible *Rmax*.

This distance is calculated to both the previous, and the next, image planes. The distance *d1* marks the distance to the previous, and the distance *d2* marks the distance to the next.

3.4.4 Calculating and traversing the C-direction range

The simple theoretical equation for finding the C-direction range from the *basePoint* value in dominant direction, *baseDom*, and half width in dominant direction, *dfDom*, is given in the equation below:

$$cDirRange = baseDom \pm dfDom$$

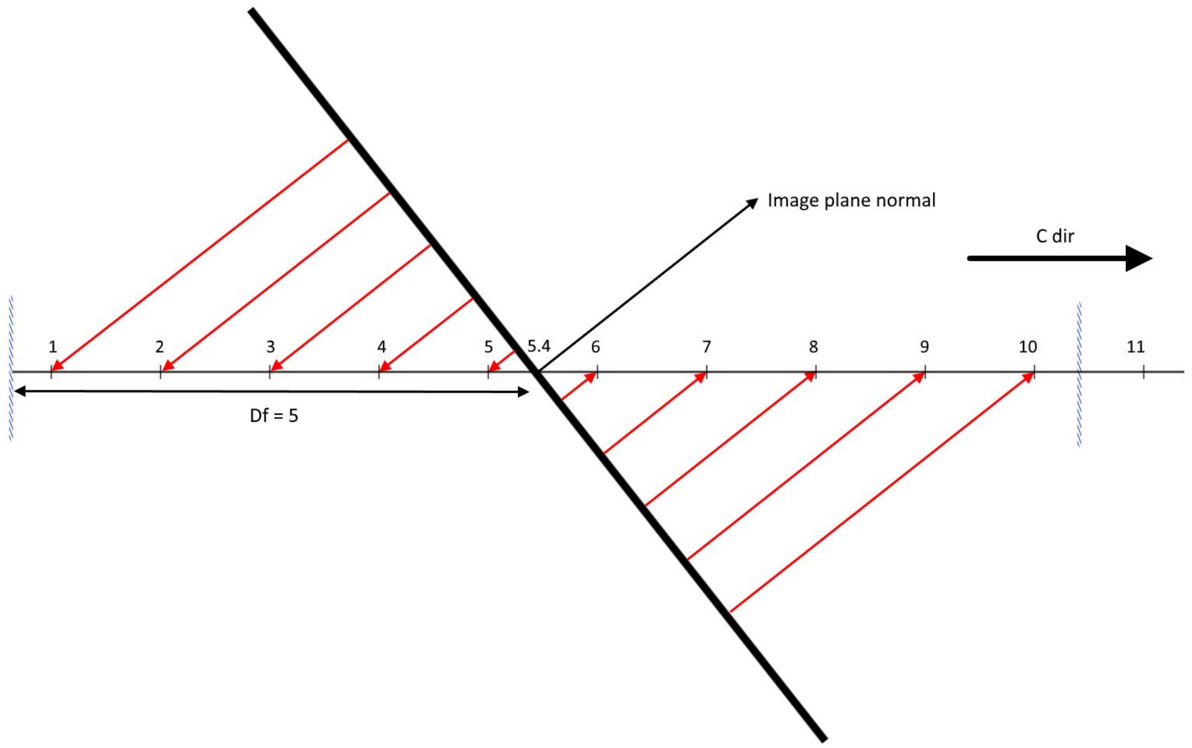


Figure 3.4: Iterations over the C axis. The central point at C axis location 5.4 is the basePoint on the C axis. This is also the intersection point between the image plane and the C axis as calculated in Section 3.4.2. This figure is for simplicity projected to a 2D view where the up-down axis can be any arbitrary direction perpendicular to the C axis. We can see the half width in dominating direction, df, is 5.0, and the cut-offs in positive and negative direction is marked by blue vertical lines at approximately 0.4 and 10.4. Each integer point between these boundaries are calculated by following the image plane vector to find a corresponding point on the image plane. These point-to-plane connections are displayed with red arrows. At the image plane location of these connections the pixel value is calculated by bilinear interpolation.

In practice it is actually calculated as a cutoff integer range limited by the volume size in dominant direction, *sizeDom*:

$$cDirRange = \begin{pmatrix} max(0, ceil(baseDom - dfDom)) \\ min(floor(baseDom + dfDom), (sizeDom - 1)) \end{pmatrix}$$

The range is traversed for each integer value between the minimum and maximum limits, ends inclusive. This effect can be seen in Figure 3.4, where all steps between the cutoffs at half width distance are iterated. The choice of this cut-off can affect how many voxels are traversed, especially when the half width, *df*, is set to the minimum, *dv*.

3.4.5 Finding the frame intersection point through distance along normal

Before accumulating values to a given *volumePoint* a pixel value and weight needs to be calculated. To calculate the pixel value the frame space location to interpolate from is required, and a distance is used to calculate the weight. For each *volumePoint* in the *cDirRange* the pixel value is taken from the closest point in the image plane. This point is found from the intersection point between a line, following the image plane normal from *volumePoint*, and the image plane. Calculating the distance to the intersection and finding the intersection point can be done in conjunction, as will be explained in this section. See Figure 3.4 for spatial reference.

When defining a plane by a *rootPoint* and a *normal* the distance from *volumePoint* to the plane can be calculated from the function in Section 2.5.2. If this calculation is performed without taking the absolute value, the distance moved, *movement*, from *volumePoint* to the plane is found instead as shown in the equation below:

$$movement = (rootPoint - volumePoint)\ dot\ normal$$

The intersection point in world space, *intersectionWorld*, can then be found by traveling *movement* distance along the normal. It is calculated by the following equation equation:

$$intersectionWorld = volumePoint + normal * movement$$

Where all variables are vectors of size 3, except for the scalar *movement* which is elementwise multiplied with *normal*.

A local intersection point in frame space will decide which the pixels will be used to calculated the pixel value. The local intersection, *intersectionLocal*, is calculated by transforming *intersectionWorld* into frame space:

$$intersectionLocal = T_{frame}^{world}(intersectionWorld)$$

Where the first two coordinates of *intersectionLocal* are the local 2D coordinates and the thirds coordinate will be approximately zero.

3.4.6 Assuring intersection is within actual frame

A intersection point residing in the plane of a image, as calculated in Section 3.4.5, can still occure outside the area in which the actual image extends. To check if the intersection point is within the actual image area *intersectionLocal* is used. The location on the X and Y axis is checked to be between zero and the input image axis size, while the Z axis is checked to be zero.

Due to floating point inaccuracies in the process of calculating *intersectionLocal* a zero value can be slightly negative. A small buffer value can be added to ensure that these points are included.

3.4.7 Calculating pixel value with bilinear interpolation

The pixel is calculated by bilinear interpolation, as described in Section 2.5.4, acquiring data from 4 surrounding pixels. Due to the inaccuracy of floating point calculations and the added buffer allowing acquisition slightly outside of the frame, a special case using only the neighbouring pixels that are actually within the frame will only use the corners residing inside of frame.

If it is necessary to minimize the number of image accesses it is possible to use the value of just the nearest pixel, requiring 1 access instead of 4. This speedup will be at the cost of reconstruction accuracy, and artefacts can occur.

3.5 Hybrid reconstruction parameters

There are mainly three parameters to this reconstruction: *dv*, *Rmax*, and *volSizeMill*. Section 3.5.1 will introduce the voxel-distance *dv* and discuss its effects. Section 3.5.2 will introduce *Rmax* and discuss its effects. Section 3.5.3 will present *volSizeMill*.

3.5.1 Parameter DV

The parameter *dv* works as a lower limit for the half width, which decide the size of the range on the dominating axis. *dv* is the distance between voxels, such that if the distance between frames are shorter than this, they will at least affect one voxel. It can alternatively be related to the slice thickness, implying that *dv* is the limit on how far the pixel value has to affect. How many voxels it affects is strongly influenced by the cut-off described in Section 3.4.4.

In this implementation the half width is calculated in volume space, which per definition implies that the distance between voxels is 1, and $dv = 1.0$ in this thesis accordingly. The value of *dv* can still be varied with minor implications, $dv = 0.5$ will usually affect only one voxel when df is limited by, which can make the appearance look more jagged, and less smooth, due to sampling theory.

3.5.2 Parameter Rmax

The parameter *Rmax* is the upper limit for the half width. It influences how distant a pixel can have influence, giving it a similar effect to the size of the hole-filling grid. Too low value can leave gaps in the volume, while a too high value can create invalid data and smoothen the volume too much.

Rmax should be set in relation to the volume size, as a higher resolution will create bigger gaps in voxel space.

3.5.3 Parameter volSizeMill

The parameter *volSizeMill* is the requested output volume size, given in million voxels. It is used for both the accumulating volume and the final output volume. The value can range from 1 to 256, where 32 is the default value, annotated as 32M. Bigger value will create more voxels to process and thereby increase runtime.

3.6 Parallelization and OpenCL implementation

An OpenCL implementation of the hybrid reconstruction algorithm has also been created, allowing major speedups when running on GPU. OpenCL is previously described in Section 2.4.

Section 3.6.1 will introduce how the algorithm can be parallelized for OpenCL. Section 3.6.2 will explain how each kernel is run and the steps necessary to make it work in OpenCL 1.2, and also show the atomic operations necessary to allow accumulating data to the voxels while ensuring consistency.

3.6.1 OpenCL implementation introduction

Implementing an algorithm in OpenCL can lead to significant speed gains, especially when running on GPU. To achieve speed improvements it is necessary to parallelize tasks, to utilize all computational power and avoid bottlenecks. All input images are accumulated to the volume without any dependencies to each other, much like the PNN, and this makes the input images a good starting point for parallelization. Unlike a PNN reconstruction, the hybrid reconstruction iterates over voxels, avoiding unnecessary calculations. Unlike the VNN, it acquires data only from a single input image, and requires no costly search. PNN and VNN reconstructions are described in Section 2.2.3.

For a OpenCL implementation there are two levels of parallelization: kernels, and work items, as described in Section 2.4.2. Input images can be parallelized in kernels, while the *aDirRange* and *bDirRange* can be parallelized to work items in each kernel. Each work item is assigned one combination of *a* and *b* as the workload for each coordinate (*a*, *b)* can be significant. Processing multiple (*a*, *b*) coordinates per work item has not been attempted, but in theory it has been shown to increase performance in multiple occasions. If such an approach is attempted, it is recommended to assure an even distribution of the workload of each work item by utilizing the distribution of distances to neighbouring planes across the image plane, which can be done by mirroring across the centre of the image to group the shortest and longest distances together in one work item.

3.6.2 OpenCL accumulation volume and atomic operations

Many variables may need to take a different form when running in an OpenCL kernel, but in the hybrid reconstruction algorithm the most notable is the accumulation volume. As OpenCL does not come with image read-writes before OpenCL 2.0, and the test system is using OpenCL 1.2, accumulation has to be done in an basic type array.

The accumulation volume will be updated from multiple units working in parallel, which is not automatically going to be thread safe. To work around this it is possible to use a lock or to provide accumulation only by atomic operations, but it should be noted that atomic operations are available only for int types in OpenCL 1.2. The OpenCL implementation of reconstruction algorithms in this thesis use a 32-bit unsigned integer (uint) array for accumulation and the atomic addition function, atomic_add, to ensure consistency. To be able to simulate a possible floating point input and the correct weighting accuracy the numbers are scaled up with a predefined *granularity*. Both the pixel value, p, and the weight, w, are multiplied with the *granularity* when accumulating values, and divided by the same amount again when normalizing. 1000 has worked out as a good value, ensuring accuracy without causing overflow, which would severely corrupt results.

3.7 Alternative 3D Ultrasound Reconstruction Methods

In addition to the hybrid reconstruction algorithm, two standard reconstructions have been implemented, as well as a variation of the hybrid method. A PNN reconstruction with hole-filling which will be used for comparison, and a trail of a VNN reconstruction. Both solutions use the same pipeline as the hybrid algorithm, with the exception of the VNN, where no normalization step is necessary. The PNN solution is implemented for both serial CPU, and OpenCL parallel GPU execution, and will be presented in Section 3.7.1. The VNN solution is implemented for serial CPU execution, and is kept at its simplest form for simple testing. Section 3.7.2 introduces the VNN solution.

Additionally, the alternative hybrid implementations is described in Section 3.7.3, utilizing a different accumulation method to avoid the need for normalization.

3.7.1 PNN 3D US Reconstruction and Hole Filling

The PNN reconstruction algorithm is very related to the hybrid reconstruction algorithm as both iterate over each input image and gather data. It can run as an option in the main hybrid accumulation loop. The implemented PNN algorithm affects one voxel per input pixel, and compounds these together with compounding with $w = 1.0$ for all pixels, as shown in Algorithm 2. The OpenCL implementation of the accumulation has no further changes other than those discussed in Section 3.6.2.

```
Algorithm 2 PNN: Accumulating input frames to volume
accumulatePNN(FRAMES, volume)
Input:
 FRAMES           The set of input frames
 volume           Initialized volume
```

```
Output:
 volume            The volume with accumulated values
```

```
function accumulatePNN(FRAMES, volume) {
  for frameNr from 0 to totNrOfFrames:
    frame = FRAMES[frameNr]
    frameTransform = getTransform(frame)
    frameSize = getSize(frame)
    for x from 0 to frameSize.x:
      for x from 0 to frameSize.y:
        pixelPos = (x, y, 0)
        volumePos = transformPoint(frameTransform, pixelPos)
        if !volumePointOutsideVolume(volumePos, volumeSize):
          p = getPixelValue(frame, pixelPos)
          accumulateValuesInVolumeData(volumePos, p, 1.0)
```

Hole-filling can be performed as part of the normalization step. Hole-filling has been implemented where the pixel is set to the average of all the actual values in the hole-filling grid. In the OpenCL case was implemented with local memory, in three dimensions this severely limits the local work group size in each direction with increasing grid size. Due to the small local work group size compare to the allocated local memory size, it might be necessary for each work item in the work group to load many values to local memory.

3.7.2 VNN 3D US Reconstruction

The VNN reconstruction algorithm operates in very different ways from the other two reconstruction algorithms, and has been implemented mostly as an alternative to check accuracy. Finally it is simply implemented as shown in Algorithm 3, with a naive solution for finding closest plane. It will not be used for results in this algorithm, due to a slow running time.

```
Algorithm 3 VNN: Adding the closest data to each voxel

reconstructVNN(FRAMES, Rmax, volume)

Input:
 FRAMES            The set of input frames

 Rmax              The maximum distance to closest frame before
                   value is defaulted to zero

 volume            Initialized volume
```

```
Output:
 volume            Reconstructed volume
```

```
function reconstructVNN(FRAMES, Rmax ,volume) {
  for voxel in volume:
    closestFrame = findClosestFrame(FRAMES, Rmax)
    closestFramePoint, dist = findClosestPoint(voxel, closestFrame)
    if dist < Rmax:
      volume[voxel] = getPixelValue(frame, closestFramePoint)
```

3.7.3 Hybrid with Alpha-blending compounding

To make an attempt to gauge the real-time capabilities of the hybrid reconstruction, an alpha-blending approach was attempted. This normalizes the pixel value with each addition, making the total normalization step unnecessary, which is essential to make the work required between frames as small as possible. So instead of accumulating one frame to the volume, and then normalizing all of it, the volume is always ready to be displayed after accumulation.

The compounding alpha-blending approach was used, as described in Section 2.5.6.3. The implementation builds on the same hybrid algorithm explained in this thesis, with the same steps as described in Section 3.1, but the accumulation step was changed. The blending approach requires us to perform multiplication, and between the old and the new value, putting one more requirement on the OpenCL kernel accumulation, as described in Section 3.6.2. OpenCL 1.2, as used in this implementation has no support for atomic read-and-multiplication, but it is possible to use a semaphore array, as discussed in Section 2.4.4. This puts a spinlock on a section of code which can only be accessed, with read or write, by one processing unit for each voxel position. This allows us to perform more steps than previously in the locked section, and allows us to use float data for higher accuracy calculations.

No real-time capable pipeline was implemented in this thesis, but finally some ideas are discussed in Section 7.5 for Further Work.

3.8 Evaluation

As this reconstruction algorithm is previously unpublished, tests have been performed to evaluate both visual quality as well as performance. We have focused on evaluating the *visual* results to be on line with other good algorithms, while speed *performance* has been evaluated to see how fast such a reconstruction can be done, and look at possibilities for a real-time solution. Valuing top-end performance at decent quality.

Section 3.8.1 will introduce the sets of input data that have been used in this evaluation, and introduce general traits of input data that will affect performance. Section 3.8.2 describes the visual quality evaluation comparing two hybrid configurations, a PNN configuration, and an existing reconstruction.

Section 3.8.3 describes how runtime performance was tested, to reflect the strengths of different reconstruction algorithms and differentiate their configurations.

3.8.1 Data sets

Input data will affect the results that can be expected from the reconstructions. This section aims at describing these effects, and introducing the data sets that have been used for this thesis. Section 3.8.1.1 introduces general traits of input data, and common aspects of all data sets used. Sections 3.8.1.2 trough 3.8.1.6 introduces the specific traits of each data set.

3.8.1.1 General traits

Different data sets will challenge the reconstruction algorithms in different ways, allowing us to not only compare their performance on optimal condition inputs, but find weaknesses and determine how well the algorithms can handle them. There are mainly three properties that will be considered: distance of data, detail of data, and overlapping data. Different situations will affect different properties, and some of the most pronounced situations will be explained below.

A change in *scan distance* determines the spread between each scan image. This is highly influential to how much gap filling has to be done by the algorithms, increasing the uncertainty it has to work with. With some training an operator should be able to get even scans with a evenly distributed distance of maximum 1 or 2 pixels[4], on a fair sized volume. In algorithms adjusting the volume to a specified voxel count rather than a static voxel-distance, this will increase the voxel-distance and thereby the resolution of the final volume. Low scan distance will increase the amount of overlapping data.

