Preface

This master thesis is the result of work carried out in 2003 at the section of Statistical Image Analysis at the department of Informatics and Mathematical Modelling at the Technical University of Denmark (DTU).

The work has been supervised by assoc. prof. Bjarne Ersbøll from DTU and senior scientist Thomas Martini Jørgensen from Risø National Laboratory, whom I would like to thank for comments, feedback and generally being very supportive.

I would also like to thank M.Sci., Ph.D Birgit Sander, Head of Laboratory at Herlev Hospital, for being very helpful during the process and for introducing me to different areas in the field of optical coherence tomography.

January 31st, 2004

Lars Gunder Knudsen

Abstract

This report covers some of the processes involved in development of software systems, with an emphasis put on quality, design, usability and the ability to handle changes in the environment. The purpose of the project behind the report, was to develop a flexible image processing system to be used in medical research in the department of ophthalmology at Herlev Hospital (in Denmark) and in relation to this, a graphically based system, possible to extend through development of pluggable modules, has been constructed. The goals of the project, regarding a high level of usability, usefulness and flexibility, have been met, which is confirmed in a user survey.

(In Danish)
Denne rapport beskriver nogle af de processer, der indgår i udviklingen af softwaresystemer, hvor der lægges vægt på kvalitet, design, brugervenlighed og mulighed for at klare forandringer i omgivelserne. Formålet med projektet bag rapporten, var at udvikle et fleksibelt billedbehandlingssystem til anvendelse i medicinsk forskning på øjenafdelingen på Herlev Sygehus. Et grafisk baseret system, med mulighed for udvidelse gennem tilkobling af udviklede moduler, er konstrueret. Projektets mål m.h.t. høj brugervenlighed, brugbarhed og fleksibilitet, ses som opnået, hvilket bekræftes ved brugerundersøgelse.

Keywords

Software architecture and design, image processing, Java, C++, XML, Optical Coherence Tomography (OCT)

Intended audience

This report aims at not only being the “traditional” result of a master thesis done at the department of Informatics and Mathematical Modelling (IMM), where project related developed software will not leave the lab, but also to function as a set of guidelines for how software projects done for customers should be handled, including areas less obvious to the novice developer. It is therefore essential that the reader will take the time, to fully understand the processes necessary to go through, prior to initiating coding.

As the goal of the project was to deliver a fully functional software system to be used by researchers in medicine and further developed, possibly by informatics students at the Technical University of Denmark (DTU), the majority of the report has been constructed, keeping in mind that it should work as a reference manual for those two groups of people as well.

The reader is assumed to have a fair knowledge in the world of software development, with skills on an intermediate level in Java and C++ programming.

Experience with basic image processing and the algorithms behind it is not a necessity but will make some parts of the thesis easier to comprehend.

Table of Contents

1. Introduction

This report describes the processes involved in developing a flexible image processing software system to be used by researchers in the field of ophthalmology (the art and science of eye medicine). Most of the techniques involved can easily be applied to other fields of software development, e.g. addressing the different problems involved in documenting the requirements stated by the customer, producing a robust design for the system, taking the time needed to refactor, etc..

To better understand the background for doing this project, a short introduction to Optical Coherence Tomography (OCT), used in acquisition of the retinal images to be processed by the system, is provided.

Optical coherence tomography is an imaging technique that produces high resolution cross sectional images of optical reflectivity. It is based on the principle of low-coherence interferometry where distance information concerning various ocular structures is extracted from time delays of reflected signals. Direct measurements of the time delay of reflections is not possible, because of the high velocity of light, while low-coherence interferometry provides a precise measurement of the echo time delay of the back-reflected light. This is done by comparing reflected light from the sample in question with light, that has traveled a path of known length to a reference mirror. For ease of comprehension, this principle is illustrated in figure 1.

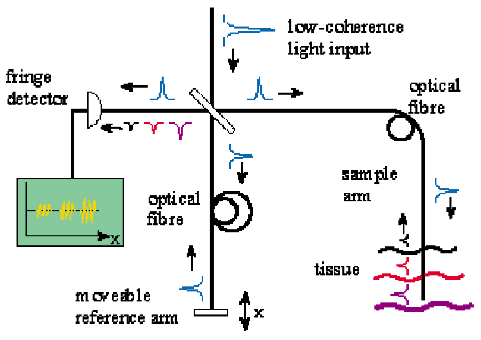


Figure 1, Principle of optical coherence tomography. (Image from: "Biomedical Optics", PhysicsWeb, June 1999, URL: http://physicsweb.org)

This technology makes it possible to visualize disease pathology in living tissue, where a biopsy would normally be performed, as the optical coherence tomography can be performed at a resolution approaching the cellular level[1].

Light waves are emitted by a super luminescent diode, the function of which can be compared to the application of sound waves when doing ultrasound scanning or x-rays in computed tomography (CT).

As a result of the high level of resolution achievable, OCT is particularly suitable for retinal thickness measurements. The acquired images can be presented as either cross sectional images or as topographic maps. Cross sectional or B-mode imaging is accomplished by acquiring a sequence of interferometric A-scans across a section of the retina. In figure 2, a histological image, of the layers in the retina, is illustrated, along with their correspondence in a visualization, done by the developed system, in figure 3.

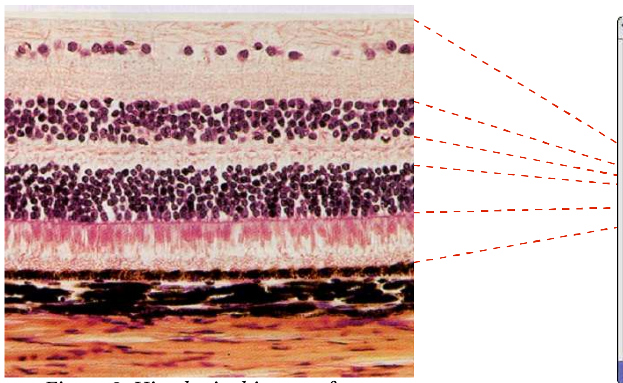


Figure 2, Histological image of a cross-section of the retina.

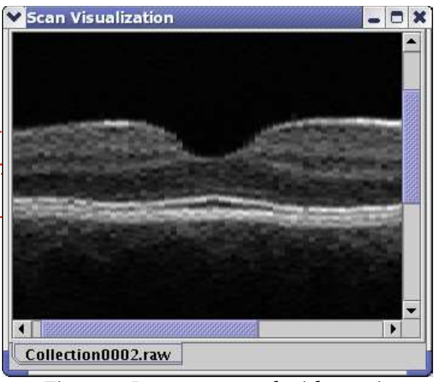


Figure 3, Image, scanned with a retina OCT scanner and processed by the system developed in this project.

This project seeks to help researchers in the field of ophthalmology, by providing a tool to perform post processing of the B-scans acquired by two different retina OCT scanners, in an easy way.

1 **Pioneering New Applications for OCT Research**, RLE Currents, Volume 11, No. 2, 1999
URL: http://rleweb.mit.edu/Publications/currents/cur11-2/11-2oct.htm

The actual image acquisition is done, using an OCT scanner connected to a customized computer system, as shown in figure4.

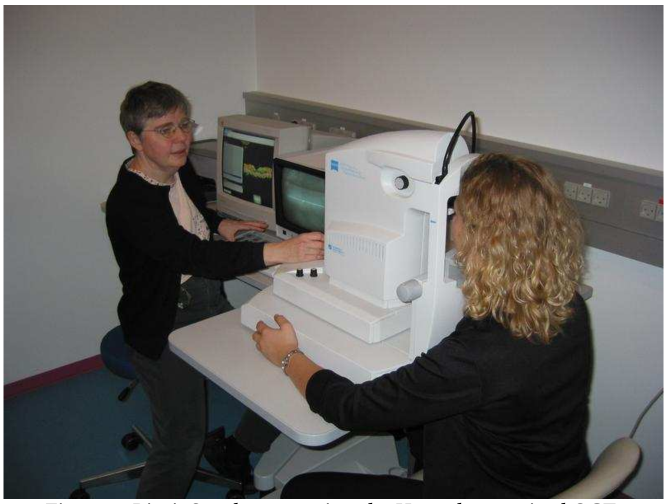


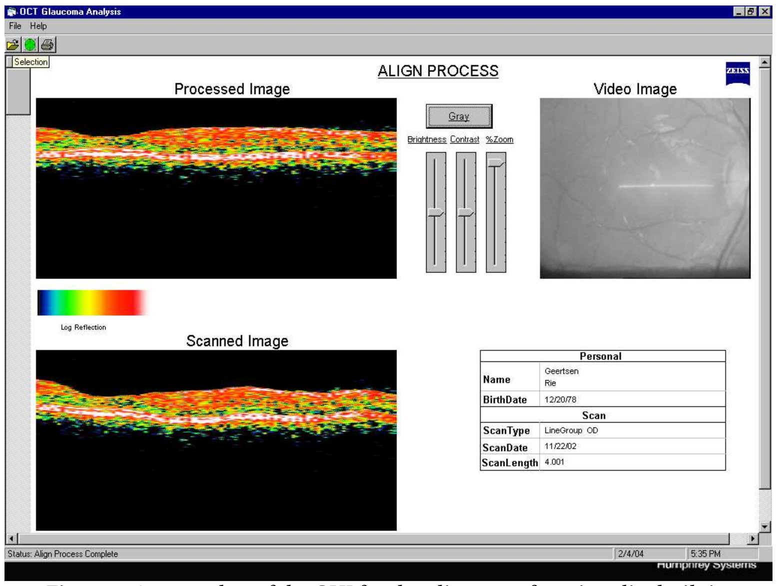
Figure 4, Birgit Sander operating the Humphrey retinal OCT scanner at Herlev Hospital.

Currently, users of the Carl Zeiss Humphrey OCT Retinal Scanner have no real alternatives than to use the accompanying software when required to do post image processing of acquired data. Very few standard processing facilities are available, one of them is the built in alignment method, shown in figure5.

In most cases, this is sufficient, but sometimes, it would be nice to have a possibility of applying more advanced and/or different algorithms, than the ones provided with the scanner.

The immediate problem identified at project start, is to be able to collect a series of acquired scans in one single image to enhance the signal/noise ratio and reveal more details, thus providing a better base for detection of eye diseases.

The aim of this project, is to produce a system, flexible enough to be extended with new functionality, integrating with existing logic, without interfering with other parts of the system.



Personal	
Name	Geertsen Rie
BirthDate	12/20/78
Scan	
ScanType	LineGroup OD
ScanDate	11/22/02
ScanLength	4.001

Status: Align Process Complete 2/4/04 5:35 PM

Figure 5, A screenshot of the GUI for the alignment functionality built into the OCT system from Carl Zeiss.

Also, it must be easy to use for people with no programming and little or no image processing skills. The software should not be limited to run on one operation system only (e.g. windows), as users and developers of the system might have different choices of platform, e.g. Linux(x86), Solaris(Sparc), etc..

It's assumed, that people, using the system, will fall into one to three categories:

1. “Developers”, extending the system by writing new code,
2. “Constructors”, using the system to produce image processing algorithms and
3. “Users”, applying produced algorithms on data to be processed.

It should be possible for developers to integrate existing code/programs with the system to minimize the need for them to rewrite their code to fit the language and platform used for the system. It is a necessity to produce a graphical user interface

(GUI) for constructors and users in general to utilize the system, but the software architecture must support the need for separation of the GUI and processing logic, should it be required to use the system under different environments, e.g. making use of produced algorithms via a text based shell.

Finally, the software should be based on free components (e.g. software released under the LGPL or similar), not bringing extra costs to users nor developers.

The report starts with a chapter, describing the processes involved in designing and developing a robust software system. Very early in the process, it is important to agree on a set of system requirements, that can be used as the overall guidelines under the entire development process. It is then up to the developer to form a set of functional specifications, where initial design considerations will be documented, based on each of these requirements.

One of the requirements, states that the system should be designed to run on different platforms. This has a major impact on the choice of programming language, external libraries and programming environment, and will therefore be dealt with in detail in a section of its own. Java was eventually chosen as the primary programming platform for the system, but before this decision was made, a large amount of time was spent, implementing the system in C++. This, however, turned out to be too complex to handle, when the framework, combined with around 10 external libraries had to be compiled and running on different operating systems (Windows, Linux). If this was to be a system, that would easily allow extensions to be made by students with average programming skills, another approach would have to be taken. The decision fell on using Java, and all the written C++ code was scrapped.

Based on the design considerations in the functional specification, the system architecture is forged, leading to a design where image processing logic will be split into small self containing modules. These modules should then be connected to form larger algorithms, done dynamically using an underlying framework, functioning as a sort of testbed. All communication to and from the framework, as well as modules intercommunicating in constructed algorithms, is done using the

Extensible Markup Language (XML). This enables users to very easily connect legacy software directly with processing components in the system, as well as deciding whether certain modules should be replaced with external entities. In the case, where more complex communication protocols are needed, it will be fairly easy to implement a new set of components to deal with this, possibly by making a bridge to C or C++ code using the Java Native Interface.

A step-by-step development guide on how to create new modules, as well as how to combine them to form algorithms, is described in detail in the tutorial section. This system is designed to be used by and developed further by people not involved in the project at the time of writing. Large parts of the report has therefore been written as reference documentation for programmers, medical researchers and others with an interest in the framework.

Many pluggable modules have been created throughout the duration of the project. These will be described in detail in a separate chapter, which includes background information for each module, requirements, implementation specific details, as well as a testing section, showing how the module functions in a context.

With all the building blocks in place, it is now possible to construct algorithms, capable of performing more complex tasks. Two large examples are provided, to show how module components can be used – and reused – in very different ways.

The first example, is an algorithm, that will accept a sequence of RAW data files, exported from an OCT scanner, and improve the signal/noise ratio by aligning and superimposing them to one resulting image. This is followed by an example, where automatically generated patterns are used in combination to generate realistic data, to be used in testing the image processing logic of a specific module.

This module was created by a future user and developer of extensions for the framework, Thomas Martini Jørgensen[2], to test how flexible the system is, when trying to extend it with new functionality. Based on OCT related software, previously made by Thomas, a new alignment method was identified to be implemented.

2 Senior Scientist, Risø National Laboratory

As the system is meant to be used in opthalmologic research at Herlev Hospital, it is important to test the usability and functionality of the system in a real life situation. Birgit Sander[3], being one of the main researchers in the field, tries to use the system, and takes us through the creation of an algorithm, capable of enhancing the edges of a series of subsequently acquired scans.

Many features are planned to be implemented after the time of writing, and some of these are described in detail. Integration with external systems, is probably one of the more interesting features, as it would enable the system to communicate directly with different OCT scanners, delivering instant processing of data – possibly while patients being treated, are still in the examination room (with the OCT scanner).

Included in the appendices, are the full list of system requirements as well as their functional specifications. It is recommended, for the reader to take a quick glance at the glossary in the back of this report, to see some of the terms and definitions used.

3 M.Sci., Ph.D., Head of Laboratory, Herlev Hospital

Important note:

Throughout the report, the "A-scans" of the acquired images, are referred to as "rows in the image". This is because, the underlying image, as well as the originating RAW data, has each A-scan represented in rows.

However, because of visual compatibility with some of the image representations in the software delivered with the OCT scanners from Carl Zeiss, the module constructed (in the software developed for the project), that visualizes these images, will mirror the image around a diagonal line in the image, so that rows become columns and vice versa – prior to putting the image on screen.

If this decision, to separate the way images are preserved in the underlying model from their graphical representation on screen, poses a problem when using the system, an extra visualization module has been constructed (VisualizeScansAsModel), where this "mirroring" of the image has been removed.

2. Imaging System Development

This chapter covers some of the major steps involved in development of the system. As the focus of the thesis is to make a robust software system, to be used in medical research, this part of the project is considered to be much more important than areas of the report covering the actual implementation of image processing algorithms. This will become more obvious, later in the report, where the produced framework allows algorithm implementation to be done, with very little effort.

Agreeing upon a list of system requirements is the first step in almost any software project. These requirements function as the overall guidelines throughout the development, and shouldn't change without a common consent between the manufacturer[4] and the customer. When requirements are in place, it's up to the software developer to run through the requirements and come up with initial suggestions of how each requirement will be addressed in a functional specification. These should not include low level implementation specific details, but more function as a set of components inspiring the actual system design.

From the beginning of the project, it was required that the system would not be confined to run on a single operating system only, as many different platforms are used in the medical institutions as well as at the universities, where this system would be assumed to have it's place in the future. The choice of development platform and programming language therefore has to be considered well, before engaging in any development activity. As it turns out, even with a good deal of effort put into making the right choice, unexpected things might turn up, and what seemed to be the perfect solution actually becomes more of a problem. In this case, it lead to a complete rewrite of two months of written code – not an easy decision to make, but definitely the right one.

In order to be able to get a greater overview of the system to be developed, it's essential to produce a high level architectural design of how all the major components will work and how they will interact . Software architecture forms the backbone for building successful software-intensive systems. An architecture

4 The company or people developing (and in most cases selling) the software.

largely permits or precludes a system's quality attributes such as performance or reliability. Architecture represents a capitalized investment, an abstract reusable model that can be transferred from one system to the next. It also represents a common vehicle for communication among a system's stakeholders, and is the arena in which conflicting goals and requirements are mediated.

The right architecture is the linchpin for software project success and the wrong one is a recipe for disaster[5]. It's only human to make mistakes, and that might be one of the main sources for us to gain good experience, and learn how not to do things in similar situations the next time around. Sometimes, however, we can't afford to make that many mistakes, while making our own experiences. In these cases, we try to learn from other peoples success or failure, and how they made it. In software architecture, “design patterns” tries to address this issue by cataloging common pitfalls and their solutions. Understanding and using design patterns in making the core architecture is a great help when trying to prevent potential weak points in the system, causing unwanted behavior (e.g. system crashes or data corruption). They can also assist in making the system more flexible and easier to extend with new features.

The developed system consists of a base framework functioning as a virtual testbed for constructing flexible image processing setups, as well as a variety of pluggable modules to be used as components in the construction. To make the extending modules as easy to make as possible, much of the logic, common to all modules, has been placed in the framework, enabling the module developers to concentrate on the mathematical models they're implementing.

Often, especially when it comes to software used in research, the ability to get the different systems to communicate – or at least be able to exchange data – is highly appreciated. Not knowing what exact systems, the users might choose to work with, it was decided to let all non-binary communication inside the system, be based on the Extensible Markup Language (XML). This also makes the system easier to extend with new features, as adding elements to an XML document would not require changes to existing logic.

5 “Software Architecture”, http://www.sei.cmu.edu/ata/ata_init.html, Carnegie Mellon University

Like in most other image manipulation and processing systems, a graphical user interface is produced to assist the user in designing algorithms and visualizing results. Inspired by the Model-View-Controller pattern, a clear separation from the underlying processing logic was made, enabling future versions of the system to be created, using a different user interface, e.g. being text based.

When focusing on the GUI, usability is an important element to remember, when creating user interfaces for systems to be used outside ones "personal lab". For this project, it has been a goal to make a system that would be easy to use, it should be noted, though, that covering all aspects of creating great usability would require much more resources than available during the development of the system.

2.1 System requirements

Before any development on the system can begin, it's a necessity to get some requirements in place[Grand98, p. 30].

These high level system requirements are the main reference points for the rest of the development process, and can be considered a sort of contract between the customer and the software developer.

High level requirements should not include implementation specific details that would not affect the system in any way visible to the customer.

After a few sessions with Birgit Sander[6], a suiting set of preliminary requirements, of things , desirable to have in the system, were agreed upon:

1. Import of the RAW data format, exported from different variants of Carl Zeiss Optical Coherence Tomography retinal scanners.
2. Image data visualization on screen.
3. Alignment of single images.
4. Collection AND alignment of a set of 2 or more acquired images, representing the same data.
5. Other digital signal processing modules, including basic arithmetic modules.

6 Head of Laboratory, Dept. of Ophthalmology, Herlev Hospital

6. The system should be possible to extend by students with average programming skills.
7. Existing pieces of related software, made by local researchers with an affiliation with the hospital, should be possible to merge with the system.

A full list of the organized requirements can be found in Appendix A.

2.1.1 MoSCoW prioritization

When listing requirements, a successful method of prioritizing them, is by using words that have meaning. Several schemes exist but a very popular method is the acronym MoSCoW[7]. The o's in MoSCoW are just there for fun and the rest of the word stands for:

- **M**ust have this
- **S**hould have this feature if at all possible.
- **C**ould have this if it does not affect anything else.
- **W**on't have at this time but would like to have in the future.

The importance of this method is that when prioritizing the words mean something and can be used to discuss what is important. A requirement listed as a "Must" is non-negotiable. Without them the system will be unworkable and useless, while “nice to have” features are classified in the other categories of "Should" and "Could”.

Requirements marked as "Won't" are potentially as important as the "Must" category. It might not be immediately obvious why, but it is one of the characteristics that makes MoSCoW such a powerful technique. Classifying something as "Won't" acknowledges that it is important, but can be left for a future release. In fact a great deal of time might be spent in trying to produce a good "Won't" list. This has three important effects[8]:

7 “MoSCoW Prioritisation”, URL: http://www.ogc.gov.uk/sdtkdev/examples/HMCE/Guidance/MSCW/moscow_prioritisation.htm
8 “Project prioritisation using MoSCoW”, URL: http://www.coleyconsulting.co.uk/moscow.htm

- Users do not have to fight to get something onto a requirements list.
- In thinking about what will be required later, affects what is asked for now.
- The designers seeing the future trend can produce solutions that can accommodate these requirements in a future release.

When setting priorities for the requirements listed for this project, the MoSCoW model was used for inspiration. However, the list of requirements is not very long compared to enterprise scale systems, where the project team would spend months, prioritizing lists of requirements that could fill a book.

It might seem like overdoing it to use the MoSCoW model for this project, but during development, it has shown an invaluable resource to have, when in doubt of what was to be implemented.

2.2 Market Analysis

Developing a full blown image processing system from the ground up, is not a small task. It is therefore essential to investigate possible alternative options, where 3rd party software could be used as a base to build on.

An extensive search for commercial as well as free tools, capable of fulfilling the high level system requirements, listed in Appendix A, was made, and

the results would reveal, if it was feasible to make a new product or not.

A few potential alternatives emerged and an overview of these are listed in tables 1 to 5, showing detailed descriptions, including pricing, licensing, compatibility, etc..

Note: Prices are converted to DKK where only found in foreign currencies.

Name	GIMP (GNU Image Manipulation Program)
Vendor	N/A (Open Source)
Description	An open source image manipulation program. It's functionality is very close to that of PhotoShop from Adobe. GIMP can be extended with plug-ins created in Lisp, working as scripts, utilizing low level functionality in the graphics processing engine.
Platform	Linux, Unix (different), Windows
Price	Free
Pro	• Many standard image processing features • Possibility to make plug-ins • Free
Con	• Does not support floating point pixel types • Not easy to design and change work flows for image processing on the fly
URL(s)	http://www.gimp.org
Summary	Using GIMP as a platform for development of the system required, would mean, that the source code for GIMP itself would have to be modified to support floating point pixel values. There is also no mechanism for easily combine pluggable modules in work flows, to be used as composite algorithms, when processing images.

Table 1, Data on GIMP

Name	Image-Pro
Vendor	Media Cybernetics
Description	Professional quality image acquisition and processing software.
Platform	Windows 98/NT/2000/XP
Price	DKK 25765.00 (Discovery version) DKK 38653.00 (Plus version)
Pro	• Supports floating point images • Many powerful image processing features • Possibility to make and buy plug-ins
Con	• Only runs on windows • Very expensive
URL(s)	http://www.mediacy.com (Vendor) http://www.unit-one.dk (Retailer)

Name	Image-Pro
Summary	Basically, Image-Pro (with the extensions needed), would probably fulfill most of the image processing requirements seen from a pure technological stand point. There are two major downsides to using Image-Pro as the base platform though: 1. It only runs on windows, making it impossible to port to other platforms, as required 2. The licensing costs are very high for people developing modules for the software, as well as those using it for research.