Scan movement, the motion over which the probe is moved. *Straight scans* will keep the slices parallel to each other. *Fan scans* are sometimes used, for instance to capture an object behind bone or other shadow-inducing objects. Fan scan are created by tilting the probe, so that distant parts of the scan is moving more than adjacent parts, from frame to frame. This creates an uneven distribution of distance across the frames, where remote areas will have more spacing to neighbouring frames than middle or near areas. Remote areas are then more suspicable of leaving gaps, while adjacent areas have to be heavily compounded.

An increasing number of *scan passes* across the volume will increase the amount of overlapping data. Multiple passes are usually used to cover a bigger area in multiple directions, but the old and the new data should overlap to create a continuous volume. Such an overlap can create a challenge, when data from different time points and points of origin has to account for inaccuracies in calibration and acquisition, as well as patient movement. Creating a representable merge will be a much harder task than merging neighbouring slices. Representing the images be done with a balance between displaying inaccuracies, and over smoothing edges.

The importance of *detail* will vary with each set and use case. Ultrasound is generally not the best modality for capturing details, but the level of detail will still change from one situation to the next. An analysis requiring clear edges will put a higher demand on the algorithm to preserve edges rather than smoothing. Operators can also make the details more clear by aligning the scan slices with the plane in which the detail is most important.

All the sets described below are *neurology scans*, acquired intra-operatively in a tumor resection. The sets are from three different patients, where set 3, 4, and 5, are acquired in three different stages of an resection. The specifics of each set will be described below, and unless otherwise mentioned they are single pass acquisitions.

3.8.1.2 Set 1: Patient A

Set 1 is a fan scan, with almost all movement done by tilting the probe. The scan is depicting a small tumor. The scan is acquired in a top-bottom direction, roughly aligning its middle slices with the axial plane.

3.8.1.3 Set 2: Patient B

Set 2 is a quite straight set, but with a bit of fanning, especially at the ends. The scan covers a large, gradual tumor, covering most of the input slice at its centre. This set is acquired by two parallel passes, with approximately 50% overlap. Edges at the overlapping parts are clearly visible, especially in the tumor. The scan is acquired in a top-bottom direction, roughly aligning it with the axial plane. This is the first set used for performance evaluation and contains a total of 660 frames, with a resolution of 330 x 552.

3.8.1.4 Set 3: Patient C - Before resection

Set 3 is also rather straight, but with a steep fan curb at one end. The scan contains a tumor infecting some tissue in between some liquid. The scan is acquired in horizontal direction, roughly aligning it with the coronal plane. This is the second set used for performance evaluation and contains a total of 573 frames, with a resolution of 229 x 552.

3.8.1.5 Set 4: Patient C - During resection

Set 4 is acquired by straight scans, but it consists of two passes resembling a V-shape. The two scans only overlap in part of the passes, and are entirely separate in the remainder. The tumor has been partially removed. The scan is acquired across the sagittal plane, slightly aligning it with the axial and coronal planes.

3.8.1.6 Set 5: Patient C - After resection

Set 5 is like set 4 acquired by two straight scans in a V-shape. In this set they are overlapping in all of the frames. Resection is complete and the tumor should be removed. The scan is acquired across the sagittal plane, slightly aligning it with the axial and coronal planes.

3.8.2 Evaluation of visual quality

A visual evaluation was done by a comparative review done by ultrasound technicians. Four reconstructions were put side-by-side from all of the abovementioned data sets, with multiple view angles, taken from axis-aligned planes in the world coordinate system. Planes were acquired from Slicer[1], by aligning the the slices with the tumor and switching between reconstructed volumes without changing the position.

The query was given as a PDF-document with background info, evaluation info, and five sets with 2-3 planes each. All sets were displayed with min/max values set to 1-255 for consistency, and it was pointed out that this might cause lower contrast than optimal scaling. The total score for each reconstruction method was the only desired variable. Respondents were allowed to score each set or image, but these were finally averaged to one total score. A score between 0.0 and 10.0 was given, evaluating the quality and usefulness of the reconstruction. 10.0 being the best score, and 5.0 indicating a just usable result without perfection. The evaluation was done with four reconstruction volumes:

A. *PNN* with hole-filling 5x5x5
B. *Hybrid* approach with Rmax 8.0 and *gaussian* weighting
C. Existing volume from the *VGDW* algorithm[9]
D. *Hybrid* approach with Rmax 8.0 and *linear* weighting

Where volume C is used as a gold standard of a good evaluation. It is used as we had access to the same input data as used in this reconstruction.

Six technicians responded to the query, four responded with a single number for each reconstruction, and two responded with scores given to each single image. Some written feedback was also acquired by some of the respondents.

3.8.3 Evaluation of performance

A performance evaluation was performed to get an accurate look at the runtime of the implementation, and to compare different settings. Two sets were used, Set 2 with straight overlapping scans, and Set 3 with a steep fan scan curve at one end. Eight configurations, of algorithm and settings, were compared, as listed in Table 3.2.

Table 3.2: Performance evaluation - Algorithm setting descriptions

Short:	**Algorithm:**	**Vol. size:**	**Extra:**
Gauss8	Hybrid	32M	Gaussian weighting, Rmax = 8.0
Lin8	Hybrid	32M	Linear weighting, Rmax = 8.0
Lin8-256	Hybrid	256M	Linear weighting, Rmax = 8.0
Lin16	Hybrid	32M	Linear weighting, Rmax = 16.0
AlphaLin8	Hybrid	32M	Alpha-blending*, linear weighting, Rmax = 8.0
PNN5	PNN	32M	5x5x5 Hole-Filling
PNN7	PNN	32M	7x7x7 Hole-Filling
PNN7-256	PNN	256M	7x7x7 Hole-Filling

**Alpha-blending here refers to alpha-blending compounding as described in Section 2.5.6.3.*

All configurations were run with a OpenCL GPU implementation. DV, as described in Section 2.5.1, was set to 1.0.

Time was recorded across three different spans to compare the different algorithms and settings, and give a basis for comparison to other algorithms. Each span reflects a different qualities of the algorithms. The spans are described in Table 3.3

Table 3.3: Performance evaluation - Timing span descriptions

Short name:	**Full name:**
Inner loop	Algorithm inner accumulation loop
Measures the time used to accumulate input data, by measuring the time used in accumulating each frame into the volume. Ignores any initializing steps and normalization. Good for checking how much time is used to add a single frame, and to measure the time of the actual work of the algorithm.	
Norm	Normalization step (incl. Hole-filling)
Measures the time used to normalize the accumulated volume to a ready-to-display scaled volume. For PNN algorithms the Hole-filling step is also performed as part of this step. Good for checking how much time is used to finalize the volume for display, an attribute that strongly varies with different approaches. Also valuable information in regards to real-time opportunities.	
Init-to-norm	Initialization-to-normalization ("*total time*")
Measures the "*total time*" used by the algorithms, starting from initialization and finishing after normalization. Loading of images is omitted as the loading process in FAST is currently nonoptimal. Good for measuring total performance of the algorithms, as well as expected total runtime.	

Only the inner loop and normalization is optimized for speed, while initialization and general steps are focused on modifiability, and can run all the used algorithms. Accumulation process and normalization might also have room for improvement in working groups, balancing the work item work load with kernels.

Table 3.4: Performance evaluation - Procedure

1. Restart the computer
2. Stop any unnecessary programs running in the background
3. Run .exe file running a set of tests and displaying runtimes
    - 3.1. Disabling OpenCL kernel cache ('CUDA_CACHE_DISABLE=1')
    - 3.2. Running for each of the two input set
        - 3.2.1. Running with each algorithm configuration
            - 3.2.1.1. A new instance is set up and run, reconstructing the volume
            - 3.2.1.2. Reconstruction times are recorded, as described in Table 3.3
        - 3.2.2. Repeat step 3.2.1 a total of 3 times per algorithm
    - 3.3. All run times are gathered and stored, the program is closed

4. Repeat step 3 a total of 3 times per restart
5. Repeat one more time from step 1 to 4, for a total of two clean runs

To acquire the timing data, the steps of Table 3.4 were taken. All the data was stored in a spreadsheet and evaluated by averaging the results for each timing in each algorithm specification in each set. Timing is performed by *high_performance_clock* of *boost::chrono*, wall time measurement that acted consistently even on the smallest values. All timings are surrounded by cl_finish commands, to capture the exact workload done within these section. The computer specifications used in this evaluation are listed in Table 3.5.

Table 3.5: Performance evaluation - Computer specifications

Property:	**Specification:**
Form factor	Full size stationary desktop computer with fan-based air cooling
Operating system (OS)	Windows 8.1 64-bit, x64-based processor
CPU	Intel(R) Core(™) i5-4460 CPU @ 3.20GHz
Memory (RAM)	16 GB
GPU #1	NVIDIA GeForce GTX 970 w/ 4GB GDDR5 memory
GPU #2	NVIDIA GeForce GTX 970 w/ 4GB GDDR5 memory
GPU stats	Cores 1664 - TMUs: 104 - ROPs: 56
GPU Driver version	R368.39 (r368_35-3) / 10.18.13.6839
OpenCL version	OpenCL 1.2
OpenCL stats	GeForce GTX 970 compute units: 13@1177MHz Work Item Sizes: 1024x1024x64 Work group size: 1024
OpenCL memory	Global mem. 4GB Constant buffer 64KB Local mem. 48KB
OpenCL SDK version	NVIDIA GPU Computing Toolkit (CUDA) v.7.5

Chapter 4: Results

In this chapter the results of the image quality evaluation and the performance evaluation will be presented. Section 4.1 presents the results of the visual quality evaluation together with the images used for this evaluation. Results are presented numerically and in graphs. Section 4.2 will thoroughly display the results of the performance evaluation, with the initial measurements, time per frame and time per million voxel. Numerical data is presented, side-by-side of graphs.

4.1 Evaluation of visual quality

Here we will present the results of the evaluation of visual quality, as described in Section 3.8.2. Section 4.1.1 introduces the results, while Section 4.1.2 display the images used for evaluation.

4.1.1 Results

The results are presented in Table 4.1. Evaluators are numbered #1 to #6 and each of their scores are listed on the row with their number, under the four options. The average value of all scores are calculated, as well as an average ranking and the mean. Evaluator #2 stated he could have stretched the scale out more. Evaluators #4 and #5 rated each individual image, so the results are averaged into one score for each reconstruction method.

Hybrid linear weighting got the best score from all evaluators, right in front of hybrid gaussian weighting, as seen in Table 4.1 and Figure 4.1. PNN reconstruction is the outlier, with much lower scores than the other three methods. Looking at Figure 4.2 there is an obvious consistency in how the evaluators have ranked the methods with their scores.

Table 4.1: Evaluation of visual quality - Results

Option:	**A**	**B**	**C**	**D**
Algorithm:	**PNN 5x5x5**	**Hybrid Gauss8**	**VGDW**	**Hybrid Lin8**
#1	4.00	7.00	6.00	8.00
#2*	4.50	5.20	5.00	5.30
#3	6.00	8.00	7.50	8.50
#4**	3.46	4.69	4.23	5.15
#5**	6.31	6.92	6.77	7.15
#6	3.00	7.00	5.00	7.00
Average:	**4.54**	**6.47**	**5.75**	**6.85**
Ranking*:**	**4.00**	**1.92**	**3.00**	**1.08**

The table shows the scores ranking from 0.0 to 10.0, where 10.0 is the best. Hybrid 'Lin8' is scored highest by all evaluators, and the hybrid 'gauss8' and reference reconstruction VGDW both score satisfactory for use. The PNN implementation is scored a bit worse, which will be discussed in Section 5.1.

** Evaluator #2 stated he could have stretched the scale out more.*
*** Evaluator #4 and #5 rated each image individually, and it is not sure the average is representable regarding the score of 5.00+ for usable results overall.*
**** Ranking is the average of the rankings of each individual evaluator. Best is 1, second is 2 and so on. In the case of a tie, like two first places, both are given 1.5, the average of the rankings they would otherwise represent.*

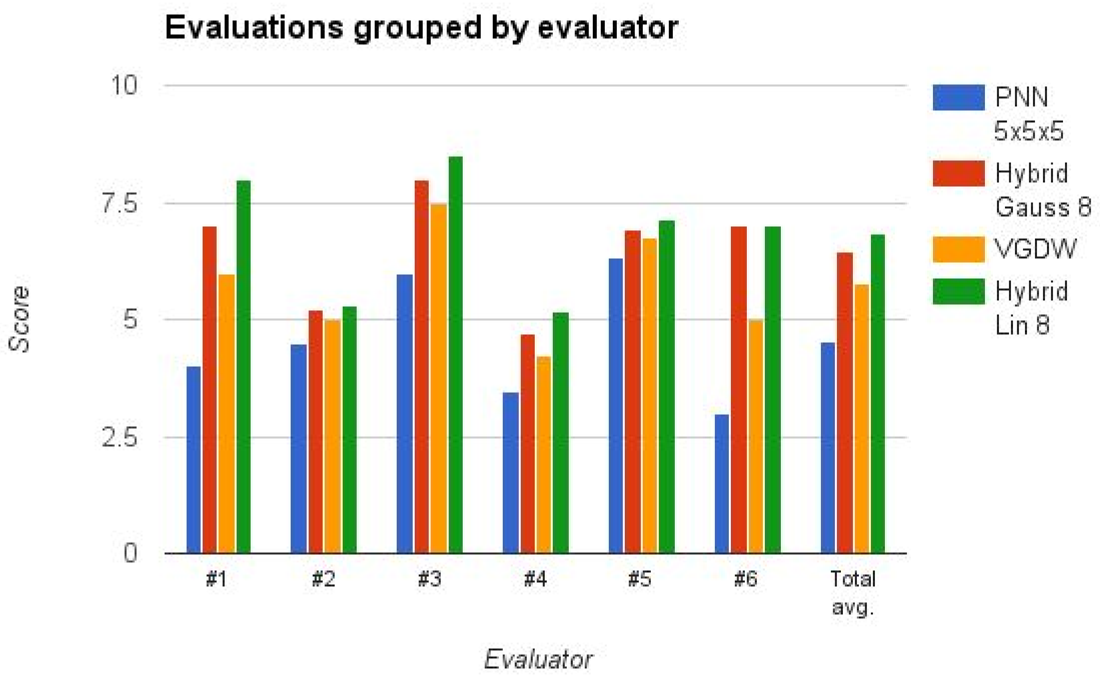


Figure 4.1: Evaluation scores grouped by evaluator.
Higher score is better. More details in Table 4.1.

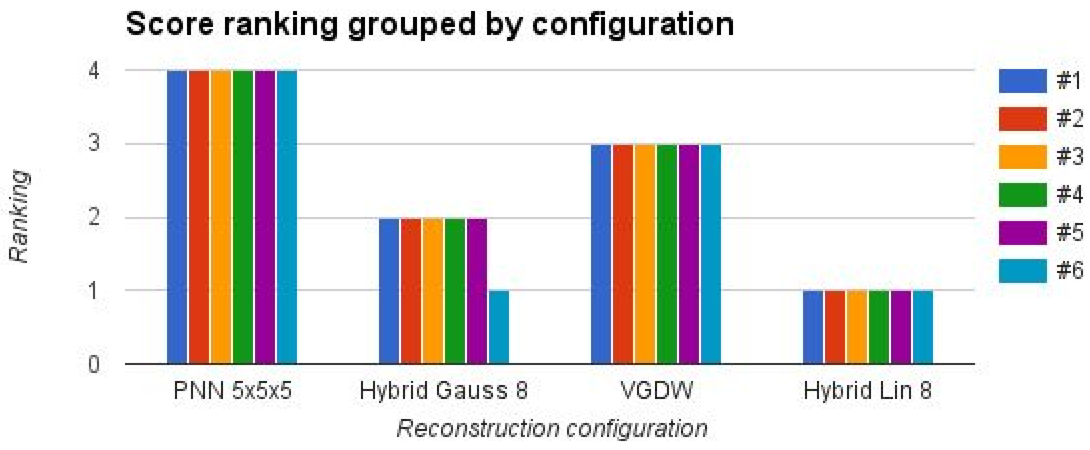


Figure 4.2: Score ranking grouped by configuration.
Lower ranking is better. Hybrid 'lin8' is scored best by all evaluators, and the ranking order of the four reconstructions are strikingly similar. See Table 4.1 for details.

4.1.2 Evaluated Images

Below the images presented to the evaluators are shown. Each image represents a full page, and were probably bigger than presented here. Set 1 used two views, set 2 three, set 3 three, set 4 two, and set 5 used three views.