Table 2, Data on Image-Pro

Name	LabVIEW
Vendor	National Instruments
Description	Graphical algorithm designer (and executor).
Platform	Linux, Windows 98/NT/2000/XP
Price	DKK 18,170.00 (Full) DKK 31,910.00 (Professional) DKK 23,740.00 (Vision Development Module [Windows])
Pro	• Multiple platforms are supported • Graphical algorithm designer
Con	• Very expensive • Not very easy to modify when standard modules are insufficient. • Vision Development Module only for Windows
URL(s)	http://www.ni.com/labview (Vendor)
Summary	This platform is probably the best candidate for the project among the alternatives listed. It's very easy to construct complex algorithms in a very short time. There are, however, some major issues, that are hard to neglect: 1. The platform is very expensive (around DKK 40,000.00 for the base + vision package) for users and developers of extensions. 2. Extensions to do more complex image processing are only delivered as *.dll's for Windows, eliminating the use of other platforms. 3. It's easy to use for builders of algorithms, but hard to make new modules, when the existing ones are insufficient.

Table 3, Data on LabVIEW

Name	MatLab
Vendor	Mathworks
Description	Mathematical scripting engine/programming language.
Platform	Linux, Windows 98/NT/2000/XP
Price	(ex.moms) DKK 21,950.00 (MatLab) DKK 29,950.00 (SimuLink) DKK 9,950.00 (Image Processing Toolbox)
Pro	• Multiple platforms are supported • Graphical algorithm designer (SimuLink) • Many universities use it (wide spread)
Con	• Very expensive • No modules created yet for image processing in SimuLink
URL(s)	http://www.mathworls.com (Vendor) http://www.comsol.dk (Retailer)
Summary	On the positive side, this platform delivers basically everything needed for the system, with the exception of a set of tools in SimuLink to reflect the image processing methods in MatLab. The price, however, is too high, when taking into account that every runtime and every development installation requires installations with added licensing fees in the area of DKK 60,000.00.

Table 4, Data on MatLab

Name	IDL
Vendor	Research Systems Inc.
Description	High level programming language, providing advanced visualization functionality in an easy way.
Platform	Linux, Windows, Unix
Price	$ 3,000.00 (IDL Personal) (≈ DKK 18,000.00)
Pro	• Multiple platforms are supported • Quick visualization of data.

Name	IDL
Con	• A bit pricey • Focuses on visualization – more than of the business logic needed to support it.
URL(s)	http://www.rsinc.com (Vendor)
Summary	A nice high level language, targeted at developers, wanting to do quick scientific visualizations. Image processing capabilities are not at the level needed for the project.

Table 5, Data on IDL

Although some of the found software packages come close, none of them could fulfill the requirements given at the project beginning regarding:

- **Portability** - The software must be able to run on multiple platforms
- **Low cost** - Users and developers of the software must not be obligated to pay high license fees
- **Floating point pixel values** - Image processing operations must support floating point pixel types.

Based on the results of the market analysis, showing very high licensing costs on software, suitable for use in this project, it is seen necessary to produce the basic framework from the ground up. This would still allow incorporation of 3rd party libraries where seen fit (e.g. vector math libraries, GUI toolkits, etc.), but bring the potential cost enforced on the users and developers to a minimum.

Note: Development costs are close to zero, as this system will be developed as part of a master thesis project. If this had been a software project done by well paid freelance developers, the choice of building extensions to a system like LabVIEW might have been more realistic, as development costs (for the system built from scratch) would be in the range of DKK 50K-100K per month per developer hired.

Including design, planning and quality assurance, with a decent development team of around 5-8 people, the total cost would easily run up in the millions (of DKK).

2.3 Functional specification

After agreeing upon a set of requirements, it is up to the software implementer(s) to try to make small design considerations for each requirement. These should not go into deep implementation specific details, but instead function as components to be used when making the complete system design.

As an example, let's assume a given requirement states the following:

“A caching mechanism must be implemented to store different types of resources in the system. The objects cashed, should not be deleted until a specifiable time period has passed, after the objects are no longer referenced anywhere else.”

In a functional specification, relating to this requirement, considerations for a possible solution to a partial design, might lead something like:

“Objects, to be stored in the cache, should be wrapped in a container class, capable of keeping track of references made to the object. When the reference count reaches zero (when the object is no longer being used), the current time plus a configurable timeout value is stored in a member variable in the wrapper class. At certain time intervals, all cached objects will be checked to see if they are timed out

AND their reference count is still zero. In this case, the object should be deleted."

The full functional specifications for the system are listed in Appendix B.

2.4 Choice of platform

When developing systems to be used by others, outside the comfortable environment of ones home PC or the local university lab, it is important to gather as much information as possible, about where the system will be used, by whom it will be used, what software platform restrictions there will be, licensing costs of 3rd party tools, performance requirements, etc.. Failing to do so, might cause the software to become more of a problem, than the solution it was meant to be.

2.4.1 Considering all the options

In the beginning of the project, a great deal of effort was put into finding the right platform for the system in terms of programming language, graphical user interface libraries, digital signal processing libraries, etc.. It was also of importance that an average student at a technical university, with maybe only a beginners programming experience, would be able to use the system and extend it, without feeling the problem would be too overwhelming to take on.

Before any choices could be made, some data mining was done to find good 3rd party software candidates for the components to be used in the system. In table 6, the 3rd party libraries, compilers, etc. that came into consideration, are listed:

No.	***Name***	***Description***	***Type***	***Language***
1	Blitz++	Fast vector math and signal processing library based on C++ metaprogramming.	Math library	C++
2	Simple DirectMedia Layer (SDL)	Extensive multi-platform media library with an emphasis on fast graphics.	Graphics library	C/C++
3	ParaGUI	A lightweight and platform independent GUI framework using the SDL library.	GUI library	C++
4	PicoGUI	Very lightweight GUI framework running on anything from embedded devices to desktops.	GUI library and server	C
5	wxWindows	Very mature and extensive GUI framework.	GUI library	C++
6	SDL_net	Low-level TCP/IP extension to the SDL library.	TCP/IP library	C++
7	Boost	C++ extensions to compensate for differences between compilers and platforms (and much more).	C++ extensions	C++
8	MinGW	Toolkit for cross compiling Windows applications from a Linux machine.	Cross compiling toolkit	C/C++
9	ImageMagick	Image manipulation framework capable of handling many known image formats.	Image framework	C/C++
10	gcc	The GNU C/C++ compiler.	Compiler	C/C++
11	Java2 SDK	The Java2 Software Development Kit including a large base of built in functionality.	Compiler and SDK	Java
12	Xerces	XML parser library.	XML library	Java or C++
13	Xalan	Extended XML handling library, includes XPath and XSLT transformations.	XML library	Java or C++

Table 6, Listing of the potential 3rd party software components, to be used in the system.

After some testing and further elimination, the following two constellations were under consideration:

1. A pure C/C++ solution, combining SDL, ParaGUI, Blitz++ and Xerces, using gcc as the primary compiler (on Linux) and MinGW to cross compile to windows.
2. A Java solution, where most libraries were included in the standard SDK, except for Xalan.

A list of pros and cons for the two different solutions was made to assist in the decision making resulting in a choice of the first option of using C++ - mainly because of the performance gain it had on the Java solution, when it comes to signal processing. Thus, this was meant to be a pure C++ solution, that would compile and run on multiple platforms – Linux (x86) and Windows being the two main target platforms.

Designs were made (see the section on architecture/system design) and the implementation process started.

After a few months of development, the complexity involved in making all libraries compile – even just on one platform alone – had grown out of proportion. I then realized, that even if this system was going to be performing very well *and* run on multiple platforms, no ordinary student - not even with moderate software development skills – would take on the challenge of getting all the pieces of the system to work together. At least not on any voluntary basis.

I now faced the hard choice of either sticking with what I had, make a nice working system, that would probably never be extended by anyone but myself, or start over – throwing away all the code I had developed, yet bringing the gained experience to make a much better system. A lot of the graphical user interface, as well as a few pieces of core processing logic modules were already in place, but that was not any excuse good enough to justify going further down the complex path.

I closed my eyes, took a deep breath and made the only right decision possible – the system code would have to be rewritten in a new environment. This time, the obvious need for the core parts of the system to be built on as few different frameworks, libraries and toolkits as possible, was the key factor that lead to Java being used.

To compensate for Java's poor performance when it comes to vector math, it was a necessity to make it possible to integrate the system with different native libraries through the Java Native Interface (JNI), that functions as a bridge between Java, running on a virtual machine, and C/C++, being compiled natively for the platform used.

JNI can be used to integrate C/C++ and Java on many levels, but for the purpose of this project, the focus has been on using JNI to call highly tuned mathematical operations written in C++.

2.5 Architecture and design

Throughout our lives, we gain experience - mostly by doing things wrong. In software development, this is also true, but sometimes you can't afford to do things wrong, and when you may finally see that some part of your architecture will break under some stressed situations, you may try to patch things up to save time – or that is.... you THINK you save time – but that's a whole different discussion (see the section on refactoring).

Creating a good design is essential to making a robust and flexible system, and a few techniques to help in the process are presented in this section.

2.5.1 Using Design Patterns

In the early 1990s, Erich Gamma, Richard Helm, John Vlissides and Ralph Johnson addressed this challenge and began to work on one of the most influential computer books of our time [Grand98, p. 1-5]: “Design Patterns”. These patterns are reusable solutions to recurring problems that occur during software development, collected in a cataloged fashion and given a name for easier recognition among developers. To give an idea of how developing using software patterns works, suppose that you need to write a class that manages motor control by encapsulating low level functionality, and providing a high level public interface to users of the class. In this case, you would not want more than one instance of the class to exist at any given time, as it might give unpredictable results, but how to manage this?

The Singleton pattern [Grand98, p.127] handles this kind of situation by controlling the instantiation mechanism internally, using a static member variable, thereby ensuring that there will never exist more than one instance.

Continuing the example of the motor controller, to implement a class using the Singleton pattern is fairly simple, as you can see from the diagram in figure 6, and the accompanying implementation example in source listing 1.

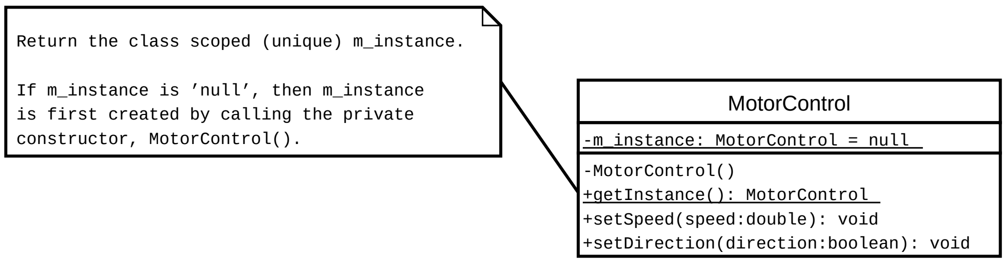

Return the class scoped (unique) m_instance.
If m_instance is 'null', then m_instance is first created by calling the private constructor, MotorControl().
MotorControl
-m_instance: MotorControl = null
-MotorControl()
+getInstance(): MotorControl
+setSpeed(speed:double): void
+setDirection(direction:boolean): void

Figure 6, UML diagram of the MotorControl class using the Singleton pattern.

Walking through the code example, notice that the class constructor has been made unavailable to the public (users of the class), forcing outside code to access it through the static getInstance() method. When using the MotorControl class, the public methods would be accessed by going through that function, as shown in source listing 2.

This way, you don't have to worry about the motor controlling methods being used in an unmanaged way.

Using patterns in general when developing software also helps when trying to figure out why the system might not behave as you expect it to and requires debugging. Here, the patterns - if used correctly - ensure that the more common design mistakes can be ruled out.

```
public class MotorControl {

        // An object to hold the class instance when created
        private static MotorControl m_instance = null;

        // A private constructor, not callable from outside this class
        private MotorControl(){
                // initialize contact with low level motor control
        }

        // The public instantiation function,
        // synchronized to ensure thread safety
        synchronized public static MotorControl getInstance(){
                if( null == m_instance ){
                        // If this is the first call to getInstance, create the object
                        m_instance = new MotorControl();
                }
                // Return the instance object
                return m_instance;
        }

        public void setSpeed(double speed){
                // set the speed...
        }

        public static final boolean LEFT = false;
        public static final boolean RIGHT = true;

        public void setDirection(boolean direction){
                // set the direction...
        }
}
```

Source 1, MotorControl class - an example of the Singleton pattern in use.

```
MotorControl ctrl = MotorControl.getInstance();
ctrl.setSpeed( 0.0 );
ctrl.setDirection( MotorControl.LEFT );
ctrl.setSpeed( 100.0 );
```

Source 2, All interaction with the MotorControl goes through the getInstance() method.

2.5.2 *Model-View-Controller*

Throughout the design and implementation, it has been an overall goal to separate program logic and graphical user interface. To achieve this, the Model-View-Controller (MVC) pattern [Buschmann96, p.125] has been perfect.

Basically, the goal of using the MVC pattern, is to make a clear distinction of what part of the code belongs to the core data storing, processing and handling, and what part of the code belongs to the current choice of visualization mechanism (e.g. GUI based, text based, printer output, etc.).

To better understand the split between model, view and controller, take a look at the diagram in figure 7, illustrating the different roles as well as ways they can communicate with one another[9].

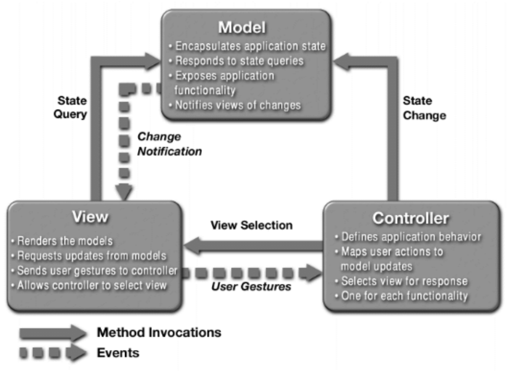

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 7, Box diagram of the Model-View-Controller pattern.

- **Model:** Representing a data container, as well as the business rules defined for accessing and updating the data.
- **View:** The view renders the contents of a model. It is the view's responsibility to maintain consistency in its presentation when the model changes. This can be achieved by using a push model, where the view registers itself with the model

9 From a web page on J2EE patterns by Sun Microsystems, Inc.
URL: "http://java.sun.com/blueprints/patterns/MVC-detailed.html".

for change notifications, or a pull model, where the view is responsible for calling the model when it needs to retrieve the most current data.

- **Controller:** The controller translates interactions with the view into actions to be performed by the model. In a stand-alone GUI client, user interactions could be button clicks or menu selections. The actions performed by the controller include activating business processes or changing the state of the model.

The module plug-in loading mechanism has been designed using this pattern, as illustrated in figure 8.

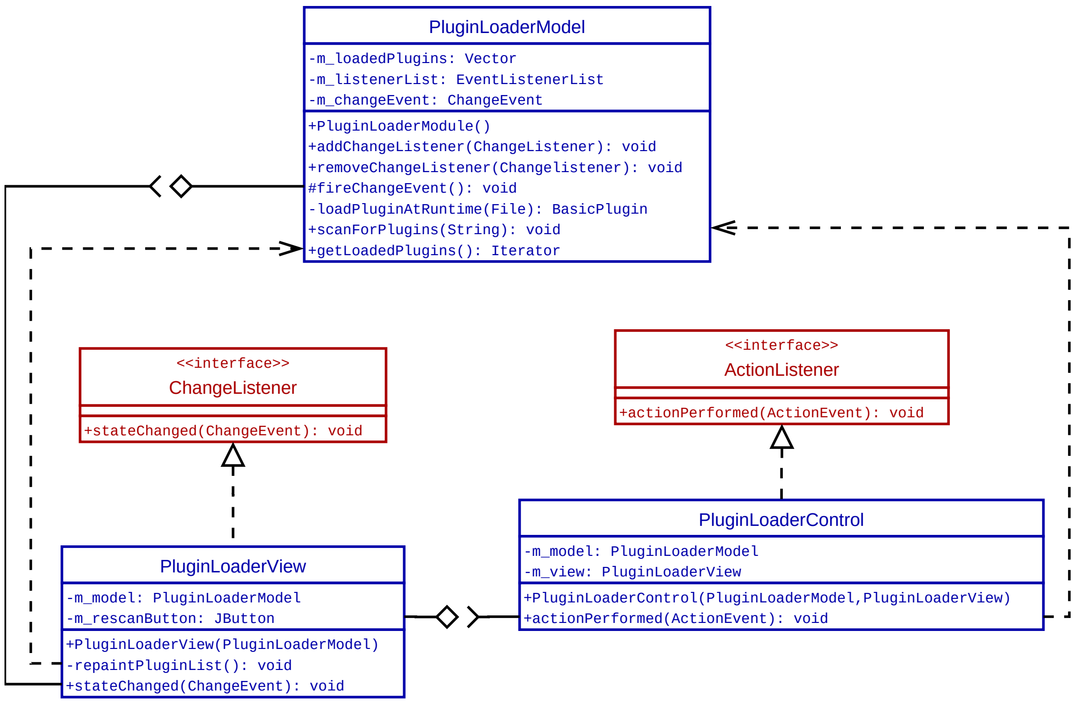

PluginLoaderModel
-m_loadedPlugins: Vector
-m_listenerList: EventListenerList
-m_changeEvent: ChangeEvent
+PluginLoaderModule()
+addChangeListener(ChangeListener): void
+removeChangeListener(Changelistener): void
#fireChangeEvent(): void
-loadPluginAtRuntime(File): BasicPlugin
+scanForPlugins(String): void
+getLoadedPlugins(): Iterator
<<interface>>
ChangeListener
+stateChanged(ChangeEvent): void
<<interface>>
ActionListener
+actionPerformed(ActionEvent): void
PluginLoaderView
-m_model: PluginLoaderModel
-m_rescanButton: JButton
+PluginLoaderView(PluginLoaderModel)
-repaintPluginList(): void
+stateChanged(ChangeEvent): void
PluginLoaderControl
-m_model: PluginLoaderModel
-m_view: PluginLoaderView
+PluginLoaderControl(PluginLoaderModel,PluginLoaderView)
+actionPerformed(ActionEvent): void

Figure 8, UML diagram showing the structure of the Model-View-Controller pattern applied to the plug-in loading mechanism.

In the current system implementation, it is possible to swap between two versions of the pluggable module loading mechanism:

1. PluginLoaderModel: The basic module loading mechanism, assuming that all modules are located in separate Java library (*.jar) files, either on a remote server or on the local file system. This model is mainly used when deploying release versions of the system.
2. PluginInternalModel: A version of the module loader, mainly used under development. This model will load all modules currently available in the development environment, thereby eliminating the need for building *.jar library files for every change made to a module.

Because of the MVC architecture used, the difficulties involved in swapping between the two models are very limited, and does not affect the view nor the controller classes.

2.6 Framework

To be able to provide the versatility required for the system, it was decided to make a solution where any extensions could be developed as small self containing modules, not requiring changes to the rest of the system when integrated.

To achieve this, much effort is put into making the framework code handle as much of the heavy complexity as possible. The idea is to make the framework do all data routing through the system, leaving individual modules (extensions) with their specific processing logic only.

2.6.1 Pluggable modules

On top of the framework, any kind of processing logic should be possible to implement, as long as the implementing classes follow the guidelines laid out by the governing interfaces. An UML diagram, showing these and including two example implementing classes, XmlFileImportPlugin and TranslateRowsPlugin, is provided in figure 9.

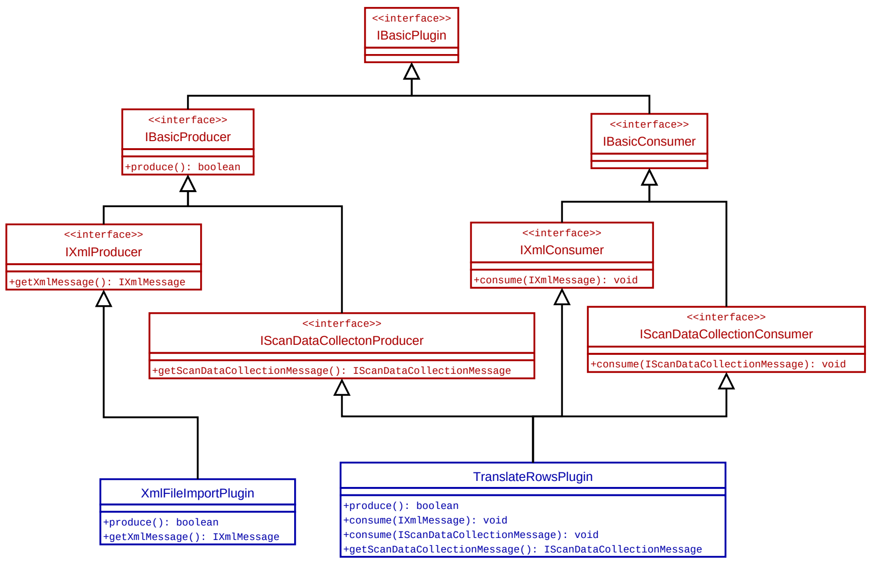

<<interface>>
IBasicPlugin
<<interface>>
IBasicProducer
+produce(): boolean
<<interface>>
IBasicConsumer
<<interface>>
IXmlProducer
+getXmlMessage(): IXmlMessage
<<interface>>
IXmlConsumer
+consume(IXmlMessage): void
<<interface>>
IScanDataCollectonProducer
+getScanDataCollectionMessage(): IScanDataCollectionMessage
<<interface>>
IScanDataCollectionConsumer
+consume(IScanDataCollectionMessage): void
XmlFileImportPlugin
+produce(): boolean
+getXmlMessage(): IXmlMessage
TranslateRowsPlugin
+produce(): boolean
+consume(IXmlMessage): void
+consume(IScanDataCollectionMessage): void
+getScanDataCollectionMessage(): IScanDataCollectionMessage

Figure 9, UML diagram, showing the interfaces to be implemented from when developing pluggable modules.

As new modules are continuously being developed, it would be impossible to cover all modules at the time of reading in this document. However, as part of the master thesis, a number of modules have been produced and tested. These modules will be described in detail in a later chapter.

The modules can either exist as independent packages (*.jar files) on disk or possibly located on a remote server, or they can be integrated in the core system package. This enables users the possibility to get a continuous upgrading of their system, as remote modules could be updated in a centralized way as often as needed (e.g. fix for one and all will be fixed).

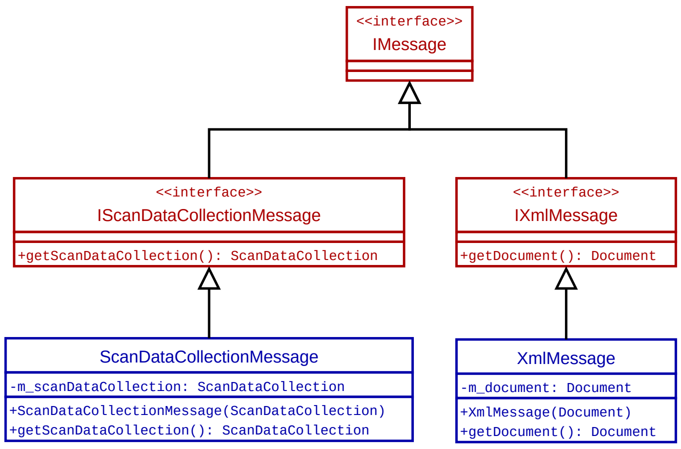
2.6.2 Message channels

The modules are connected through data channels visualized as colored arrows. Currently, the framework supports two types of channels:

- A **ScanDataCollection** (**red** arrows in the diagram, see section on the GUI) channel transporting raw image data in double precision arrays and

- an **Xml** (**blue** arrows in the diagram, see section on the GUI) channel transporting XML encoded information (e.g. a vector of integers, etc.).

When trying to connect two modules in the testbed diagram, looking at the interfaces, implemented by the two modules, the framework will decide which data channel is available for the connection, if any. If more than one type of channel can be made between two modules being connected, the user will be presented with a list to choose from. A UML diagram, showing the channel and message interface relationship, is illustrated in figure 10.



<<interface>>
IMessage
<<interface>>
IScanDataCollectionMessage
+getScanDataCollection(): ScanDataCollection
<<interface>>
IXmlMessage
+getDocument(): Document
ScanDataCollectionMessage
-m_scanDataCollection: ScanDataCollection
+ScanDataCollectionMessage(ScanDataCollection)
+getScanDataCollection(): ScanDataCollection
XmlMessage
-m_document: Document
+XmlMessage(Document)
+getDocument(): Document

Figure 10, The two types of messages, currently available in the system.