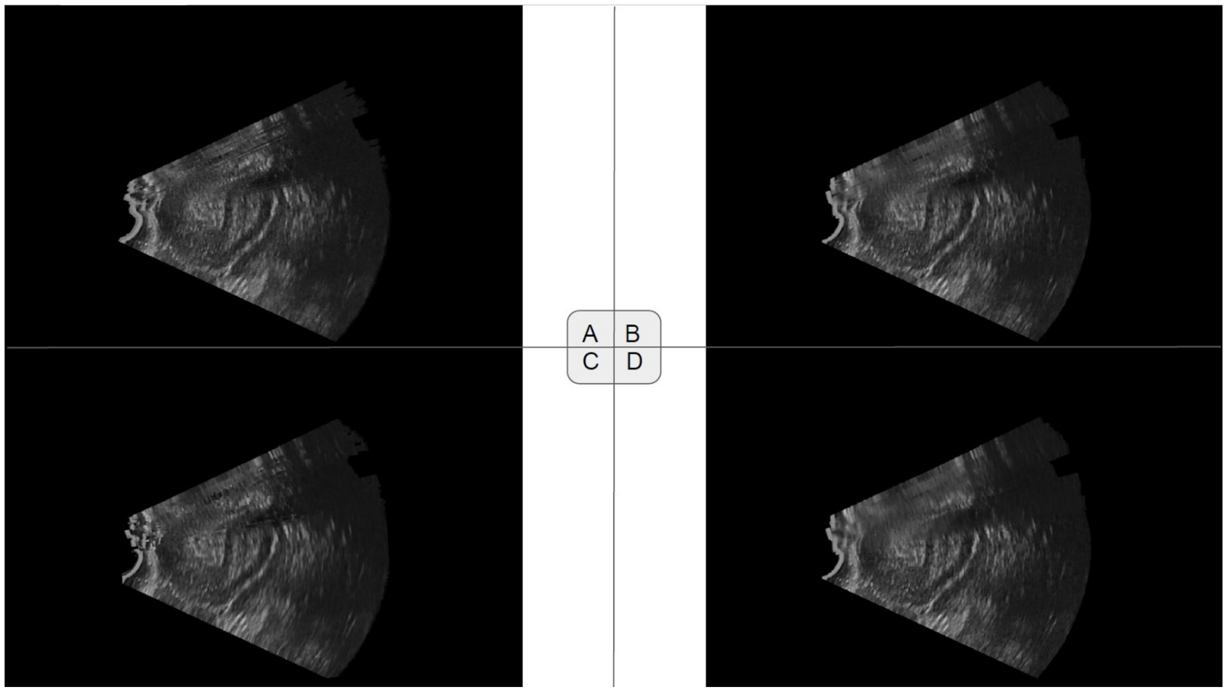


Figure 4.3: Set 1 - View 1: Coronal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

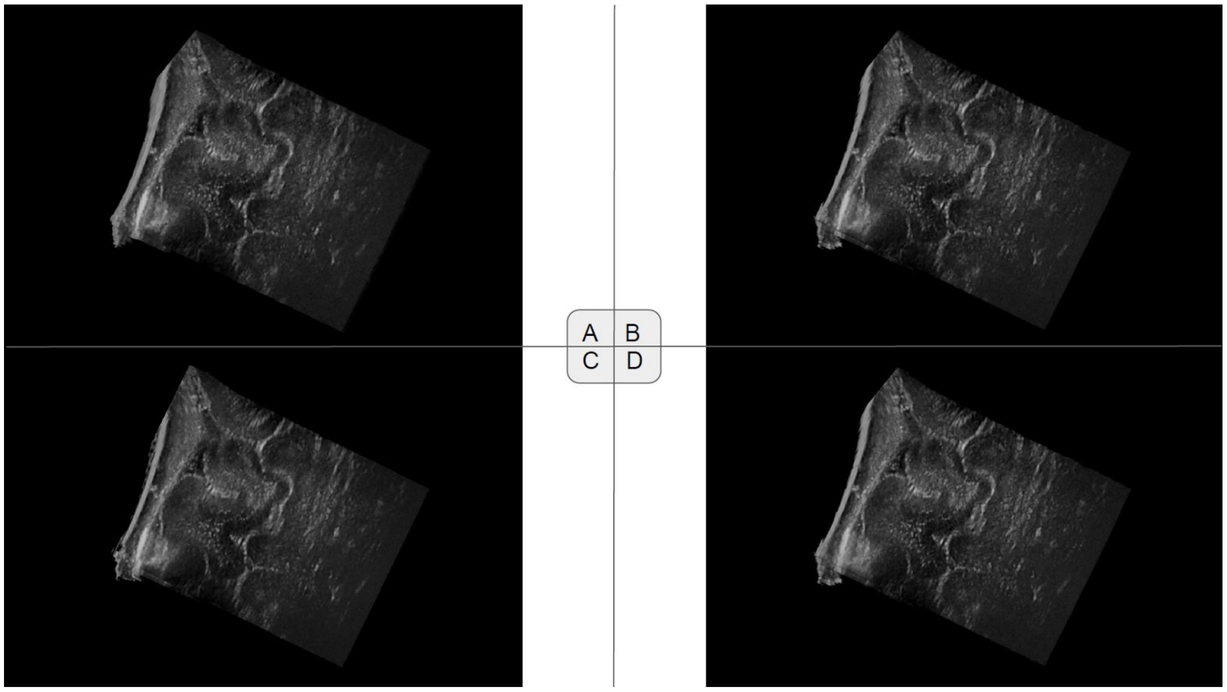


Figure 4.4: Set 1 - View 2: Axial
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

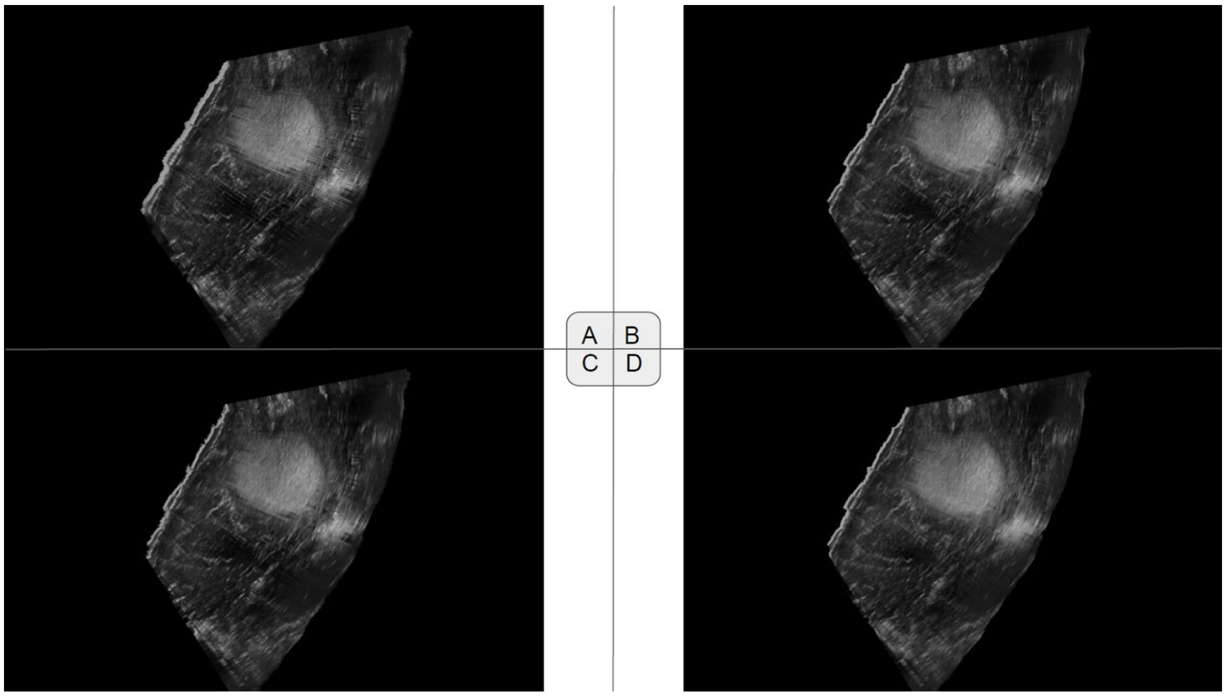


Figure 4.5: Set 2 - View 1: Coronal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

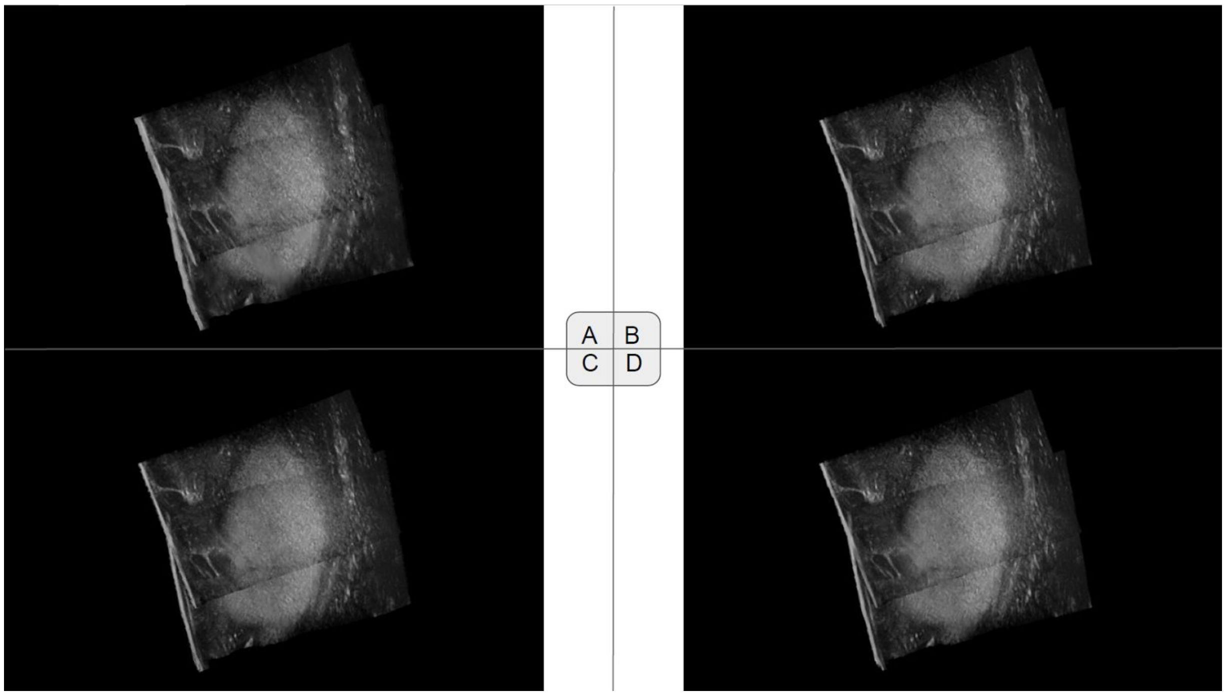


Figure 4.6: Set 2 - View 2: Axial
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

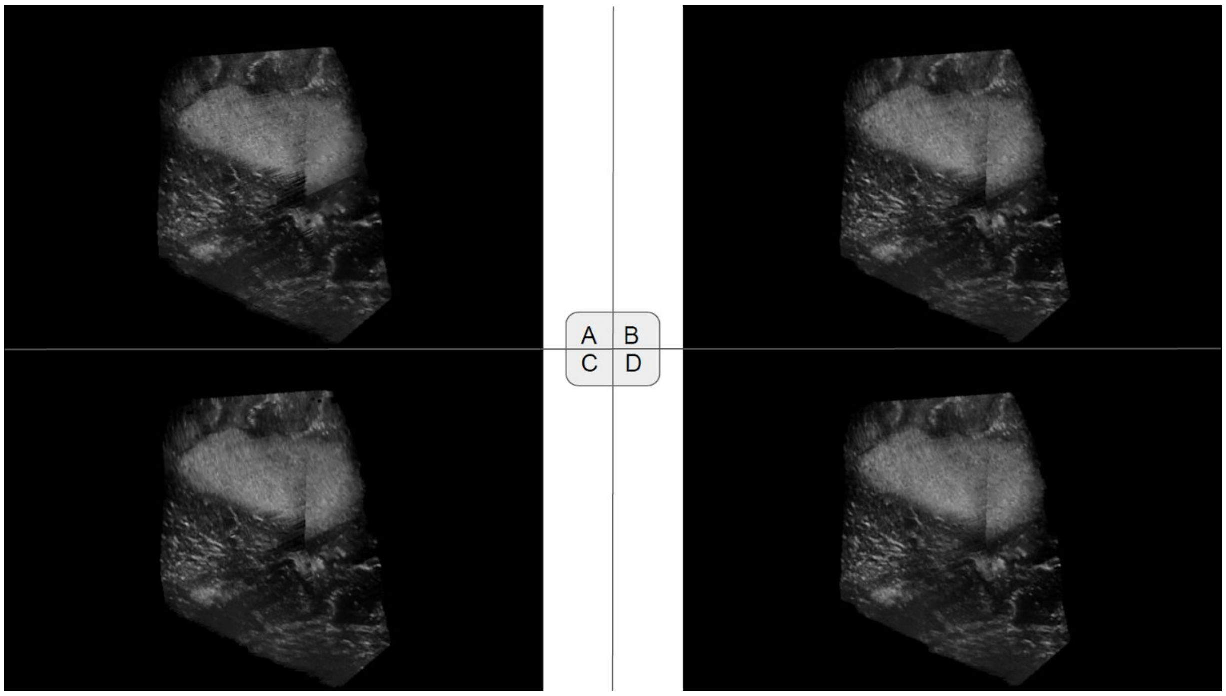


Figure 4.7: Set 2 - View 3: Sagittal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

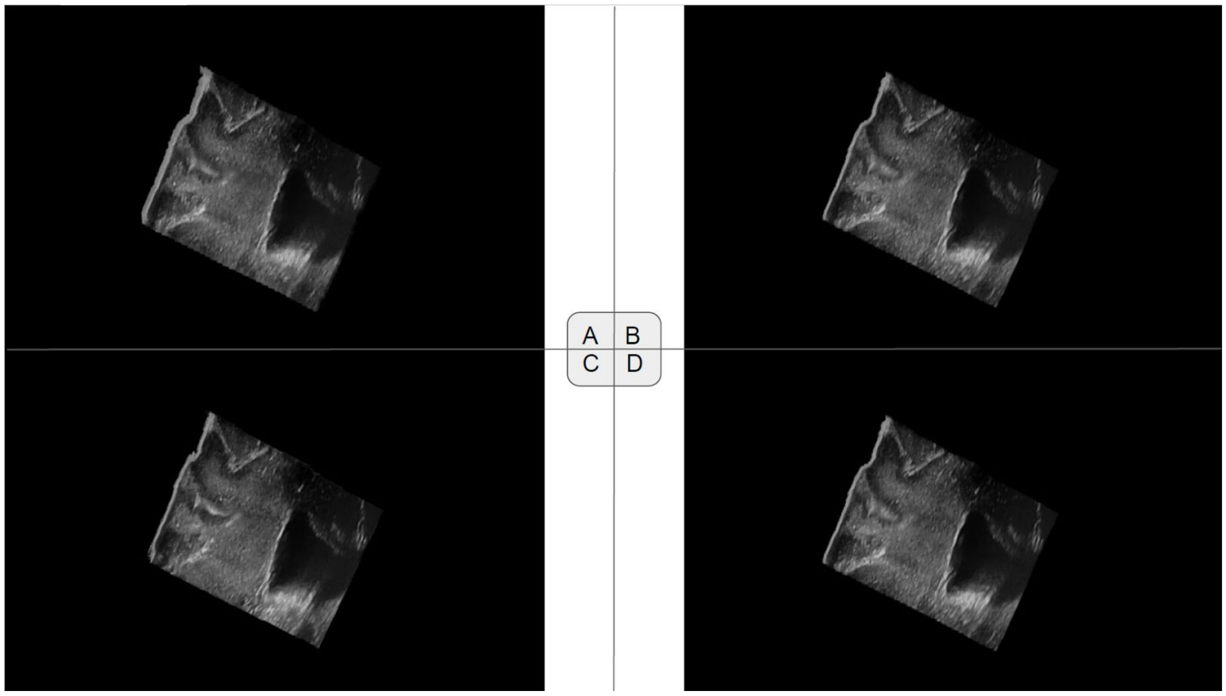


Figure 4.8: Set 3 - View 1: Coronal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

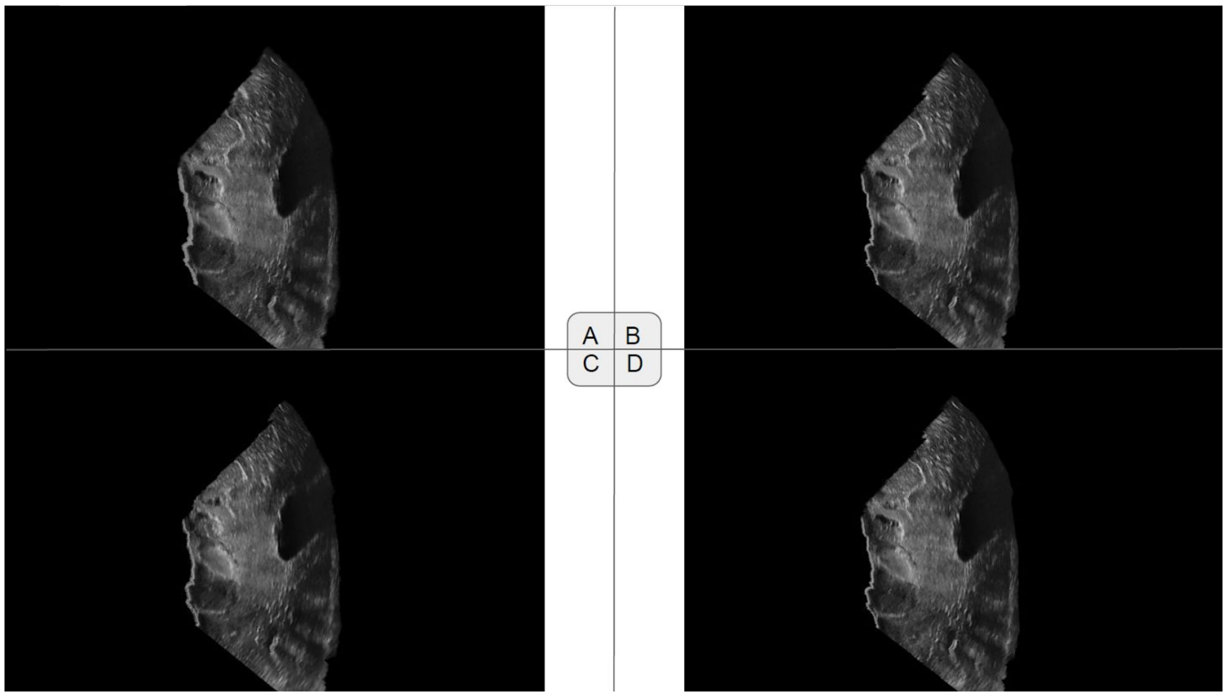


Figure 4.9: Set 3 - View 2: Axial
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

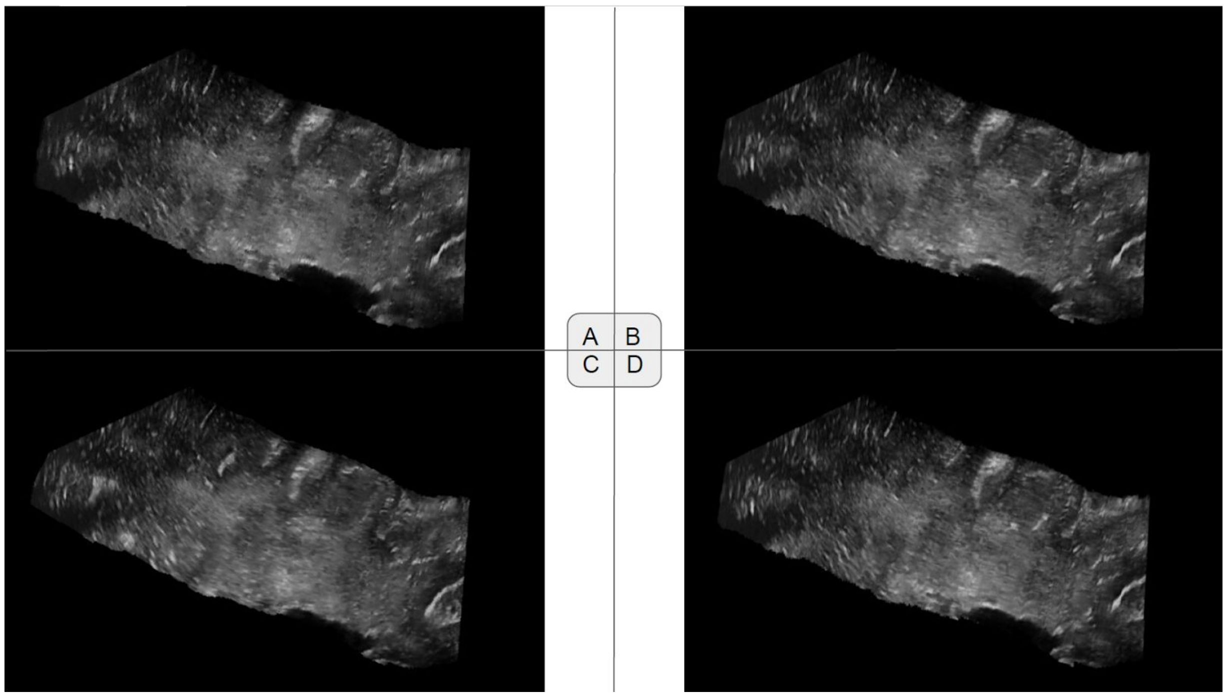


Figure 4.10: Set 3 - View 3: Sagittal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

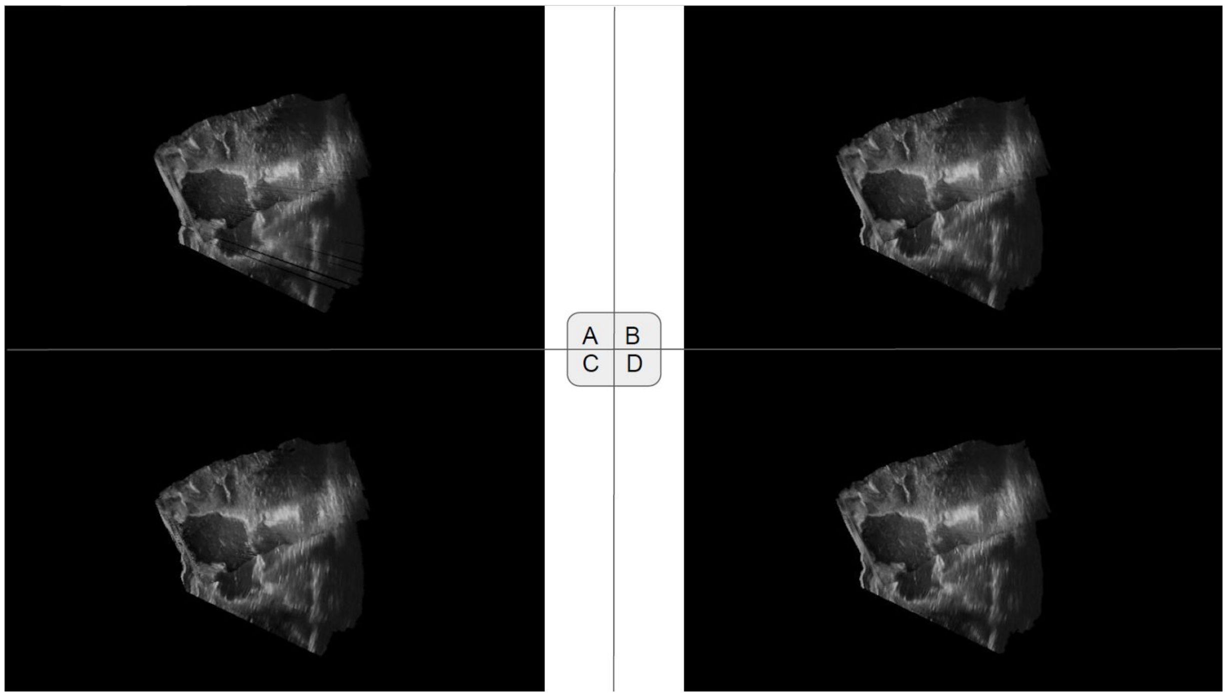


Figure 4.11: Set 4 - View 1: Axial
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

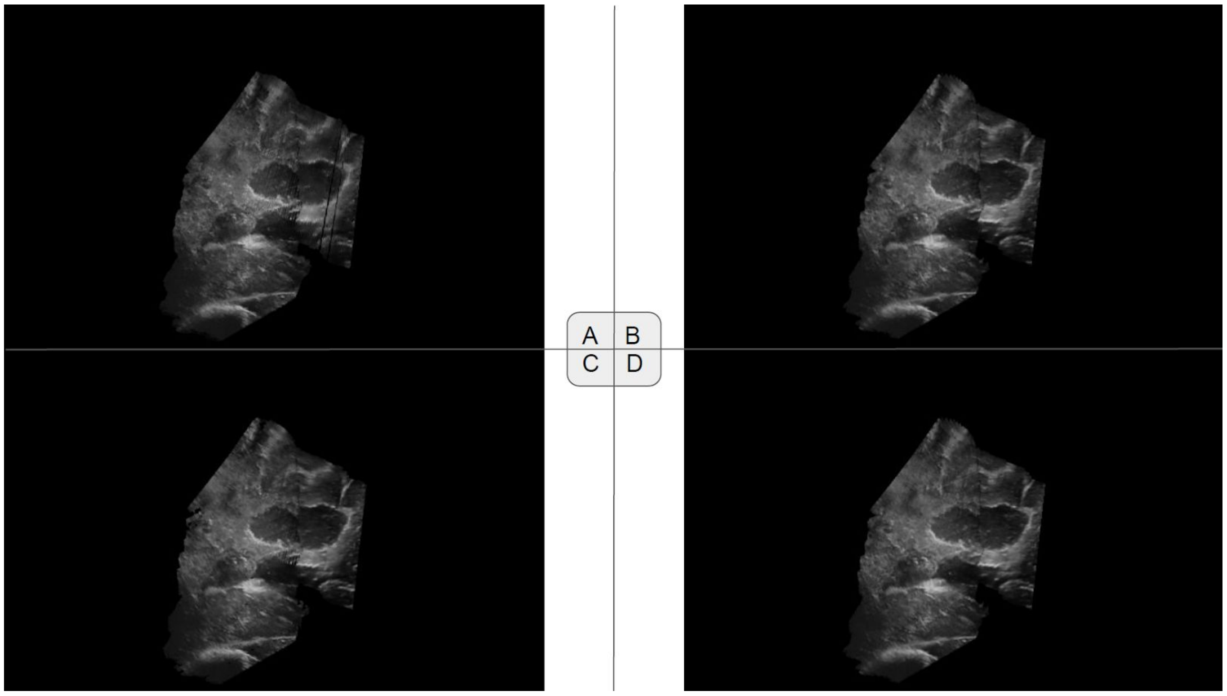


Figure 4.12: Set 4 - View 2: Sagittal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

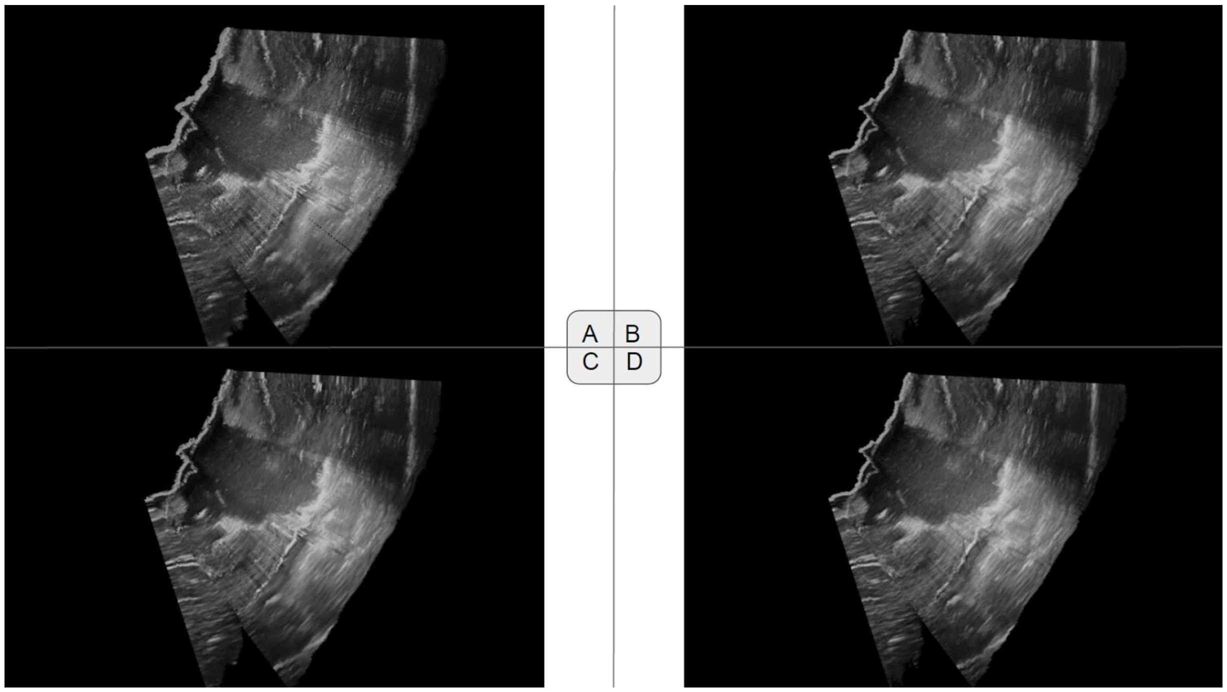


Figure 4.13: Set 5 - View 1: Coronal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

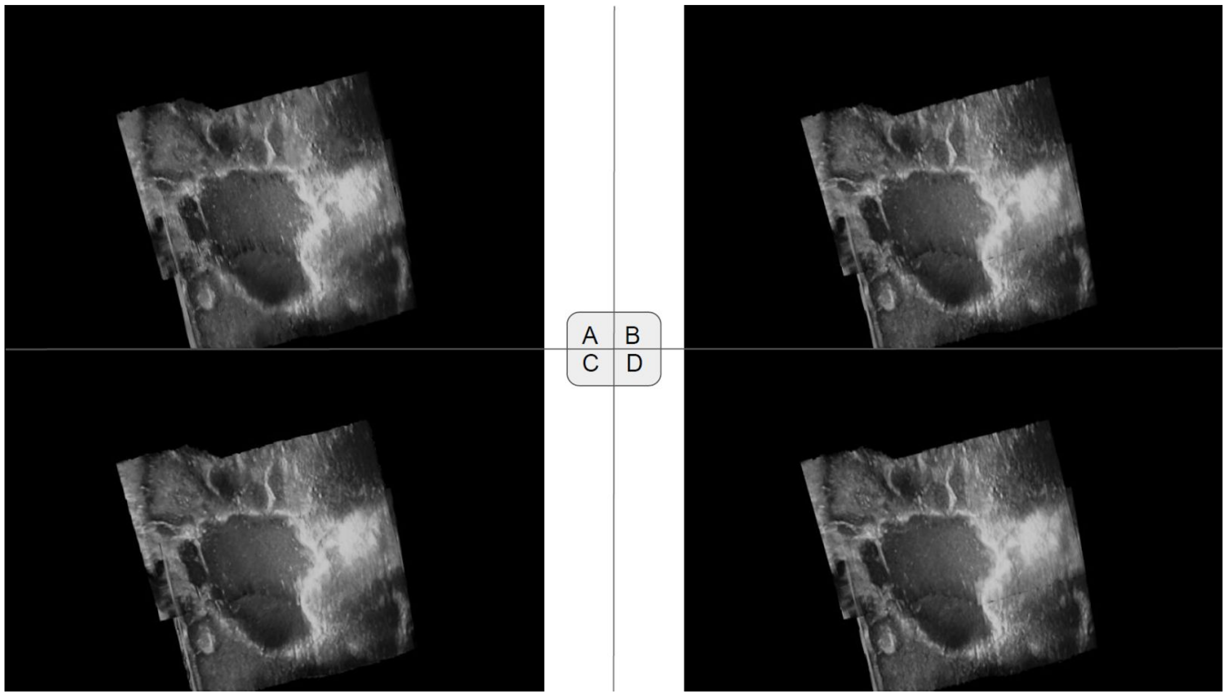


Figure 4.14: Set 5 - View 2: Axial
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

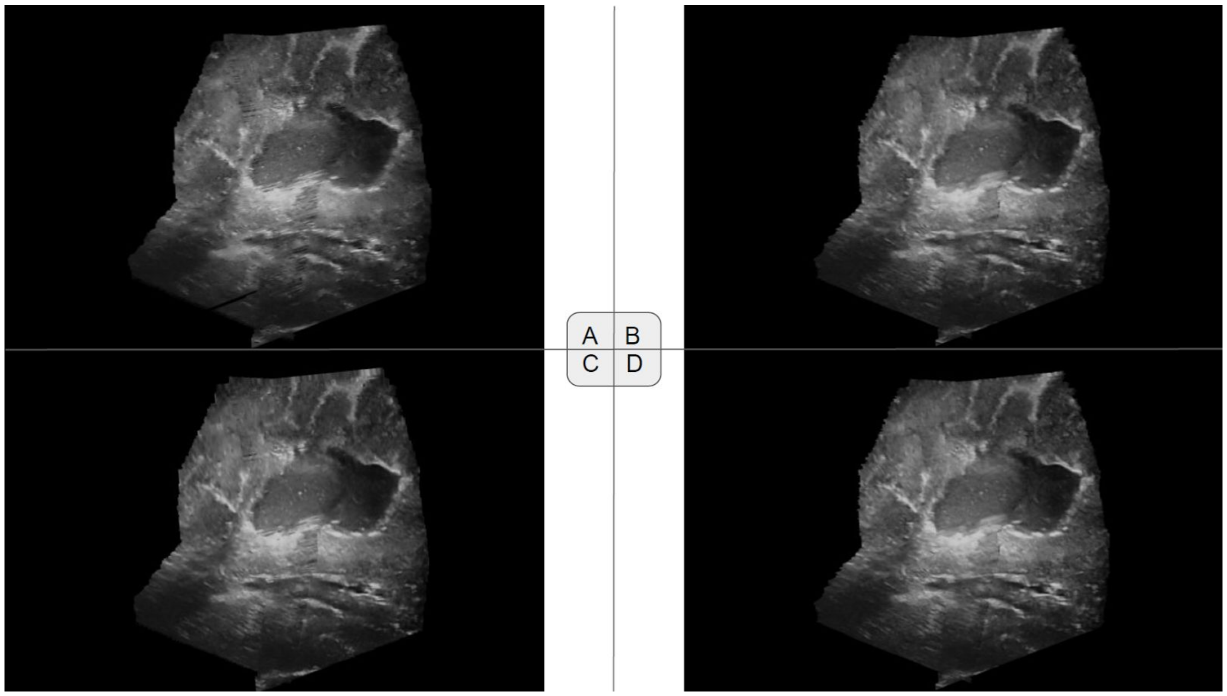


Figure 4.15: Set 5 - View 3: Sagittal
Top: PNN (A), Hybrid with Gaussian (B); Below: VGDW (C), Hybrid with Linear (D)

4.2 Evaluation of performance

In this section we will present the results of the evaluation of performance, as described in Section 3.8.3. Results are divided into three sections, one for each timing described in Table 3.3. The processing of the results are described in Section 4.2.1, from the relative comparison to the baseline method, to two timing adjustments.

Section 4.2.2 introduces results of the algorithm accumulation loop, depicting the time spent on just accumulation. Section 4.2.3 display the results of the normalization step, performing the finalizing steps of normalization of pixel values according to weights, and hole-filling where necessary. Section 4.2.4 shows the performance results of the total time, initialization-to-normalization, with loading times omitted.