2.7 GUI design and implementation

The graphical user interface (GUI) is the part of the framework, the user sees and interacts with to control the system. It should not contain any business logic, but only present what lies in the model code underneath, as well as feeding user commands to the framework.

As the system has been implemented in Java, Swing will be used when implementing the GUI. Swing is based on lightweight components [Geary99, p.6-16], and is directly portable to other platforms, where Java is supported.

In this section, we will run through the different parts of the GUI, including the pluggable module list, the testbed diagram view and the details panel for modules selected in the diagram.

Some design considerations were made regarding usability and separation from the underlying framework code. These areas are covered in the end of this section.

2.7.1 Overview of the user interface

The main user interface for the system has been designed to function as a sort of testbed, where components can be placed and wired together. The components are connected using message channels, capable of transporting different types of data through them. Message channels are visualized as arrows in different colors, going from "producer" modules to "consumer" modules. Figure 11 shows the main graphical view, where a simple diagram has been constructed consisting of a data phantom and a visualization module.

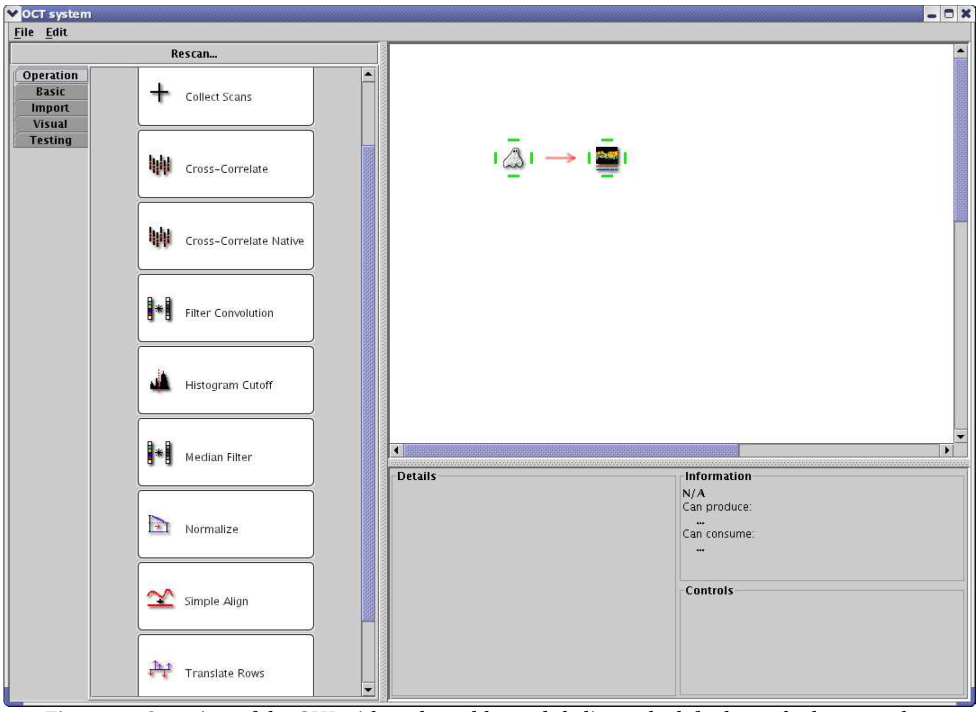


Figure 11, Overview of the GUI with a pluggable module list to the left, the testbed area to the upper right and a panel, showing details for selected modules below.

The pluggable component modules are listed under different categories to the left and new instances of the modules can be dragged to the testbed using the mouse. Connecting two modules, is done by pressing the right mouse button on the “producing” module, dragging a channel to the “consuming” module.

A module may or may not have a details panel associated with it. If so, the details panel will appear under the algorithm testbed area, when a module is selected in the diagram.

2.7.2 Usability

When developing systems that are going to be used by people with different types of background, it is vital, that some effort has been put into making the visible parts user friendly. This area of software development has gained an increasing place of importance - especially as more and more non-technical people get involved in computing. As an example, Luca Passani, one of the 'gurus' in the field of usability, tries to narrow the gap between developers and users, by explaining the possible problems when engineers develop for the masses [Arehart00, chapter 7].

Trying to cover all sorts of aspects of usability when developing a system like this, would be impossible, given the resources available to the project. However, during development, it has been a key issue to keep in mind that the software should not only function, but also try to meet the usability requirements given.

Being the developer of the system, it's hard to give an objective view on if and how the usability requirements were met. On page 125, a statement made by Birgit Sander (the “customer”), explains her view on this issue (in Danish).

2.8 XML for compatibility

The Extensible Markup Language (XML) is a World Wide Web Consortium (http://w3c.org) recommended standard for creating information documents . Unlike HTML, which contains only words and links to pictures with some document formatting, an XML document, in general, contains elements which can be used to define information structures [Berg99, p.562].

2.8.1 Diagram file format

The system supports saving and loading of diagrams in a human readable XML format. This enables the user to easily create other pieces of independent software, that may be used to modify the files, and thereby loosely interact with the platform. A typical minimalistic example, consisting of a raw data phantom connected to a visualization module, saved in a file, can be seen in source listing 3.

The mechanism for storing the state of the diagram, is taking advantage of the fact that XML structures can be nested in a tree structure, and child nodes in the tree only needs to know of the immediate parent node on which to extend. This means, for example, that the Diagram class only needs to know how to handle a <Diagram> element, passing all the handling of <Component> elements to the base class of the components. Each component then decides how to handle its given <Component> element based on the component type ("ModuleModel" or "ChannelModel"), which will again result in a call to a handler in the respective classes.

To give an idea of the effectiveness of this approach, take a look at the part of the code from the RawDataPhantom class, that is responsible for storing and retrieving of state information in source listing 4.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<Diagram>
 <Component type="ModuleModel" componentId="11...:-7fff">
  <PropertyMap>
   <Property name="xpos" value="234" />
   <Property name="ypos" value="124" />
  </PropertyMap>
  <Plugin name="com.dtu.oct.plugin.visualizescans.VisualizeScansPlugin" />
 </Component>
 <Component type="ModuleModel" componentId="11...:-8000">
  <PropertyMap>
   <Property name="xpos" value="120" />
   <Property name="ypos" value="124" />
  </PropertyMap>
  <Plugin name="com.dtu.oct.plugin.rawdataphantom.RawDataPhantomPlugin">
   <Property name="selectedPattern" value="Checker Board" />
  </Plugin>
 </Component>
 <Component type="ChannelModel" componentId="11...:-7ffe">
  <PropertyMap />
  <Channel>
   <Property name="consumerId" value="11...:-7fff" />
   <Property name="producerId" value="11...:-8000" />
   <Property name="messageType" value="ScanDataCollection" />
  </Channel>
 </Component>
</Diagram>
```

Source 3, Contents of an example diagram stored in an XML file. NOTE: component IDs have been shortened to make them fit here.

A quick view on the code, reveals that the only state information handled specifically by the class (RawDataPhantom) is the type of pattern selected.

Getting back to the stored file, here follows an explanation of each of the main structures in the diagram XML:

The XML header (<?xml version="1.0" encoding="UTF-8"?>) is always present in some form in an XML document, giving hints to XML parsers on how the document is stored.

```
public void initialize(Element pluginElement){
        super.initialize(pluginElement);
        System.out.println("RawDataPhantom initialized called...");

        try {
                AttributeNode propertyValue =
                        (AttributeNode)XPathAPI.selectSingleNode(
                                pluginElement,
                                "Property[@name='selectedPattern']/@value");
                if( null != propertyValue){
                        this.setPatternGenerator(propertyValue.getValue());
                        m_mainPanel.updateView();
                }
        } catch (Exception ex){
                System.out.println(""+ex);
        }

}

public void saveState(Element pluginElement){
        super.saveState(pluginElement);
        System.out.println("RawDataPhantom saveState called...");

        Document xmlDoc = pluginElement.getOwnerDocument();

        Element propertyElement =
                (Element)pluginElement.appendChild(
                        xmlDoc.createElement("Property") );
        propertyElement.setAttribute("name","selectedPattern");
        propertyElement.setAttribute("value",""+getPatternGeneratorName());
}
```

Source 4, Seralization logic of the RawDataPhantomPlugin class.

In the files, stored by the system, there should be a root element, <Diagram>, encapsulating different types of components existing in the diagram. At the time of writing, there are two types of components available:

- A ModuleModel, representing a pluggable module (e.g. a RawDataImport module) and
- a ChannelModel, representing a data channel between modules (e.g. a ScanDataCollection message channel).

Each component has an associated componentId, used to be able to uniquely identify a component, e.g. when relating producer and consumer ModuleModels in ChannelModels. This relationship is illustrated in source listing 5.

```
...
 <Component type="ModuleModel" componentId="11...:-7fff">
   <Plugin name="com.dtu.oct.plugin.visualizescans.VisualizeScansPlugin" />
 </Component>
 <Component type="ModuleModel" componentId="11...:-8000">
   <Plugin name="com.dtu.oct.plugin.rawdataphantom.RawDataPhantomPlugin">
     <Property name="selectedPattern" value="Checker Board" />
   </Plugin>
 </Component>
...
   <Channel>
     <Property name="consumerId" value="11...:-7fff" />
     <Property name="producerId" value="11...:-8000" />
     <Property name="messageType" value="ScanDataCollection" />
   </Channel>
...
```

Source 5, Channel components contain references to module components by their component ID.

Before continuing with the element descriptions, it might be helpful with a graphical view of the diagram represented in the file. A simple algorithm, consisting of a data phantom connected to a visualization module, is illustrated in figure 12.

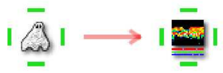


Figure 12, Simple diagram: visualizing a phantom pattern.

As clearly visible in the diagram, the two components are aligned horizontally with a ScanDataCollection message channel arrow in between.

As the model itself does not contain specific View related member variables, the positioning of modules in the visualization of the diagram, is stored in a property map[10] inside the modules.

10 A property map is a (name, value) pair dictionary.

```
...
  <PropertyMap>
    <Property name="xpos" value="234" />
    <Property name="ypos" value="124" />
  </PropertyMap>
...
```

Source 6, Graphical module positions in the diagram view are saved as properties in an independent property map.

A property map is a map of key-value pairs that contain any key related to any value. This way, the view is not tightly coupled to the model code and the model will allow other kinds of views to store their specific information in the map. The XML representation of the map, in the stored workspace file, is shown in source listing 6.

The "Checker Board" pattern is selected in the graphical view (see figure 13), and the serialization code snippet from source listing 4, combined with this selection, results in the XML element listed in source listing 7 when the diagram is saved.

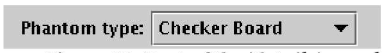

Phantom type: Checker Board

Figure 13, Part of the 'details' panel for the phantom pattern generator module.

```
...
<Property name="selectedPattern" value="Checker Board" />
...
```

Source 7, The 'selectedPattern' property of the RawDataPhantom is set to “Checker Board” in the XML file.

In the end of this section, a small example is provided, of how the use of XML makes it easy to extend the system with new features to be saved in the workspace file. Late in the project, while designing a very large algorithm diagram, it became obvious, that there was a need for a possibility of labeling the module icons in the diagram, as it became too complex to remember the settings for each module when trying to get an overview.

This kind of functionality was to be available for all kinds of modules, including new ones being created in the future, and it shouldn't involve any code changes to the individual module implementations. Using the existing mechanism for getting and setting properties for modules (like in the case of the positioning of the icons, as mentioned before), the implementation was limited to two places in the code:

1. The user interface for setting the module label, added next to the “Consume”/”Produce” buttons in the plug-in details panel (around 15 lines of code).
2. When painting the icons in the visual diagram, the label text needed to be shown (around 10 lines of code).

Both places, the code would communicate directly with the generic code for the modules (the ModuleModel class), which would from then on automatically handle storing and retrieval of module labels. In source listing 8, the two lines of code inserted are shown.

In the GUI code, where the user can set the label (nameTag):

```
m_module.setProperty("oct_nameTag", m_txtNameTag.getText());
```

The visualization code would grab the label using:

```
String nameTag = getModel().getProperty("oct_nameTag");
```

Source 8, Example of how the developed get/set property mechanism can ease the addition of new state information.

Looking at a snippet of one of the new XML files produced in source listing 9, the new property, oct_nameTag, containing the label information, is automatically inserted after the code change.