4.2.1 Performance: Result processing

The results were grouped for each combination of input set and algorithm configuration, and the measured runtimes were averaged over each group. This average is called *Runtime timing* later in Section 4.2, and is used as a baseline for the other calculations.

Accumulation loop, and initialization-to-normalization results were adjusted for number of frames of each set. Set 2, as described in Section 3.8.1.3, contained 660 frames, while set 3, as described in Section

3.8.1.4, contained 573 frames. Runtimes were divided by these numbers to align the runtimes of the two sets. In addition this number shows the work done by accumulating a single frame.

Normalization, and initialization-to-normalization results were adjusted for number of voxels in the volume. Most configurations ran at 32M, while two ran at 256M. Runtimes were divided by millions of voxels in the volumes to compare how they scale.

Each set of results is displayed with a figure and a table. The table contains a row for the timings of both set 2 and 3, as well as two rows comparing the timings relatively to the baseline configuration, '*lin8*'. '*Rel2*' is set 2 performance relative to the baseline set 2 performance, and the same for '*Rel3*' and set 3. All runtimes are in milliseconds, and the numbers used for calculation are more accurate than displayed in the tables.

4.2.2 Performance: Algorithm inner accumulation loop

The performance results of the algorithm inner accumulation loop are displayed in this section, which is the frame accumulation without any overhead, as described in Table 3.3. Total accumulation time for the configurations are shown in Figure 4.16, and Table 4.2, which gives a good picture of accumulation workload of the reconstructions. Timing per frame is shown in Figure 4.17 and Table 4.3, giving a picture of the workload for the average incremental reconstruction step.

Total runtime of accumulation appears to stay under 300 ms for all configurations except '*lin8_256*', which is scaled up with a multiplier of less than *x2*, which is not bad for the size. '*Lin16*' is as fast as '*lin8*', and these are the fastest configurations. Adjusted for number of frames, the accumulation stay under 0.40 ms for the faster configurations. This corresponds to an accumulation speed of 2500 frames/s.

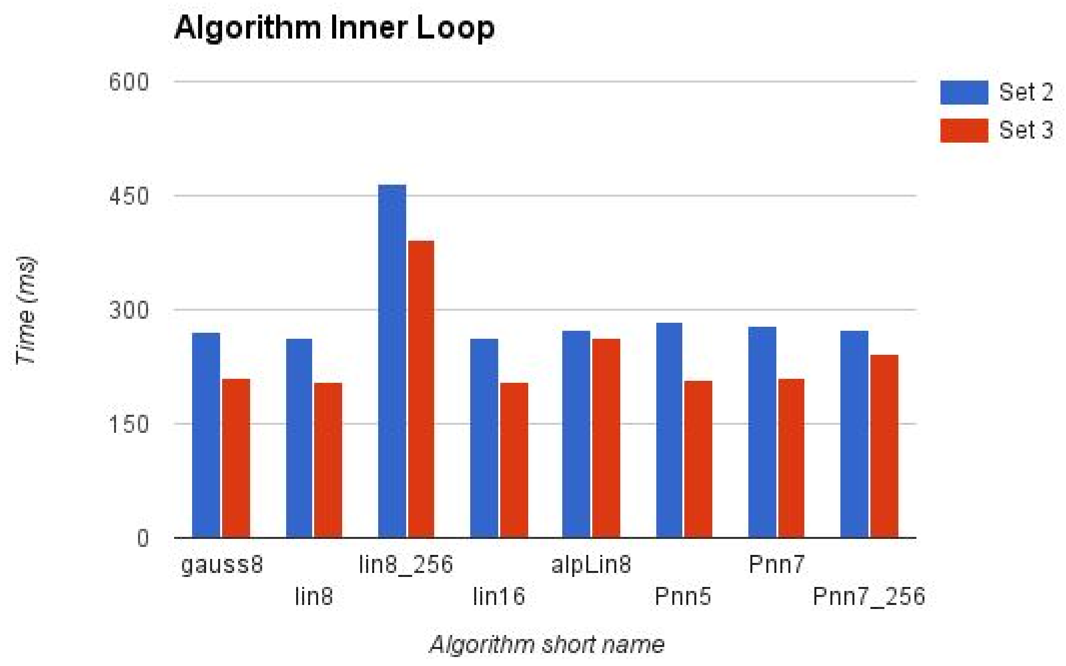


Figure 4.16: Runtime timing - Algorithm accumulation loop.

Table 4.2: Runtime timing (ms) - Algorithm accumulation loop.

	gauss8	***lin8***	**lin8_256**	**lin16**	**alpLin8**	**Pnn5**	**Pnn7**	**Pnn7_256**
Set 2	270.97	***262.57***	465.79	262.72	274.61	284.94	278.89	274.67
Set 3	209.22	*205.89*	392.33	**204.00**	263.33	208.67	210.89	241.06
Rel 2	103%	***100%***	177%	**100%**	105%	109%	106%	105%
Rel 3	102%	*100%*	191%	**99%**	128%	101%	102%	117%

'Lin8' is the baseline for relative(Rel) performance. The fastest measurements are shown with a light background.

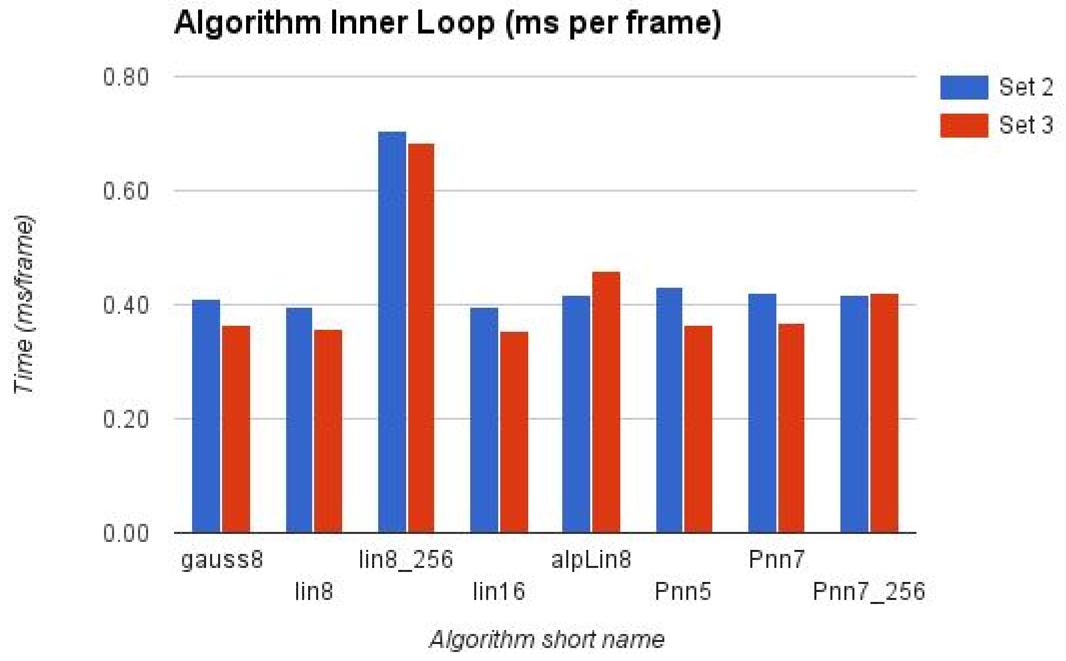


Figure 4.17: Time per frame - Algorithm accumulation loop

Table 4.3: Time per frame (ms) - Algorithm accumulation loop.

	gauss8	***lin8***	lin8_256	lin16	alpLin8	Pnn5	Pnn7	Pnn7_256
Set 2	0.41	***0.40***	0.71	**0.40**	0.42	0.43	0.42	0.42
Set 3	0.37	***0.36***	0.68	**0.36**	0.46	**0.36**	0.37	0.42
Rel 2	103%	***100%***	177%	**100%**	105%	109%	106%	105%
Rel 3	102%	*100%*	191%	**99%**	128%	101%	102%	117%

'Lin8' is the baseline for relative(Rel) performance. The fastest measurements are shown with a light background.

4.2.3 Performance: Normalization

The performance results of the normalization step is displayed in this section. Gives a good understanding of the work each configuration needs to do to be ready-to-display. Figure 4.18 and Table 4.4 show total normalization times, while Figure 4.19 and Table 4.5 present the normalization times, adjusted for volume size.

The normalization times range from 3 to 1432 ms. Hybrid configurations are significantly faster than their PNN counterparts, with no need for hole-filling. Due to this difference, hybrid configurations are displayed with their own figures below, in addition to the total ones. The three 32M hybrid approaches clock in just around 3.3 ms, while the '*Pnn7_256*' uses 434 times as much at approximately 1450 ms. Adjusted for volume sizes it is clear the similar configurations like '*lin8*' and '*lin8_256*', and '*Pnn7*' and

'*Pnn7_256*' are close in timing. The hole-filling of the PNN appear to scale significantly with increased neighbourhood. The hybrid configuration '*alpLin8*' is ready normalized by accumulation, and needs no normalization. It is therefore omitted when calculating the best runtime below.

Table 4.4: Runtime timing (ms) - Normalization.

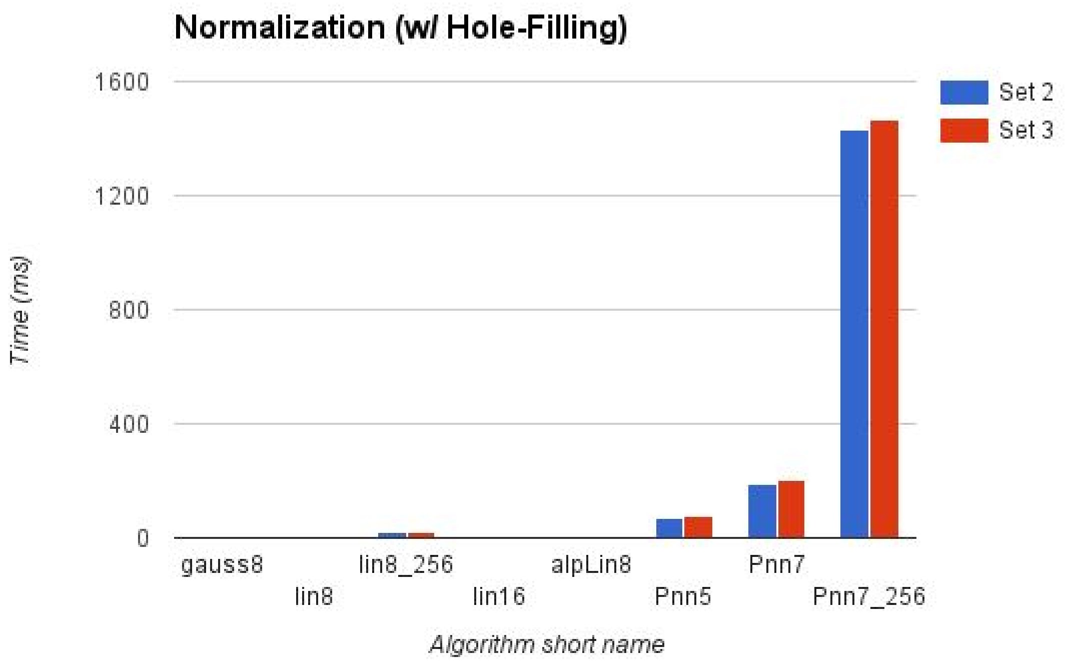
	gauss8	***lin8***	**lin8_256**	**lin16**	**alpLin8**	**Pnn5**	**Pnn7**	**Pnn7_256**
Set 2	3.32	*3.30*	21.47	**3.28**	-	72.02	188.93	1,431.55
Set 3	**3.23**	*3.24*	20.85	3.25	-	76.16	205.02	1,469.13
Rel 2	101%	*100%*	651%	**99%**	-	2183%	5726%	43388%
Rel 3	**100%**	***100%***	644%	**100%**	-	2353%	6333%	45382%

'Lin8' is the baseline for relative(Rel) performance.The fastest measurements are shown with a light background. Configuration 'algLin8' requires no normalization.

Table 4.5: Time per million voxel (ms) - Normalization.

	gauss8	***lin8***	**lin8_256**	**lin16**	**alpLin8**	**Pnn5**	**Pnn7**	**Pnn7_256**
Set 2	0.10	*0.10*	**0.08**	0.10	-	2.25	5.90	5.59
Set 3	0.10	*0.10*	**0.08**	0.10	-	2.38	6.41	5.74
Rel 2	101%	*100%*	**81%**	99%	-	2183%	5726%	5423%
Rel 3	100%	*100%*	**81%**	100%	-	2353%	6333%	5673%

'Lin8' is the baseline for relative(Rel) performance. The fastest measurements are shown with a light background. Configuration 'algLin8' requires no normalization.



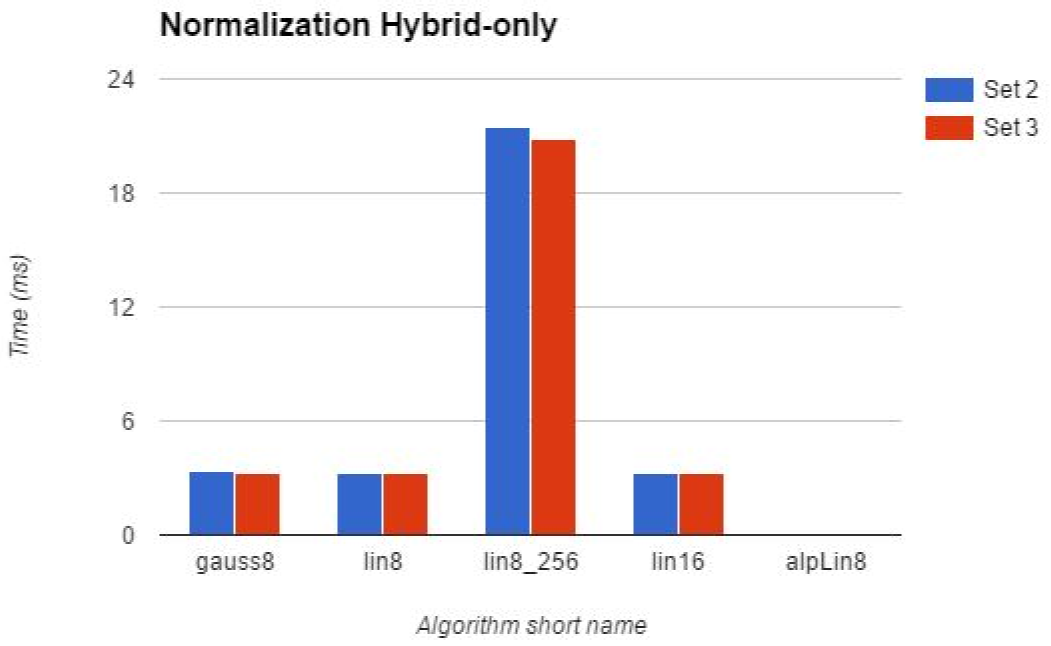
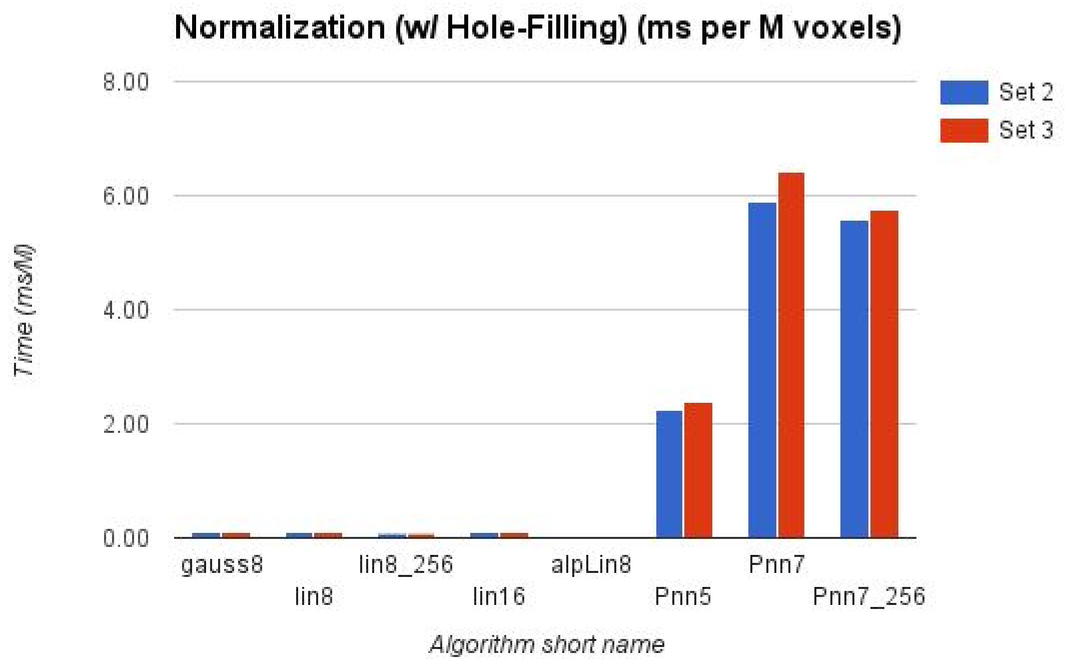


Figure 4.18: Runtime timing - Normalization. Top: All normalization methods compared stretches the scale, so hybrid methods are almost invisible. Bottom: Hybrid-only chart to distinguish between configurations. Configuration 'alpLin8' has no need to perform any normalization and uses zero time.



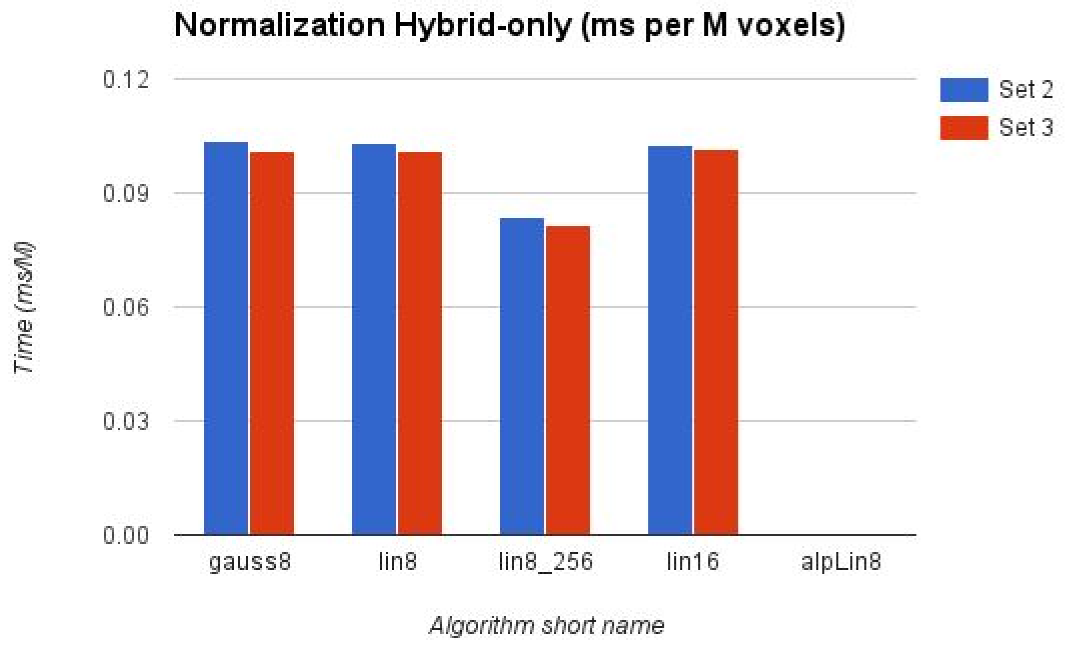


Figure 4.19: Time per million voxel - Normalization. Top: all normalization. Bellow: Only hybrid configurations. Notice how 256M reconstruction volumes now come on line with the corresponding 32M volumes.

4.2.4 Performance: Initialization-to-Normalization

The performance results of the initialization-to-normalization timing is displayed in this section, giving a total timing to rate the overall runtime of the algorithms. Figure 4.20 and Table 4.6 present the actual timings. Figure 4.21 and Table 4.7 adjusts for frames, giving us a rough estimate of the runtime of each frame. Figure 4.22 and Table 4.8 adjusts for volume voxels, giving a good basis for scaling and throughput.

Total runtime of initialization-to-normalization stay under 1 second for most algorithms. '*Pnn7_256*' is far behind after bad normalization results, while '*lin8_256*' hovers around 1 second. '*Lin8*' and '*lin16*' are fastest, but among the 32M reconstructions the timing increase is roughly just 30% from best to worst. The 32M timings are just above 1 ms/frame, and the frame count adjustment seems to even out the differences between the two sets with the exception of configuration '*Pnn7_256*'. Adjusted for voxel count the 256M volumes perform a lot better than the 32M volumes, implying that the initialization-to-normalization timing does not scale much with increasing volume. The scaling for both Hybrid and PNN approaches turn out at *x1.5* and *x3.6* for the *x8* volume size increase from 32M to 256M.

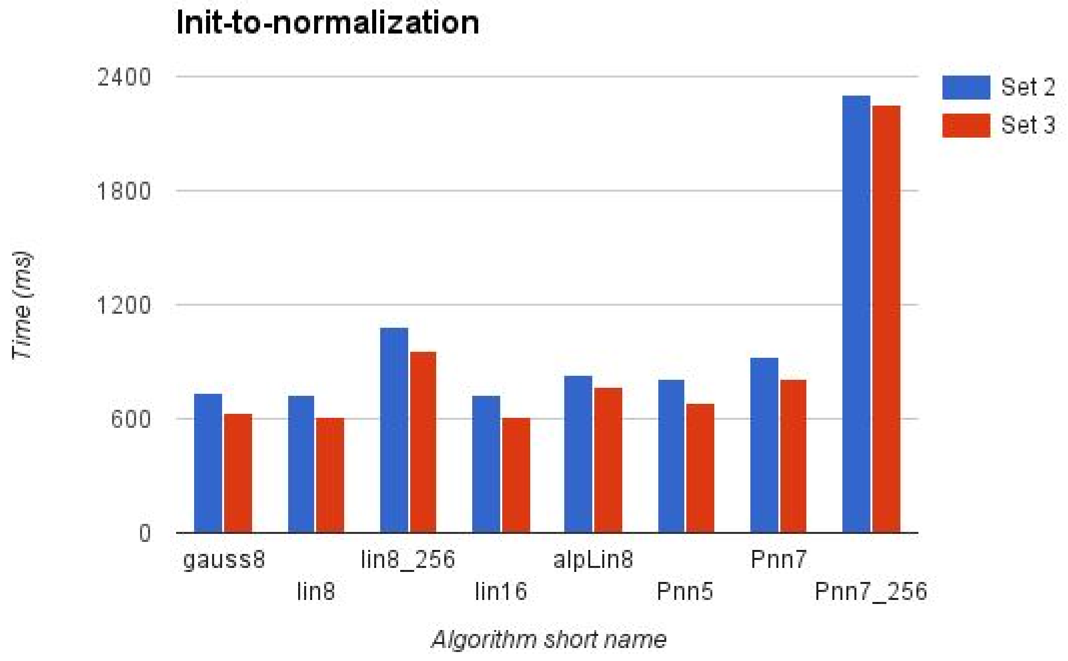


Figure 4.20: Runtime timing - Init-to-normalization

Table 4.6: Runtime timing (ms) - Init-to-normalization

	gauss8	*lin8*	lin8_256	lin16	alpLin8	Pnn5	Pnn7	Pnn7_256
Set 2	739.62	***721.82***	1,088.94	723.44	829.28	807.72	921.33	2,302.94
Set 3	629.00	*615.67*	961.44	**614.56**	768.44	682.44	813.50	2,249.17
Rel 2	102%	***100%***	151%	**100%**	115%	112%	128%	319%
Rel 3	102%	***100%***	156%	**100%**	125%	111%	132%	365%

'Lin8' is the baseline for relative(Rel) performance. The fastest measurements are shown with a light background.

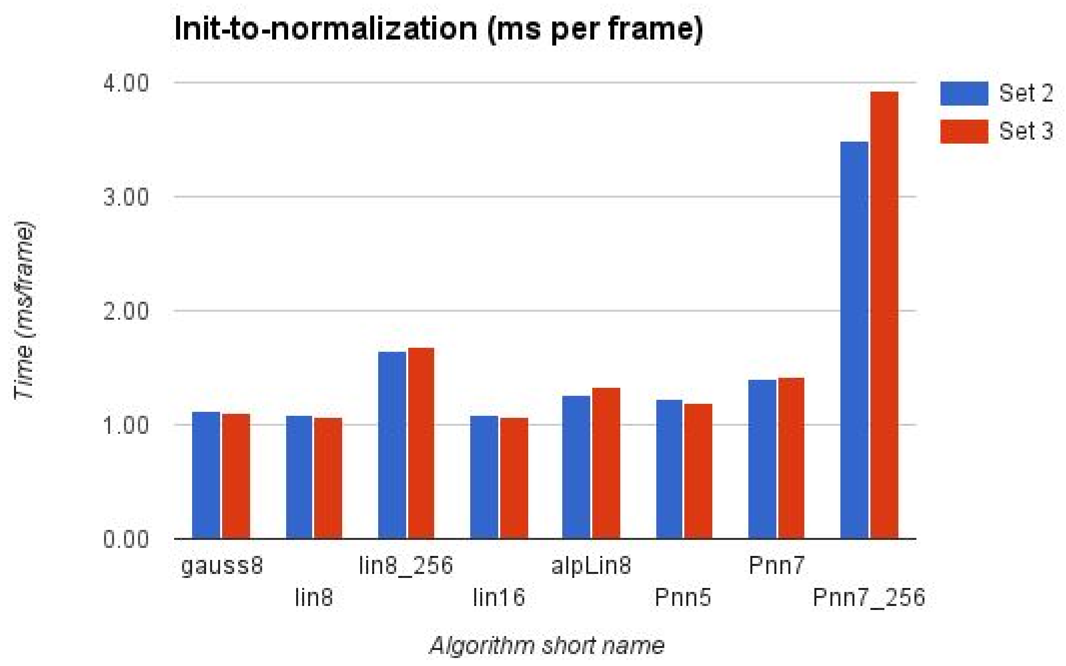


Figure 4.21: Time per frame - Total time. Even results between the two sets, but notice the offset with configuration 'Pnn7_256'.

Table 4.7: Time per frame (ms) - Total time

	gauss8	*lin8*	lin8_256	lin16	alpLin8	Pnn5	Pnn7	Pnn7_256
Set 2	1.12	***1.09***	1.65	1.10	1.26	1.22	1.40	3.49
Set 3	1.10	***1.07***	1.68	**1.07**	1.34	1.19	1.42	3.93
Rel 2	102%	***100%***	151%	**100%**	115%	112%	128%	319%
Rel 3	102%	***100%***	156%	**100%**	125%	111%	132%	365%

'Lin8' is the baseline for relative(Rel) performance. The fastest measurements are shown with a light background. Even results between the two sets, but notice the outlier configuration 'Pnn7_256'.

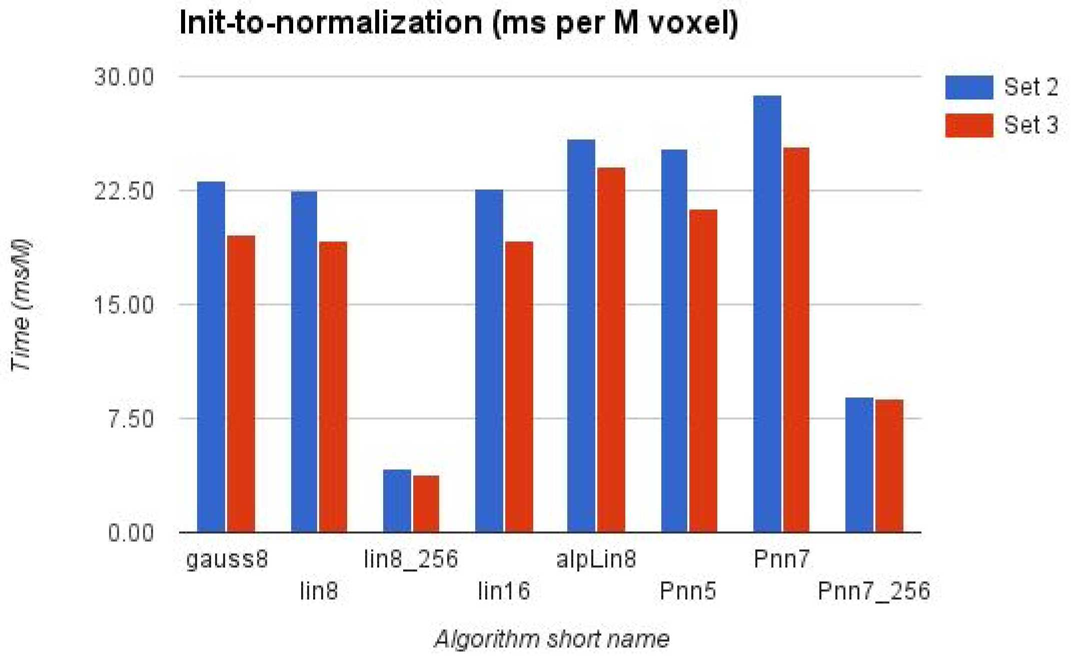


Figure 4.22: Time per million voxel - Total time. Relative to volume size the bigger volumes a lot faster on the total process. Only a few parts relate directly with volume size however.

Table 4.8: Time per million voxel (ms) - Total time

	gauss8	***lin8***	**lin8_256**	**lin16**	**alpLin8**	**Pnn5**	**Pnn7**	**Pnn7_256**
Set 2	23.11	*22.56*	**4.25**	22.61	25.91	25.24	28.79	9.00
Set 3	19.66	*19.24*	**3.76**	19.20	24.01	21.33	25.42	8.79
Rel 2	102%	*100%*	**19%**	100%	115%	112%	128%	40%
Rel 3	102%	*100%*	**20%**	100%	125%	111%	132%	46%

'Lin8' is the baseline for relative(Rel) performance. The fastest measurements are shown with a light background. Relative to volume size the bigger volumes a lot faster on the total process. Only a few parts relate directly with volume size however.

Chapter 5: Discussion

This chapter will discuss the the findings of the results presented in chapter 4. Section 5.1 will review the results of the evaluation of visual quality, and Section 5.2 will discuss the results of the performance evaluation.

5.1 Visual quality

The visual evaluation used tumor input data to let the evaluators score the reconstruction quality in a real world setting. This evaluation was performed to test the achieval of the visual quality goals described in Section 1.1, looking for quality reconstruction, with options for further improvement. Each of the five input sets presented different properties to test multiple effects of reconstruction configuration, from compounding dense, overlapping data, to filling in sparse areas. In input sets 2, 4, and 5, there are two passes of overlapping data, creating an obvious edge where the two slices meet. As with other artefacts in ultrasound imaging there is no single best way to handle this. Some algorithms can in effect smooth them out by applying a wide influence from each pixel, while others will make a reconstruction more directly representing the input images, even if it has less of an visual appeal. This visual appeal of a smoother look seems to have impacted the evaluation results, as both of the two top ranked reconstruction configurations produced this effect.

Many of the images appeared difficult to distinguish, and the general appearance becomes paramount, overshadowing the importance of details. However, none of the images represent details particularly bad, and the strong scores of the two *smoother* algorithms proves that the level of detail is sufficient. Another effect of the importance of general appearance is that images with unfilled gaps appear very faulty, and produce the worst individual image scores in this evaluation. The PNN configuration have very noticeable gaps in the reconstruction of set 4, despite using a 5x5x5 neighbourhood for hole-filling step in a 32M volume, and judging by the single scores of the two evaluators rating each image, this is where they really fall behind the other approaches. The scores of the single ratings also reveal that some images score lower in general, and have a bigger spread between the best and the worst reconstruction. These images tend to see input frames from the side, making all the tracking inaccuracies visible as staggered lines through the volume, as seen in Figure 4.13, and Figure 4.5, as well as in the aforementioned Set 4, in Figures 4.11-.12.

The evaluation scores given in Table 4.1, show that the score ranges and centers, of each evaluator vary. This is understandable, as there is no universally defined way to rank the image quality of 3D reconstructions. Making it score-based rather than rank-based extends this problem. On the other hand, the score-based approach displays that some of the algorithms are very similar in quality, just as expected. Especially the two hybrid-based configurations tend to be rated closely, and the VGDW configuration is never too far away. The two evaluators scoring images individually also have a smaller final range between evaluations, a natural effect of averaging numbers. Evaluator #2 stated that he could have used a wider range, but found input positioning errors to be too significant. It is unclear if all evaluators have related their scores to our proposed minimum rating of 5.0 for usable results, or how they define usable results. In particular it is uncertain if the averaging individual image ratings fairly represent these scores to this minimum rating.

While all results point to satisfactory evaluations for the hybrid algorithm, it should also be noted that the scores could have been higher with a different method of presentation. Slices are acquired from world space planes rather than volume planes or directly aligning the slice with the input frames, which would significantly improve quality. In this evaluation, the slices are taken from as many planes as possible for each set, to show that it can represent all of them.

If we evaluate how the scores of each evaluator ranks the algorithm configurations, there is evidently a high order of consistency, as seen in Figure 4.2. Undoubtedly there is some means of randomness making it this coherent.

It is a delight to see that the two hybrid configurations are ranked first, and second, of the four configurations. I think it is too soon to thereby suggest that they are universally better than the VGDW approach, but we can assume that it performs in line with this approach on overall visual quality. This is surprising, and makes it more than sufficient to cover the goal of this implementation, to acquire decent visual quality at top-end performance, as described in Section 1.1.

It is also surprising to see the linear weighting perform better than the gaussian weighting. This can be caused by an mistakenly high value of the standard deviation, causing the gaussian weight to blur more than it is designed to do. This mistake was discovered after evaluation, and later fixed. As we had no time to reevaluate visual quality, the results of this fix have been added in Appendix A, with a simple weight distribution comparison and visual comparison.

5.2 Performance

Performance evaluation was performed to test the achieval of the goals described in Section 1.1, seeking top-end performance, and work towards real-time capable performance. In short, the evaluation compared all the configurations up against a baseline configuration, hybrid '*lin8*'. This configuration was seen as an optimal configuration, *Rmax = 8.0*, of the default hybrid reconstruction with *linear weighting*, and general compounding. This baseline allows us to compare how changes to different variables affect performance, without exhaustively testing all combination of variables. Both reconstruction algorithms, hybrid and Pnn, are pixel-based approaches, implemented with OpenCL for GPU execution. Working on a similar basis is crucial for making comparable results.

Pixel-based approaches have the advantage that the runtime of each frame is totally independant of how many other frames are to be added, giving them a big advantage over voxel-based approaches that are required to search through input images to find the closest input image.