```
…
 <Component type="ModuleModel"
componentId="11d1def534ea1be0:78dc4c:fa64ad2736:-7ffd">
   <PropertyMap>
     <Property name="xpos" value="175" />
     <Property name="ypos" value="126" />
     <Property name="oct_nameTag" value="Mean 5" />
   </PropertyMap>
   <Plugin name="com.dtu.oct.plugin.filterconvolution.FilterConvolutionPlugin" />
 </Component>
…
```

Source 9, Snippet from a saved XML workspace file, containing the newly created "oct_nameTag" property.

Old workspace files, not including this label, was still fully compatible with the system, and new files (with the label set) would also load on the previous version of the system.

2.8.2 Module intercommunication

To make the systems inner workings as flexible and open as possible, some of the modules can be split up in two (or more) parts, communicating with each other using XML. Some image processing module, for example, could be split up in a data analysis part and a data modification part, where the data modification module would take two inputs:

- A ScanDataCollectionMessage, containing the data and
- an XmlMessage, containing the data modifier.

This approach has some advantages:

- When developing similar data analyzing modules, it would be possible to reuse the corresponding data modifying module, thereby cutting down development time as well as limiting the amount of possible introduced bugs in the system.
- Sometimes, the data to be analyzed might have gone through some data preprocessing modules, leaving the data polluted, yet suitable for that specific type of analysis. In these cases, it is preferable to have the possibility of applying

the produced data modifiers to an earlier version of the data.

- If the data analyzing modules available are inadequate in some way for a specific problem, it would then be possible to import the data modifier as an XML file, produced by some other entity (e.g. MatLab).
- A possibility to export the produced data modifiers to be used in other software packages or to be used in documentation in general.

2.9 Tutorial on making pluggable modules

This section will go through a step-by-step guide on how to make a new module beginning with the problem of having identified an algorithm, that cannot be constructed using available pluggable modules.

Before creating anything, the software packages used for development, as well as the code base for the system itself, must be installed.

2.9.1 Identifying the problem

In some cases, it might be desirable to invert an image, e.g. for printing, where a dark foreground on a bright background in general uses less ink than the opposite.

After some experimentation with the system, it is realized, that inversion of pixel values (multiplying them by -1) is not possible given the existing modules. It is therefore desirable, to create a new module, doing just that.

2.9.2 Prerequisites

First, the appropriate software, listed in table 7, needs to be downloaded.

Software	***URL***
Java 2 SDK version 1.4 (or higher)	http://java.sun.com
Eclipse Java IDE (this IDE is not required but is assumed to be used throughout the tutorial).	http://www.eclipse.org
Xalan Java 2 (Extended XML tools including XPath and XSLT functionality).	http://xml.apache.org

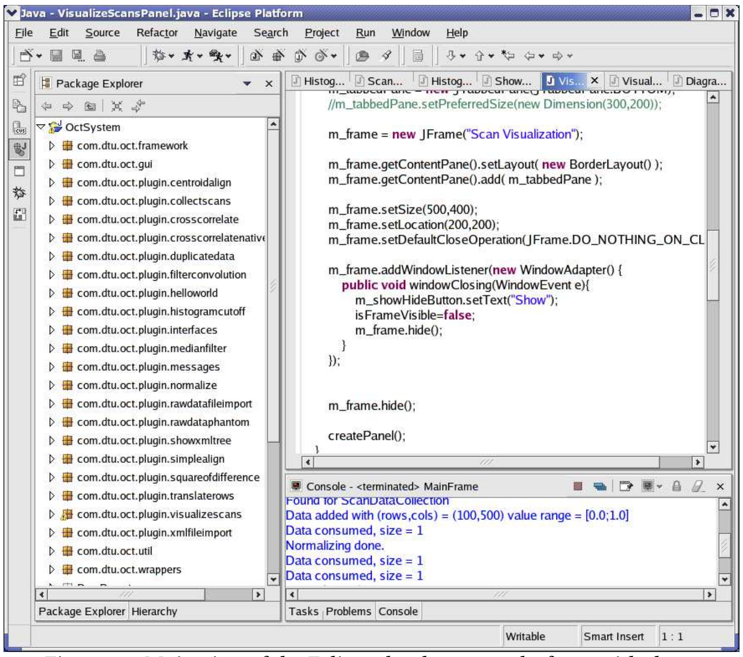
Table 7, Software needed to complete the tutorial. The Java2 platform and Xalan are also needed when using the system at runtime.

Native libraries require a C/C++ compiler to build. On Unix/Linux systems, any recent version of gcc will suffice (gcc3.2 was used during under development of the system). On the MS Windows platform, Visual Studio 6.0 has been tested to work, building shared libraries (*.dll) compatible with the OCT Imaging system.

> **Note:** Some parts of the Blitz++ vector math library will not compile correctly with the Microsoft compiler, as Visual Studio 6.0 is not fully compatible with the template use in Blitz++.

- Install the Java SDK, then Eclipse and add the extra libraries from Xalan under the Java external libraries directory ($JAVA_HOME/jre/lib/ext).
- Now, start Eclipse.
- The source code for the system can be imported into Eclipse using the import tool available under **File->Import**.

After successfully getting the IDE up and running, expand the OCT project, open a few classes, and a view, similar to the one in figure 14, should appear.


Java - VisualizeScansPanel.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Histog... Scan... Histog... Show... Vis... Visual... Diagra...
Console - <terminated> MainFrame
Package Explorer Hierarchy
Tasks Problems Console
Writable Smart Insert 1 : 1

```
OctSystem
  com.dtu.oct.framework
  com.dtu.oct.gui
  com.dtu.oct.plugin.centroidalign
  com.dtu.oct.plugin.collectscans
  com.dtu.oct.plugin.crosscorrelate
  com.dtu.oct.plugin.crosscorrelatenative
  com.dtu.oct.plugin.duplicatedata
  com.dtu.oct.plugin.filterconvolution
  com.dtu.oct.plugin.helloworld
  com.dtu.oct.plugin.histogramcutoff
  com.dtu.oct.plugin.interfaces
  com.dtu.oct.plugin.medianfilter
  com.dtu.oct.plugin.messages
  com.dtu.oct.plugin.normalize
  com.dtu.oct.plugin.rawdatafileimport
  com.dtu.oct.plugin.rawdataphantom
  com.dtu.oct.plugin.showxmltree
  com.dtu.oct.plugin.simplealign
  com.dtu.oct.plugin.squareofdifference
  com.dtu.oct.plugin.translaterows
  com.dtu.oct.plugin.visualizescans
  com.dtu.oct.plugin.xmlfileimport
  com.dtu.oct.util
  com.dtu.oct.wrappers
```

```
    //m_tabbedPane.setPreferredSize(new Dimension(300,200));

    m_frame = new JFrame("Scan Visualization");

    m_frame.getContentPane().setLayout( new BorderLayout() );
    m_frame.getContentPane().add( m_tabbedPane );

    m_frame.setSize(500,400);
    m_frame.setLocation(200,200);
    m_frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CL

    m_frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e){
        m_showHideButton.setText("Show");
        isFrameVisible=false;
        m_frame.hide();
      }
    });


    m_frame.hide();

    createPanel();
  }
```

```
Found for ScanDataCollection
Data added with (rows,cols) = (100,500) value range = [0.0;1.0]
Data consumed, size = 1
Normalizing done.
Data consumed, size = 1
Data consumed, size = 1
```

Figure 14, Main view of the Eclipse development platform with the OctSystem project open.

2.9.3 Beginning development

To the left, a long list of package names is displayed – many of them with the word “plugin” in the middle.

- Right click on one of the packages and select New->Package.
- Name the new package: “com.dtu.oct.plugin.invertimage” (see figure 15).

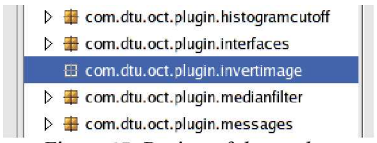
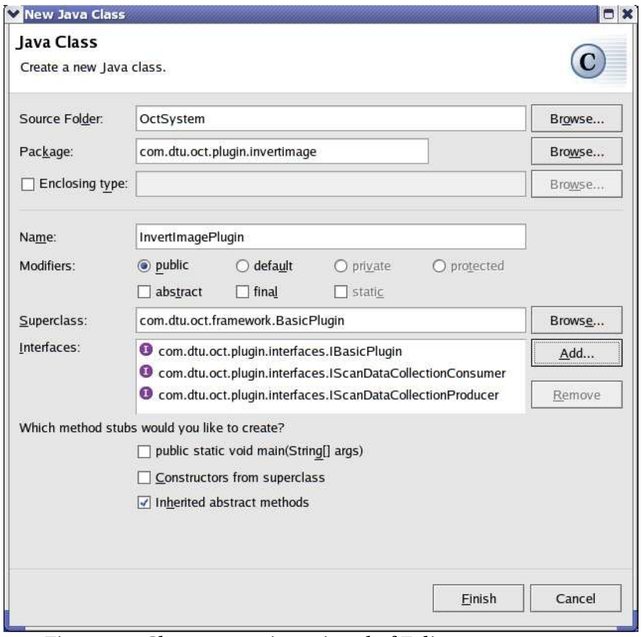

com.dtu.oct.plugin.histogramcutoff
com.dtu.oct.plugin.interfaces
com.dtu.oct.plugin.invertimage
com.dtu.oct.plugin.medianfilter
com.dtu.oct.plugin.messages

Figure 15, Region of the package and class view of Eclipse, with the newly generated package selected.

- Right click on the new package name and select New->Class
- Fill in the information as shown in figure 16 and press “Finish”.


New Java Class
Java Class
Create a new Java class.

Source Folder: OctSystem
Package: com.dtu.oct.plugin.invertimage
[ ] Enclosing type:

Name: InvertImagePlugin
Modifiers: [● public] ○ default ○ private ○ protected
[ ] abstract [ ] final [ ] static
Superclass: com.dtu.oct.framework.BasicPlugin
Interfaces:
com.dtu.oct.plugin.interfaces.IBasicPlugin
com.dtu.oct.plugin.interfaces.IScanDataCollectionConsumer
com.dtu.oct.plugin.interfaces.IScanDataCollectionProducer

Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods

Figure 16, Class generation wizard of Eclipse.

The reason for the interfaces are as follows:

- **IBasicPlugin:** Enforces basic plug-in functionality to be present.
- **IScanDataCollectionConsumer:** Making sure, that the module will be able to consume RAW data.
- **IScanDataCollectionProducer:** Enforces production methods to be present, making sure, RAW data will be produced.

To make the plug-in module work in the framework, implement the constructor and basic info methods as shown in source listing 10.

```
...
// the constructor is empty
public InvertImagePlugin(){
}

// the plugin should have a name
public String getName(){
        final String name = "Invert Image";
        return name;
}

// the category this plugin belongs in
public String getType(){
        return "Operation";
}

// a short description
public String getDescription(){
        final String desc = "Invert image pixel values.";
        return desc;
}
...
```

Source 10, Basic constructor and informational methods of the InvertImagePlugin class.

While using the module during development, the module can be added to the list of plug-ins to be loaded automatically at system startup (as opposed to making a new *.jar library file every time a change is made). This modification is illustrated in source listing 11.

```
...
this.addPlugin(new TranslateRowsPlugin(), "/images/translateRowsPlugin_32.png");
this.addPlugin(new VisualizeScansPlugin(), "/images/visualizeRgbPlugin_32.png");
this.addPlugin(new XmlImportPlugin(), "/images/importPlugin_32.png");
this.addPlugin(new InvertImagePlugin(), "/images/defaultPlugin_32.png");

// tell the listeners that the model changed
fireChangeEvent();
...
```

Source 11, Adding the created plug-in to the list of automatically loaded modules in the PluginInternalModel class.

The shell of the module is now fully implemented, and development of the actual processing logic can be done.

2.9.4 Implementing processing logic

When the IScanDataCollectionMessage consumer and producer interfaces were selected in the class creation window, some empty functions were automatically added to the class:

- "**void** consume(IScanDataCollectionMessage message)": The method called by the framework to deliver a ScanData collection to the module.
- "IScanDataCollectionMessage getScanDataCollectionMessage()": The framework calls this method, when processed data has to be fetched from the module again.
- "**boolean** produce()": This method is invoked, when the "Produce" button is pressed, or the framework is doing a back-propagation sequence to force all "parents" of a module in the diagram graph to produce data.

These are the functions, needed to be concentrated on, when implementing the processing logic. In the case of the module in this tutorial, both the consume and the getScanDataCollectionMessage functions are fairly simple to implement, as their job will mainly consist of assigning the incoming ScanData collection reference (to the object) to the member variable and vice versa (see source listing 12).