The first timing is the *accumulation loop*, described with other timings in Table 3.4. The runtime timing, described in Table 4.2, measures the timing of adding the input data to the accumulation buffer. Initially, it is very noticeable how steady the results are in this section, especially after adjusting for input image count. Figure 4.17 and Table 4.3 show the latter results, which makes it clear that all the results revolve around 0.4 ms per frame, with the exception of '*lin8-256*' in a 256M volume at approximately 0.7 ms per frame. The PNN 256M reconstruction however, does exactly the same amount of work as the two PNN 32M in this phase, as just a single datapoint is added for each input pixel, no matter how big the volume is. For something that should be an order of magnitude less workload, the PNN is timed surprisingly close to the Hybrid approaches. Maybe at this rate of processing, the creation and initialization of kernels are significantly more costly than the algorithm, and run times are thereby normalized in this fashion. This can be addressed by increasing the workload of each work item, processing more pixels each, like implemented by Gobbi & Peters[4].

The second timing is *normalization*, measuring the time used to adjust all accumulated pixel values with their respective weights. Timings are found in Tables 4.4-5. This is a purely voxel-based step, and as expected it scales highly with the increased volume sizes. The *x8* increase in volume size from 32M to 256M increases normalization time with a multiplier of *x6.5,* and *x7.2,* for hybrid, and PNN respectively. PNNs are a lot slower in this step due to the fact that they also process hole-filling in the normalization process. Utilizing local memory as an intermediate step between the two improved run time, but this effect is strongly diminished due to local memory restrictments above a neighbourhood of 5x5x5. Hybrid volumes normalize in roughly 3.3ms, with a *x6.5* increase with a 256M volume, and between a *x22* and a *x434* multiplier for PNN solutions relative to the baseline hybrid configuration. Most notably the alpha-blending compounding approach requires no normalization, giving it a big advantage over the other algorithms. Adjusted for number of voxels in the volume, the normalization comes down to estimately 0.1 ms per million voxel for the baseline, slightly faster for the 256M hybrid approach, and between 2, and 6 ms, per million voxel for the PNN approaches.

The final timing is *total timing*, from *initialization-to-normalization*. The baseline configuration initializes the volume, does pre-calculations, accumulates values, and normalizes the volume to a ready-to-display 3D volume in 0.72 sec, and 0.62 sec, for set 2 and set 3 respectively, as seen in Table 4.6. These contain 660 and 573 input images each, which brings it to roughly 1 ms per frame, as seen in Table 4.7. Adjusted for voxels it is performed at approximately 21ms per million voxels, as seen in Table 4.8. Total reconstruction time is shorter than the time used for input acquisition. Again, other reconstruction times are quite similar to the baseline, with the exception of the 256M volumes which seem to increase by roughly the extra time they used in the normalization process, plus some time most likely spent on initializing these volumes. As the '*Pnn7_256*' configuration used approximately 1.45 sec in normalization, it is already far behind the others in the total timing, ending up between *x3.2* and *x3.7* the time used by the baseline. Adjusted for voxels in the volume the 256M reconstructions are however a lot faster per million voxels, implying that the total workload does not scale as fast as the volume size.

In the end, performance evaluation comes down to comparing the methods. This is quite convenient and simple with the baseline, though more configuration points could give a much better overview of the situation. With '*lin8*' as a baseline, '*gauss8*', and '*lin16*' runs at approximately the same time both in accumulation, normalization, and total time. '*Lin16*' has doubled the maximum half width distance, which should result in some modifier up towards 2, but rather it runs at the same run time as the baseline, and sometimes faster. We see that as a sign that the algorithm will not collect any more data point due to this increase, and can be considered *exhausted*. Further, the '*alpLin8*' configuration runs approximately 20% slower than baseline, which puts it right between the two 32M PNN configurations in total run time. Finally, the 256M Hybrid reconstruction is approximately 53% behind baseline, while the 256M PNN 7x7x7 is between a 219% and 265% behind baseline, and between 150% and 177% behind its 32M counterpart PNN.

The hybrid scales at *x1.5* run time from an *x8* increase in volume size, compared to to the PNNs *x2.5*. Gaussian weight runs at the same time as linear weight, and PNNs are together with alpha-blending slightly behind. The only significant outlier is the '*Pnn7_256*' configuration, which runs more than twice as slow as the other 256M reconstruction.

With complete reconstructions in less than a second, and sub-millisecond accumulation of frames, the implementation can definitely satisfy the goals put up for the top-end performance. Alpha blend compounding, which can easily adapt to a real-time use case, impresses with an approximate 0.44 ms per frame for accumulation. This is over 2000 frames/s, far above the 30 frames/s of Gobbi & Peters[4], and the 166 frames/s incremental reconstruction of Dai et al.[2]. With a good incremental visualization, like introduced by Dai et al., our algorithm would easily achieve real-time reconstruction and visualization. On top of this it has lots of room to experiment with quality improving steps, as it is over 10 times as fast as the Dai et al. solution.

Tests were run on a Nvidia GeForce GTX 970, a high-end graphics card. A new graphic card generation is already out, and graphics cards will continue to increase in performance. This will bring us even more opportunities for further implementations, be it real-time, or high quality, solutions.

Chapter 6: Conclusions

Freehand 3D ultrasound reconstruction is an exciting extension to the ultrasound imaging modality. With low cost and high versatility it can be a real challenger to CT and MR imaging, increasing speed and portability at the cost of image quality. The practical value of the reconstruction will improve as better algorithms produce better results. The purpose of this thesis was to implement an reconstruction in FAST, and utilize GPU processing to ensure a quick solution. A not previously published algorithm was used, and it has shown to perform greatly in terms of both performance and quality.

6.1 Conclusions

This thesis introduced a new approach, hybrid, combining the advantages of pixel-based and voxel-based methods, making it highly parallel and incremental. Working with the basis of input frames, but using the volume voxels to insert data allows us to avoid computationally expensive interpolation. It will efficiently fill gaps of varying sizes. This is the first implementation in FAST, testing the capabilities of the framework. A parallel GPU solution was implemented for both PNN and hybrid approaches, and compared in speed and quality. Two weighting schemes, inverse linear and gaussian, and two methods of accumulation, compounding with and without blending, were tested with good results.

A high level of performance was achieved with the GPU implementation, reconstructing a full volume in less than a second. When given an alpha-blending inspired compounding, the algorithm can accumulate input frames at a rate of 2000 frames/s to a ready-scaled output volume, leaving more than enough time for incremental visualization for a real-time result. The hybrid approach scaled significantly better than a PNN with hole-filling with increasing volume sizes, but otherwise no significant performance deviations were found. Alpha-blending compounding performed only slightly slower than corresponding compounding reconstructions, fast enough to make it optimal for real-time solutions with no need for normalization.

It can be difficult to evaluate 3D reconstructions in terms of quality, as it is impossible to define one gold standard, and it can be inaccurate to calculate quality metrics directly from the volume. We tested the algorithms with a visual evaluation performed by ultrasound technicians. The evaluation proved great results for the two hybrid configurations, placing both in front of the reference volume. The simple inverse linear weighting was rated slightly ahead of the gaussian weighting, but might perform better after recent implementation updates.

We can conclude that the hybrid reconstruction algorithm achieves high quality with top-end performance.

Chapter 7: Further work

Throughout this thesis a lot of ideas were thought of, many of which had to be left out for later. We have kept some of the ideas that we find particularly interesting, and will share them in this chapter as thoughts for further work or general inspiration for similar projects.

Section 7.1 explains the importance of mitigating errors of tracking and motion, Section 7.2 addresses the possibility to use raw scan line input, and Section 7.3 explores alternatives for improving weighting functions. Section 7.4 gives some ideas for making the optimal base algorithm, while Section 7.5 rounds it off by discussing further steps in the direction of a proper real-time solution, which this algorithm is more than capable of.

7.1 Correction of errors from tracking and motion

Tracking positions will introduce errors induced both by tracking inaccuracies and motion of the patient. Handling these problems can be a deal-breaker, as the resulting reconstructions are only as good as their input. The fix can be performed either as a pre-processing step on the input data, or maybe as an internal part in the reconstruction algorithm.

Tracking can be smoothed by comparing neighbouring images to each other, and aligning them such that the images overlap more accurately. This will work well when sets are dense, and scan movement directional.

To correct for motion occurring internally in the patient, there are different features one can address. Motion caused by the cardiovascular system can be *gated* to ensure it is in the same phase. Varying pressure from the probe can also cause inaccuracies, but should mostly be handled by the operator.

7.2 Raw scan line input

The hybrid approach is able to use raw scan line input instead of B-scan input images, but the implementation was left out due to limited time. It is possible that this would improve visual quality or performance.

7.3 Alternative weight functions

All weight based approaches are heavily influenced by their weighting functions. Improvements could be done to the simple inverse linear, or gaussian, weighting to improve their reconstruction results.

Primarily, the general weighting scheme should be decided by experimenting with both inverse linear, and gaussian weighting. Discovering how they can best relate to the properties of the input, and reconstruction variables like half width and *Rmax*, should represent the possibility that a voxel is influenced at this distance from the input pixel. Calculating the optimal standard deviation for gaussian weighting, will find a balance between smoothing and detail. The standard deviation might be based on *Rmax*, *df*, *dv*, be static, or be mixed and limited by multiple of these.

Otherwise, weighting options should be researched to handle specific problems. Tord Øygård worked on some of these problems in his thesis[9]. Valuing a higher brightness appears to give good results in maintaining brightness, but a more continuous scalar weighting could be used, weighting the brightest pixels double of the darkest ones by a formula such as:

$$w = w_{init} \cdot (1 + \frac{p}{p_{max}})$$

A formula to value contrastive pixels, both bright and dark, or simply a higher differentiation than above, could also be used. VGDW also tested lateness, on the basis that the operator could pass over the area again if errors occurred. With a fast reconstruction as described in this paper, it would be better to restart the scan, as overlapping scan passes creates more inaccurate results than a single clean scan. The input variance adaptation of the VGDW is also very interesting, switching how smoothly the input is weighted depending on how detailed the area is. This can remove noise, while preserving edges.

It would be interesting to see if a pixel weight, based on computer vision techniques, could further enhance detail or diminish artefact impact in the reconstruction. Traditional methods like edge detection or segmentation could be used, or the modern approach of convolutional neural nets used to learn strengths and weaknesses of the input. Knowing the origin point of the scan lines could give a reasonable basis for understanding the artefacts of the input.

Researching optimal weighting functions, and experimenting with new weighting options will be essential to create optimal reconstructions. Once discovered, these weighting systems can be used by most weight based reconstruction algorithms, to acquire a similar improvement.

7.4 Research of improved algorithm visual quality

The visual quality of the hybrid approach can be improved, even if it would go at the cost of performance. It would be interesting to research possible variations, and see how they compare visually. Drawing ideas from algorithms that can give the implementation higher quality output, or simply testing how fast reconstruction algorithms with high visual performance could perform on a modern GPU. The current level of GPU processing can make a lot of approaches applicable, allowing us to sacrifice performance for improved quality.

The hybrid approach is simplified to traverse voxels along a dominant axis from a pixel point. It can be reimplemented to traverse from each input pixel, along the image normal. It can trilinearly interpolate points along the normal, making it a discrete approach to the DW method with ellipsoid shaped neighbourhood. This allows it to increase the level of detail while maintaining a lower complexity than the DW method.

Generally, multiple methods should be looked into, finding the best visual performance within a reasonable reconstruction time.

7.5 Implementing complete real-time solution

This thesis has shown that it is possible to achieve a very high rate of frame accumulation. However, a real-time implementation will require multiple steps to be performed, in a total of max 40 milliseconds. Implementing a framework that can incrementally adjust, accumulate, and visualize the volume would be the basis for real-time tests. Incremental visualization should be able to guarantee under 40 milliseconds even for difficult scanning patterns. Incremental accumulation was performed in under 1 millisecond in this thesis. Incremental volume adjustments will either require the volume to be expanded as the probe approaches the edges of the volume, or start with a big volume and change visualization to present the relevant subregion. A fast method for loading of images should also be considered.

This incremental re-initialization of the volume can be time-consuming, as it will require initializing a new volume, copying data into the new volume, and doing a complete new visualisation of the volume. Simplifications should be made where viable, to make the transition as smooth as possible. It can be possible for the volume adjustment step to do some of the work in between the previous images, and spin over the accumulation of some frames while visualization is performed.

If we adjust the volume every 30 frames, after frame 28 the new volume is calculated and initialized, after frame 29 the volume is copied over from the old volume, with frame 30 the last data is copied over and a 0.2s complete visualization is started in the time frame 6 incremental visualizations normally would be performed. Finally after visualization, the 6 input frames are accumulated with the new frame almost instantly before a new incremental visualization with all of them, and the incremental steps are resumed. Now this is just an example with stipulated times, and the final workings should be figured out with comprehensive testing.

Allowing for a good frame around the accumulation could make it possible to achieve real-time solutions, and with the reconstruction presented in this thesis it would be quite durable during different circumstances.

8 Bibliography

[1] "3D Slicer." *3D Slicer*. Web. 12 Oct. 2016.

[2] Dai, Yakang, Jie Tian, Di Dong, Guorui Yan, and Hairong Zheng. "Real-Time Visualized Freehand 3D Ultrasound Reconstruction Based on GPU." *IEEE Transactions on Information Technology in Biomedicine* 14.6 (2010): 1338-345. Print.

[3] "David W. Gohara, Ph.D." *David W Gohara PhD*. Web. 12 Oct. 2016.

[4] Gobbi, David G., and Terry M. Peters. "Interactive Intra-operative 3D Ultrasound Reconstruction and Visualization." *Medical Image Computing and Computer-Assisted Intervention — MICCAI 2002 Lecture Notes in Computer Science* (2002): 156-63. Print.

[5] Gohara, David. "Episode 2 - OpenCL Fundamentals." *YouTube*. YouTube, 18 June 2013. Web. 12 Oct. 2016.

[6] Holm, Sverre, "Ultrasound Imaging" *Fysikkens Verden (1999)*. Print.

[7] Huang, Qinghua, Yongping Zheng, Minhua Lu, Tianfu Wang, and Siping Chen. "A New Adaptive Interpolation Algorithm for 3D Ultrasound Imaging with Speckle Reduction and Edge Preservation." *Computerized Medical Imaging and Graphics* 33.2 (2009): 100-10. Print.

[8] "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems." *The Khronos Group*. Web. 12 Oct. 2016.

[9] Oygard, Tord. "Improved Distance Weighted GPU-based 3D Ultrasound Reconstruction Methods." (2014).

[10] Rohling, Robert, Andrew Gee, and Laurence Berman. "A Comparison of Freehand Three-dimensional Ultrasound Reconstruction Techniques." *Medical Image Analysis* 3.4 (1999): 339-59. Print.

[11] Smistad, Erik, Mohammadmehdi Bozorgi, and Frank Lindseth. "FAST: Framework for

Heterogeneous Medical Image Computing and Visualization." *Int J CARS International Journal of Computer Assisted Radiology and Surgery* 10.11 (2015): 1811-822. Print.

[12] Smistad. "Smistad/FAST." *GitHub*. 08 June 2016. Web. 12 Oct. 2016.

[13] Solberg, Ole Vegard, Frank Lindseth, Hans Torp, Richard E. Blake, and Toril A. Nagelhus Hernes. "Freehand 3D Ultrasound Reconstruction Algorithms—A Review." *Ultrasound in Medicine & Biology* 33.7 (2007): 991-1009. Print.

[14] Theoharis, T. *Graphics & Visualization: Principles & Algorithms*. Wellesley, MA: A.K. Peters, 2008. Print.

9 Appendix

Appendix A - Weighting updates with updated results

In the discussion in Section 5.1 the gaussian weighting was found to be weaker than expected, compared to the linear weighting, based on the evaluation scores of Section 4.1.1. Gaussian weighting was looked into, and the calculation function found faulty. A too high standard deviation created a small weight differentiation, creating a weight spread, more resembling a constant weight, and a box filter, than traditional gaussian functions. A new weight calculation was created to closer reflect the demands to the weight distribution, and compared in this Appendix with the existing solutions. The new gaussian weighting is implemented as described in background Section 2.5.5.2, with a constant *k* at 1.0, and a minimum standard deviation of 0.5.

For clarity the weighting functions are graphed below, comparing weight to distance at different half width values, from 1.0 to 8.0. Weights are limited to the areas in which each respective half width limits them to. Figure A.1 display the original gaussian weighting, while Figure A.2 show the suggested new gaussian weighting. They have a different amplitude scaling through *k*, but more importantly the standard deviation is made smaller than the old version, allowing it to use more of the gaussian characteristic curve. For comparison the inverse linear weighting is included in Figure A.3.

With the variability of both a local, and a global max distance, it is difficult to set a universal weighting function that respectfully represents all distances. A gaussian function with a static global standard deviation is possible, but a too small value will give little differentiation to bigger values, and a too big value will give little differentiation to smaller values. The goal would be to make sure overlapping accumulations have similar weight at the same distance, while respectfully differentiating different distances.

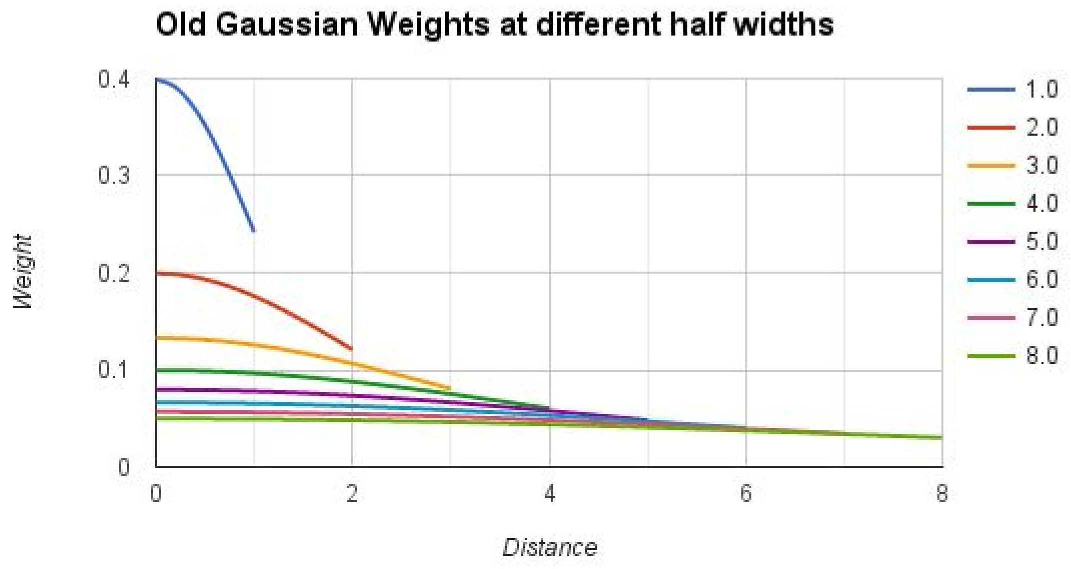


Figure A.1: Old Gaussian Weights - The legend on the right represents different half widths, where a half width of 1.0 implies a local maximum distance of 1.0, and this is where the minimum weight will be produced for this half width. It can be seen that this minimum value is still ~60% of maximum weight, giving a smaller contrast than intended.

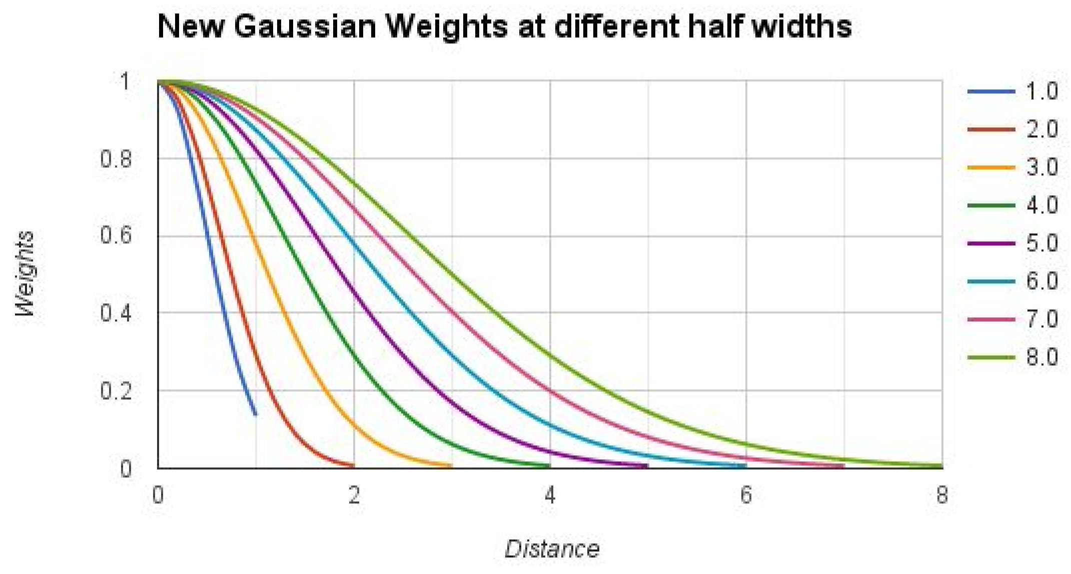


Figure A.2: New Gaussian Weights - The legend on the right represent different half widths. Weights are now scaled for width to use the full slope of the gaussian curve. No amplitude scaling. Half width 1.0 is affected by the minimum standard deviation of 0.5, and is therefore closer to half width 2.0 than otherwise expected.

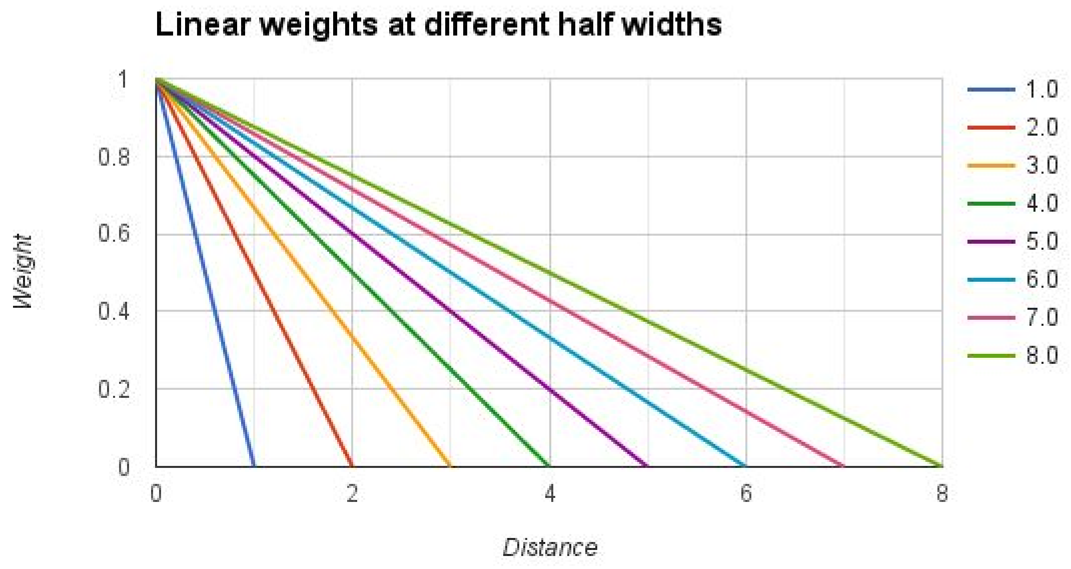
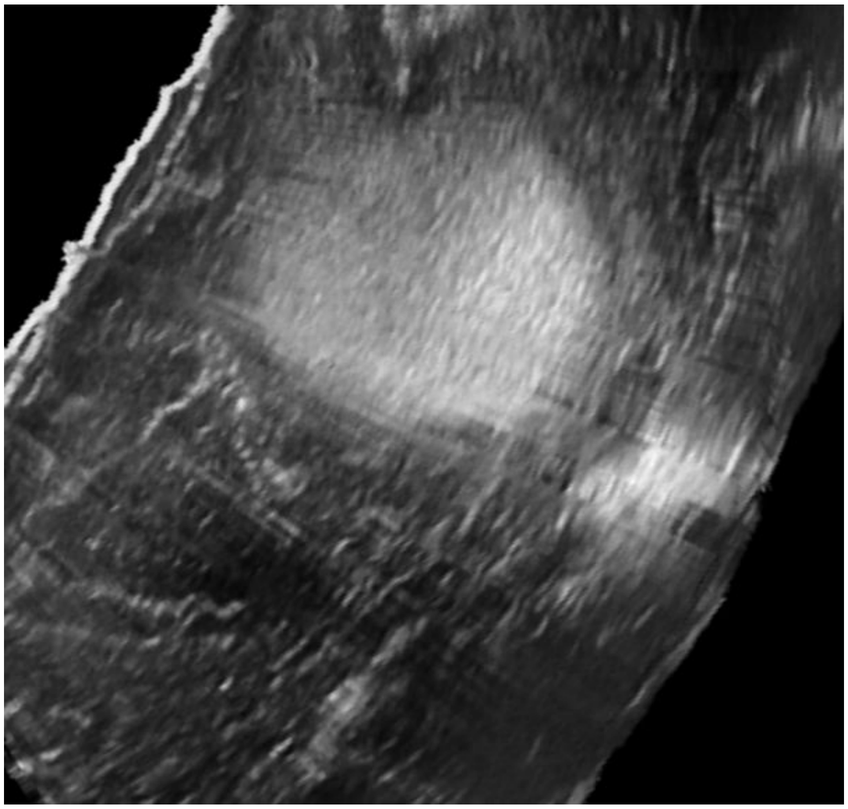


Figure A.3: Linear Weights - The legend on the right represent different half widths. Inverse linear scaling between the zero-point and the given half width, crossing the zero weight at the half width.

Further follows some side-by-side visual comparisons of the old, and the new gaussian weighting schemes, as well as linear for reference. Figure A.4 display a overview cut of the visual results, while Figures A.5 and Figure A.6 zoom in on certain details from this cut, for a side-by-side comparison.



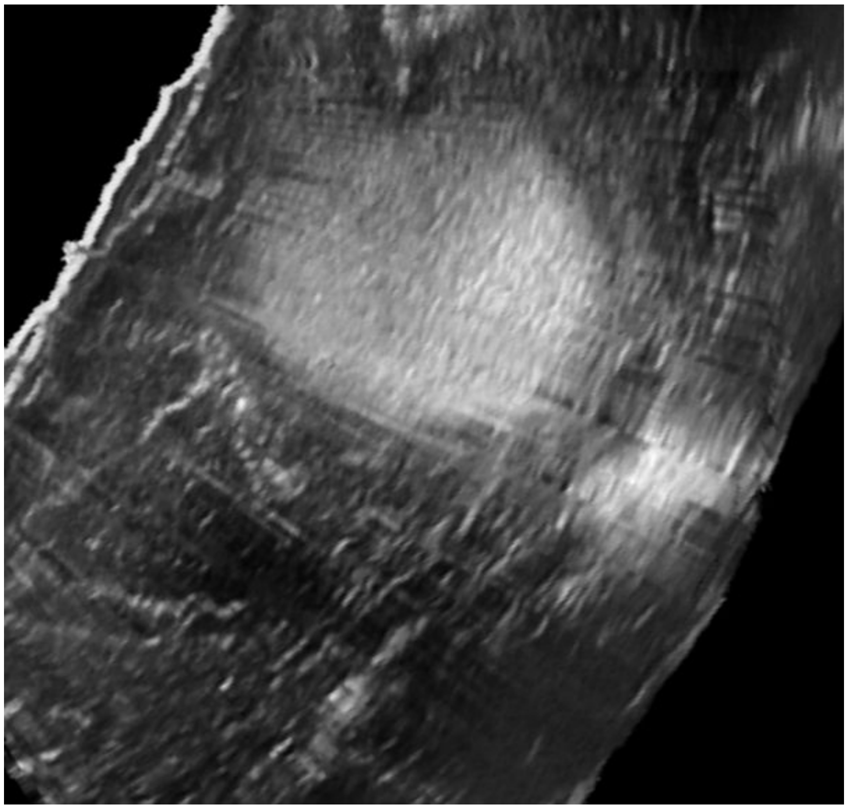


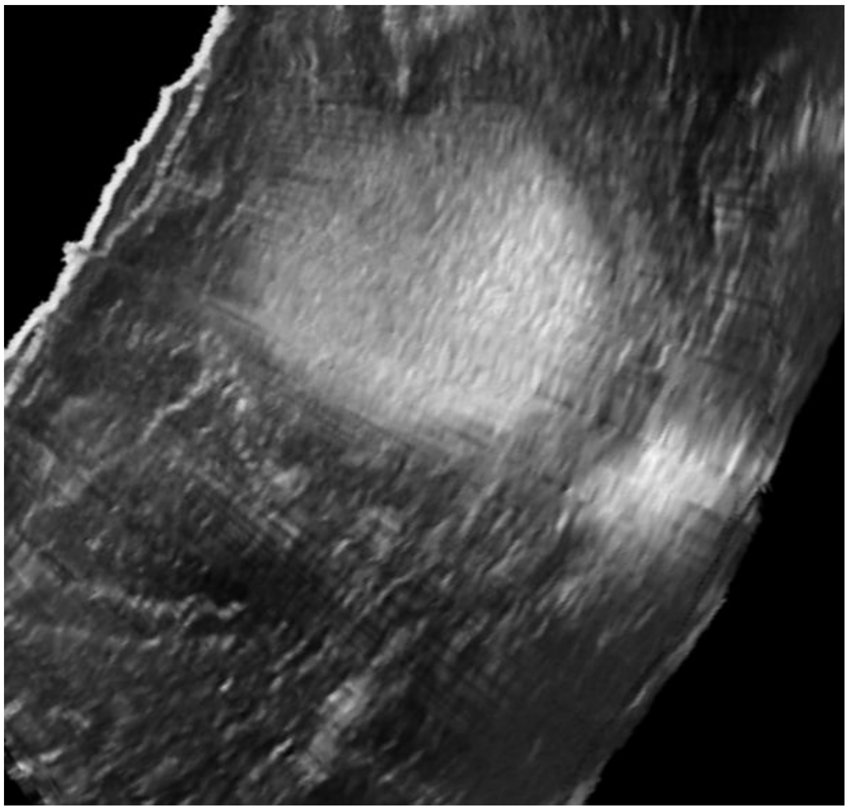
Figure A.4: Visual results - Overview cut. Multiple input frame can be seen in the slice, but more or less jagged lines.

(Top) The results of the new gaussian weight scheme;

(Bot) The results of the old gaussian weight scheme;

(Next page) The results of the linear weight scheme;

The gaussian results appear to contain more detail, but the old results create some quite jagged edges. The linear results appear smooth, but make the darker areas brighter than the other two.



Linear weight scheme.
See description on the previous page.

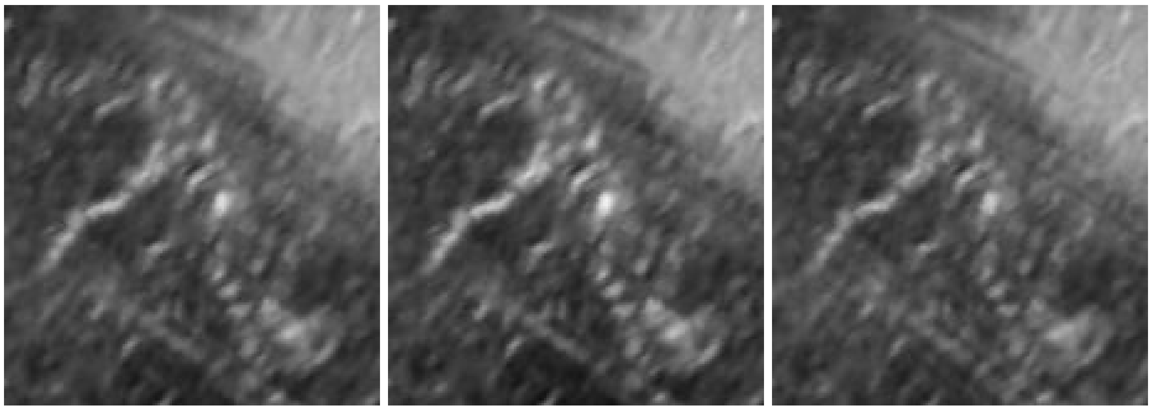


Figure A.5: Visual results - Detail cut. Focused on the detail in the left side of the slice. (Left) The results of the new gaussian weight scheme; (Middle) The results of the old gaussian weight scheme; (Right) The results of the linear weight scheme; The bright detail is clearer, and brighter in the two gaussian reconstructions, especially the old weighting is brighter. The new weighting appears to be in the middle ground between the other two, with good smoothing and good dark detail.

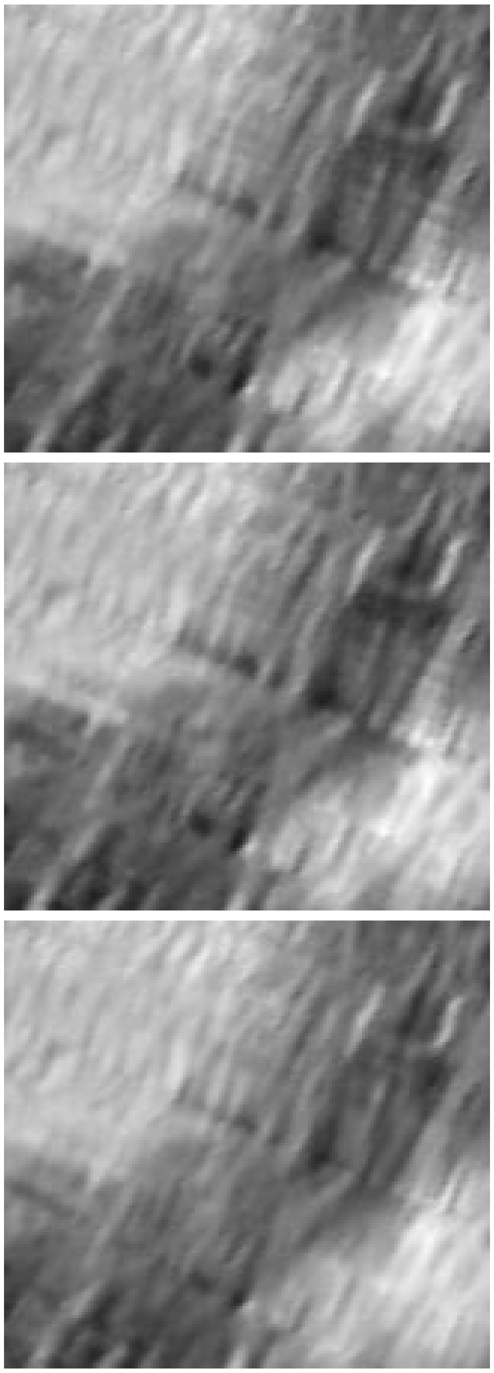


Figure A.6: Visual results - Tumor edge cut. Focused on the bottom right section of the big white tumor in the middle of the main cut.

(Top) The results of the new gaussian weight scheme;

(Middle) The results of the old gaussian weight scheme;

(Bottom) The results of the linear weight scheme;

The two gaussian functions appear to keep more dark detail in the bottom left corner, and on the diagonal towards the top right corner. However they have a bigger slice gap, in the middle of the cut. The new gaussian weight seems to handle this cut, and others, slightly better than the old gaussian. This effect can also be seen in parts other parts of Figure A.4.