```
...
ScanDataCollection m_dataCollection = null;

// a press on the consume button will invoke this method - providing incoming data
public void consume(IScanDataCollectionMessage message) {
        // get a reference to the incoming data
        m_dataCollection = message.getScanDataCollection();
}

// this method will be called by the framework to return processed data
public IScanDataCollectionMessage getScanDataCollectionMessage() {
        ScanDataCollectionMessage message = new ScanDataCollectionMessage(null);
        if( isDataReady() ){
                message = new ScanDataCollectionMessage(m_dataCollection);
        }
        return message;
}
...
```

Source 12, Functionality to support the framework in communicating with the module over ScanDataCollection message channels.

The actual processing happens in the “produce“ function, where all incoming ScanData objects will be pixel-wise inverted. Source listing 13 shows the code needed to actually perform the operation. This includes the loop, iterating over the incoming ScanData collection, fetching of the internal data array and handling of properties.

```
...
// a press on the 'Produce' button in the gui will invoke this method
public boolean produce() {
        boolean result = true;
        Iterator it = m_dataCollection.getIterator();
        while(it.hasNext()){
                ScanData data = (ScanData)it.next();
                double[] rawData = data.getData();
                try {
                        // get original max/min values
                        double maxValue = ((Double)
                                (data.getProperty("maxvalue"))).doubleValue();
                        double minValue = ((Double)
                                (data.getProperty("minvalue"))).doubleValue();

                        // invert image
                        for(int i=0;i<rawData.length;i++){
                                rawData[i] = -rawData[i];
                        }

                        // swap positions of min/max value
                        data.addProperty("maxvalue",new Double(-minValue));
                        data.addProperty("minvalue",new Double(-maxValue));
                } catch (Exception e){
                        System.out.println(""+e);
                }
        }
        setDataReady(true);
        return result;
}
...
```

Source 13, All ScanData objects in a collection are pixel-wise inverted when the "produce" method of the InvertImage module is called.

2.9.5 Testing the module

When finished developing a new module, one is of course eager to see if it works. Try to start the system and find the module under its assigned category, “Operation”. After verifying, that the module is indeed present in the overview panel to the left, use the left mouse button to drag an instance of it to the testbed area, as illustrated in figure 17.

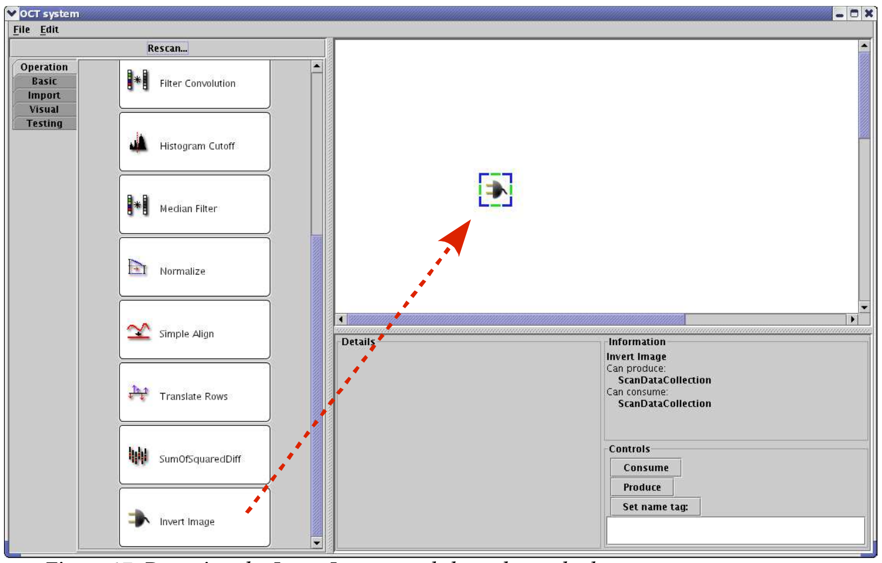


Figure 17, Dragging the InvertImage module to the testbed area.

At this point, the algorithm is completed by adding the following three extra modules:

- 1 RAW data file import module,
- 1 Data duplication module and
- 1 Scan data visualization module.

Connect them as shown in figure 18.

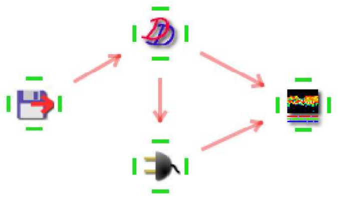


Figure 18, Simple diagram to test the freshly created "InvertImage" module. Note: The module has a default icon, as no custom icon has been assigned.

Select the RAW data file import module and select to load a file, exported from an OCT scanner system. Then select the visualization module and press “Consume” to start the back-propagating enforcement of data production (throughout the diagram graph).

When done, press the “Show” button and inspect the two resulting images, shown in figures 19 and 20.

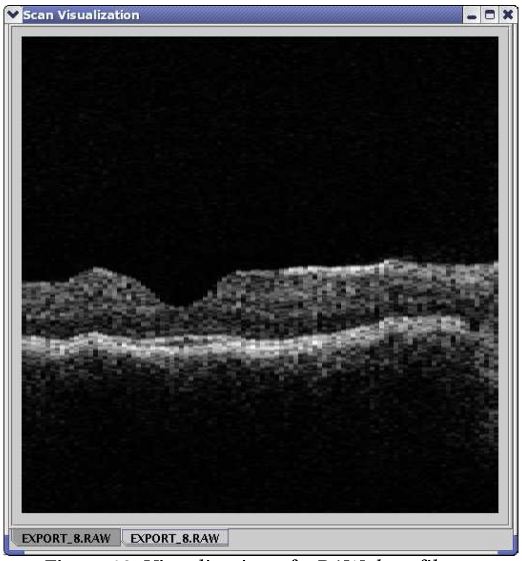


Figure 19, Visualization of a RAW data file, loaded from disk.

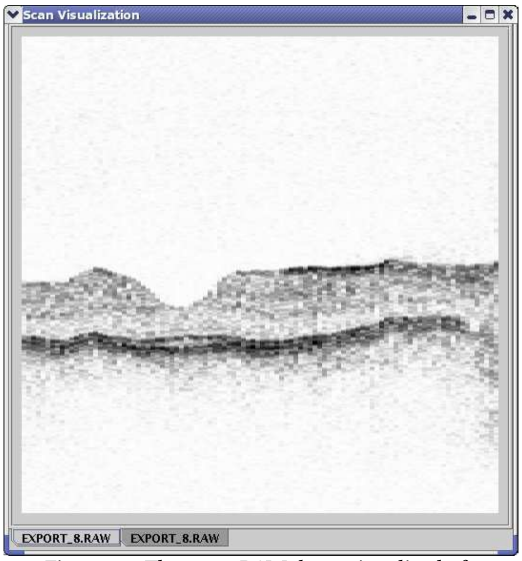


Figure 20, The same RAW data, visualized after going through the new InvertImage module.

By visually inspecting the result of the inversion in figure 20, compared to the original in figure 19, it seems that the ImageInvert module does what it is supposed to.

2.10 Integrating with C/C++

One of the nice features in Java, is its ability to integrate with native (platform dependent) code. In most cases, this is used as a way to speed up performance, as Java, being platform independent, can't be optimized for specific CPU architectures. One of the preferred native programming languages is C++, as it provides an excellent base for the advanced developer to create very versatile platform portable source code.

2.10.1 Using the Java Native Interface

The Java Native Interface (JNI) is a part of the Java SDK and allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly.

Programming through the JNI framework lets you use native methods to do many operations. Native methods may represent legacy applications, or they may be written explicitly to solve a problem, that is best handled outside of the Java programming environment.

For the purpose of this project, the JNI has been used to integrate with code written in C++, mainly for performance reasons.

2.10.2 The power of templates

A key feature of C++ is support for template programming. Templates are normally used to generalize function implementations for many data types. At compile time, these template functions are evaluated for all implementing code using them. If a matching template is found, the compiler constructs a function tailored to the calling code. A typical example of a template function could be the implementation of the 'max' operation shown in source listing 14.

```
template<class T>
int max(T a, T b)
{
  int result = 0;
  if(a < b)
  {
    result = -1;
  }
  else if(b < a)
  {
    result = 1;
  }
  return result;
}
```

Source 14, The 'max' function implemented generically with the use of templates.

If somewhere in the code, a max(T a, T b) is called, where T could be any type available, e.g. double, int, float, etc., the template function is compiled and linked for this specific type. Generation of the specialized functions require the '<' (less than) operator to be defined for the type 'T'.

In this project, we are interested in a specific area of the possibilities of template support in C++ compilers, called metaprogramming. The idea of a metaprogram is to program a program or, in other words, to "lay out code that the programming system executes to generate new code that implements the functionality we really want" [Josuttis02, p.301]. The functionality behind metaprogramming, can be compared to that of functional programming languages like Lisp, where endless recursion, endless arrays, functions calling functions and no variables are present.

The power function class, in source listing 15, is a nice example of how metaprogramming can improve runtime performance by doing calculations at compile time.

```
#ifndef POW_H
#define POW_H

template<int T, int N>
class Pow {
  public:
    static int const result = T * Pow<T, N-1>::result;
};

template<int T>
class Pow<T, 0> {
  public:
    static int const result = 1;
};

#endif
```

Source 15, Implementing the 'power' function using template metaprogramming.

To use the function, one simply needs to get a templated reference to the static 'result' member constant in the 'Pow' class, and the compiler will go through an iterative process, unraveling 'Pow::result' values until 'N' becomes zero, where the function will stop the iterations and return '1' as the last factor. An example of how this would be called from the code can be seen in source listing 16.

```
#include <iostream>
#include "pow.h"

using namespace std;

int main()
{
  cout << "Pow<2,3> = "<< Pow<2,3>::result << endl;
  cout << "Pow<4,3> = "<< Pow<4,3>::result << endl;
  return 0;
}
```

Source 16, Using the Pow function by making a reference to the internal result member.

Output from compiling and running the program is shown in text listing 1.

It is very important to note, that the “pow” calculation is not done at program runtime, but 2*2*2 = 8 and 4*4*4 = 64 is actually calculated by the compiler and assigned to a constant integer at compile time. The program execution therefore basically just prints a constant value.

```
$ g++ pow.cpp -o pow
$ ./pow
Pow<2,3> = 8
Pow<4,3> = 64
```

Text 1, Output from compiling and running the Pow function example.

One might think that this could have been handled easier by just calling some built in math function, but the real benefits come when scaling the amount of data to be processed to something in the order of what we're dealing with in the OCT case – both in execution time and low complexity in the code.

While developing the project, a template based math library, Blitz++, was used to do fast vector calculations. Blitz++ gets a heavy performance boost, mainly because of the nature and capabilities of metaprograms, basically eliminating the use of temporary internal variables. This brings blitz++ to a level, where it can sometimes outperform programs written in Fortran[11].

2.11 Taking the time needed to rewrite and refactor

Sometimes, the best way to do things is not the most obvious at the time where decisions are made to choose the path of development. That, however, does not mean that you should just accept a suboptimal solution, when more information is available, showing this fact. When developing reliable software, it is vital to put one's pride aside, and accept the fact that people make mistakes. In my time as a professional software developer, I have seen many projects go bad, mainly because of stubborn project managers or system designers not accepting, that what they

11 "What is Blitz++", OONumerics, URL: http://www.oonumerics.org/blitz/whatis.html

thought to be right at project start might not be true at a later stage. This is not necessarily resulting from a lack of design competence, but because you just can't predict the future.

Over the years, many processes have been developed to help to prevent this from happening. Extreme Programming (XP), being one of the more famous of these so called agile processes, describe methods and guidelines to deal with these issues.

Many thoughts and methods relating to the whole XP philosophy can be found in the introductory book, “Extreme Programming Explained: Embrace Change”, by Kent Beck. Covering all aspects would be outside the scope of this project, but just to give an idea of how XP takes commonsense to extreme levels, here is a short list of some of the core principles [Beck00]:

- “If code reviews are good, we'll review code all the time.
- If testing is good, everybody will test all the time, even the customers.
- If design is good, we'll make it part of everybody's daily business (refactoring).
- If simplicity is good, we'll always leave the system with the simplest design that supports its current functionality (the simplest thing that could possibly work).
- If architecture is important, everybody will work defining and refining the architecture all the time.
- If integration testing is important, then we'll integrate and test several times a day.
- If short iterations are good, we'll make the iterations really, really short – seconds and minutes and hours, not weeks and months and years.”

Most of these principles are meant to be used by large development teams, but some of the ideas have been kept in mind under development of this system.

Doing refactoring, to make the code more simple and easier to change, is always important[Beck00, p. 58]. As an example, the initial implementation of the framework had a lot of redundant code, as it was going through its prototyping phase, but when most of the functionality seemed to work, much development time was spent on refactoring the code to make it faster, more reliable and easier to change.

2.12 Summary

A lot of steps are involved in developing high quality software. Some of them might seem ridiculous when first encountered, but after a while, software development will never seem the same without them.

In this chapter, some of these steps have been described, as well as providing background information on the processes involved in developing the image processing system related to this report. Hopefully, this has given a good foundation for understanding the inner workings of the framework.

Producing a set of valid system requirements, is always a good starting point for development. Without them, it can become very hard to reach the goals set for development to a satisfactory level.

When developing flexible systems, designed for change, one can enjoy the benefits of using Design Patterns in producing the system architecture, developed to help with providing solutions to recognizable design problems.

3. Pluggable modules provided

The following sections function as a sort of module reference manual for users of the system, as well as providing a theoretical background for the pluggable modules provided as a part of the project.

With each module description, you will find the following:

- An short table containing an overview, including name, graphical image icon (in the GUI), a short description, as well as a list of possible input and output channels supported by the module.
- A theory section providing background information, about the underlying mathematical or technical methods used.
- Implementation specific details, including code snippets where seen fit.
- A results section, containing usage examples and tests showing the module working in a context.

3.1 Collect Scans module

Name	CollectScans
Graphical Icon	+
Consumes messages of type	ScanDataCollection
Produces messages of type	ScanDataCollection
Description	This module will perform a pixel-wise addition of all incoming RAW data scans. The images must be of same dimension.

Table 8, Overview of the Collect Scans module.

Theory/background

An operation that will do basic pixel-wise addition of images, is a desirable feature to have for several reasons, e.g.:

- It can be used when trying to reduce the noise by collecting scans of the same data [Carstensen97, p.39-40] (see chapter 4 for an in-depth example).
- When trying to visualize difference images – e.g. to visualize changes in acquired scans over time – the series of images can be superimposed to create a motion shadow effect.
- Combining phantom generated patterns to form new patterns, suitable for testing newly generated image processing modules.

Requirements

The module needs to consume and produce a ScanDataCollection message. The incoming collection must consist of one or more ScanData (image) elements. The outgoing message consists of a ScanDataCollection message containing the stacked/superimposed ScanData.

Implementation

Some basic arithmetic operations are implemented in the ScanData class itself, as seen in source listing 17).

```
...
public void add(ScanData value) throws Exception {
        if(m_data.length != value.getData().length){
                throw new Exception("ScanData lengths do not match!");
        }

        double[] valueData = value.getData();

        for(int i=0;i<m_data.length;i++){
                m_data[i] += valueData[i];
        }
}

public void sub(ScanData value) throws Exception {
        if(m_data.length != value.getData().length){
                throw new Exception("ScanData lengths do not match!");
        }

        for(int i=0;i<m_data.length;i++){
                m_data[i] -= value.getData()[i];
        }
}

public void add(double value){
        for(int i=0;i<m_data.length;i++){
                m_data[i] += value;
        }
}

public void mul(double value){
        for(int i=0;i<m_data.length;i++){
                m_data[i] *= value;
        }
}
...
```

Source 17, Pixel wise arithmetic operations implemented in the ScanData class.

When the CollectScans module consumes a ScanDataCollection message on it's incoming channel, all contained ScanData objects (images) are added pixel-wise, and the summed image put into a new object (see source listing 18).

```
...
while(it.hasNext()){
          ScanData data = (ScanData)it.next();
          if( 0 == count ){
                    dataCopy = new ScanData(data);
          } else {
                    dataCopy.add(data);
          }
          count++;
}

double[] rawData = dataCopy.getData();
double maxValue = rawData[0];
double minValue = rawData[0];

for(int i=0;i<rawData.length;i++){
          if(rawData[i]>maxValue)maxValue=rawData[i];
          if(rawData[i]<minValue)minValue=rawData[i];
}

dataCopy.addProperty("maxvalue",new Double(maxValue));
dataCopy.addProperty("minvalue",new Double(minValue));
...
```

Source 18, Adding all incoming ScanData objects in a collection followed by an evaluation of the minimum and maximum pixel values in the outgoing data.

Usage/Results

The CollectScans module will pixel-wise add all images on incoming RAW data * (ScanDataCollection) channels. In this example we will take a checker board pattern, generated by the RawDataPhantom, and add some salt & pepper noise to it.

Design a small algorithm consisting of:

- 2 raw data phantom modules,
- 1 scan collection module and
- 1 visualization module.

Connect the modules as shown in figure 21.

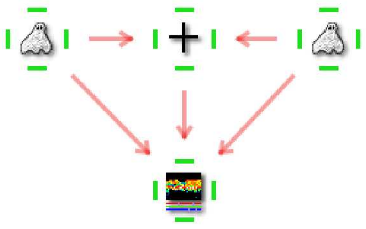


Figure 21, Simple algorithm, used to illustrate the use of the Collect Scans module.

Set one of the data phantoms to produce a “Checker Board” pattern, and the other to produce a “Salt & Pepper Noise” pattern.

Select the visualization module, press “Consume”, then “Show” to see the results illustrated in figures 22-24:

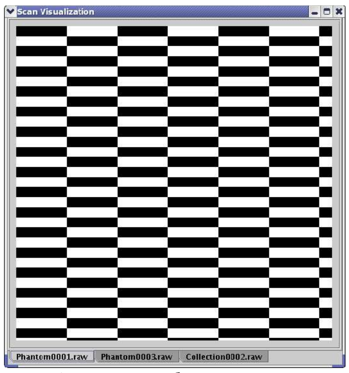


Figure 22, A phantom generated "Checker Board" pattern.

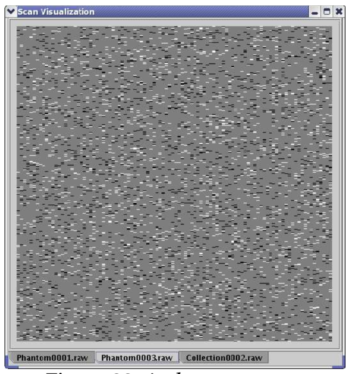


Figure 23, A phantom generated "Salt & Pepper Noise" pattern.

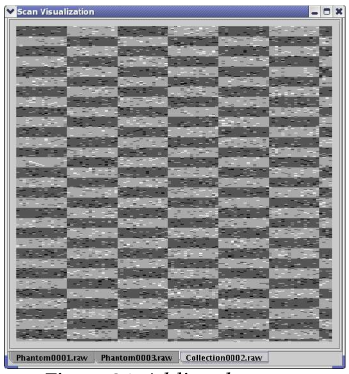


Figure 24, Adding the two images, produces a noisy checker board pattern.

Figure 24 shows the combined image, were noise is applied to a checker board pattern.

3.2 Cross-Correlate module

Name	Cross-Correlate
Graphical Icon	
Consumes messages of type	ScanDataCollection
Produces messages of type	Xml
Description	Performs a series of cross-correlations between rows in the incoming data. Based on results of these correlations, an optimal alignment vector is produced.

Table 9, Overview of the Cross-Correlate module.

Theory/background

When trying to align A-scans in a noisy (and jumpy) acquired image, a method is needed for measuring how much each A-scan row should be translated to make it align with the others next to it.

Consider how two data signals, equal in length and possibly sharing waveforms, might be compared. A good estimate of how alike the signals are would be to take the sum of the products obtained by multiplying the two signals point for point [Ifeachor98, p.184]. If the two signals are out of phase (not aligned), the maximum sum would be found by translating one of the signals to be aligned with the other before doing the calculations (see figure 25). This phase shift corresponds to pixel-wise row translation in digital images.

For alignment purposes, the interest is focused on finding the offset of this translation, as stated above, corresponding to the index of the maximum value found in the vector resulting from cross-correlating subsequent rows in the image data.

If, by any chance, a row (A-scan) is missing from the data, the Cross-Correlation module should instead use the closest existing row for comparison.

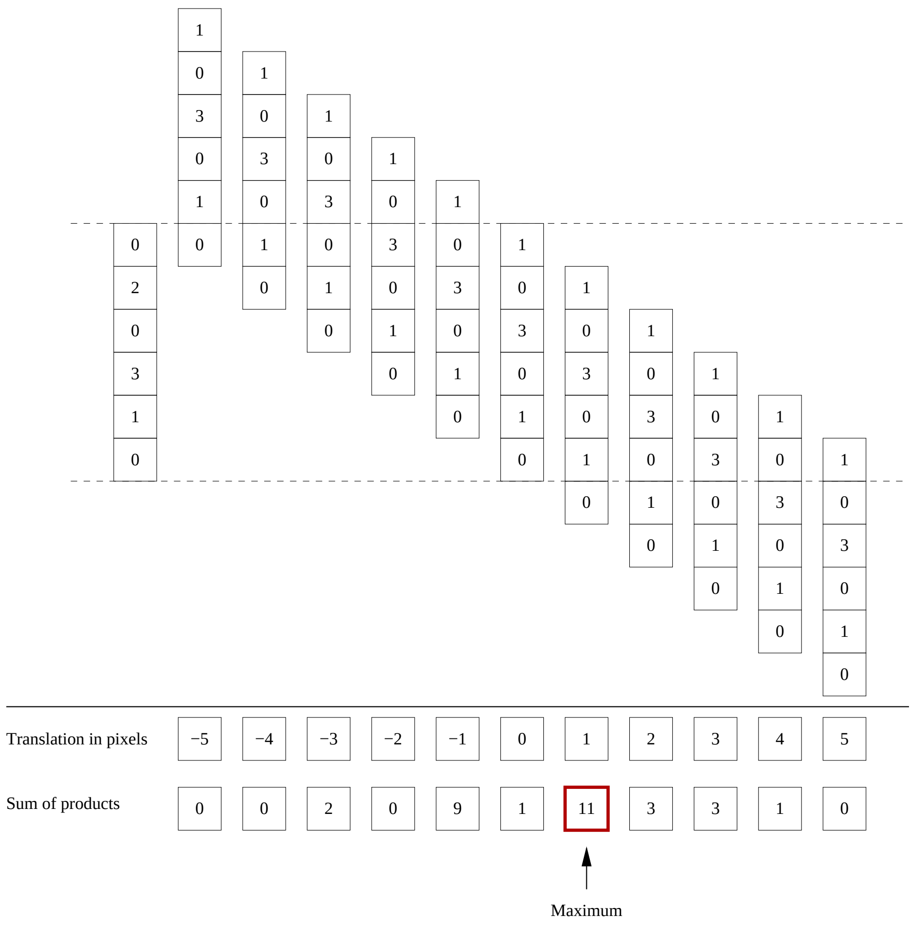


Figure 25, Illustrating how the best match is found by translating a signal to find the maximum sum of squares when doing pixel-wise multiplications.

In some cases, it is desirable to be able to align a set of images in a ScanData collection in such a way that the first image in the collection functions as a template for how subsequent images should be aligned. This, for example, is a necessary feature to have when trying to improve the signal/noise ratio by superimposing RAW data images (see chapter 4).

Requirements

The module must be able to perform a cross-correlation for alignment in at least two distinct ways:

1. Between coherent A-scans in an acquired image.
2. Between corresponding A-scans in a series of acquired images.

The result should be an XML message containing the calculated optimal translation vectors for the incoming ScanData collections. Element values in the translation vector should be based on results found when selecting the index of the maximum value in the vector, produced by cross-correlating coherent data rows within or between images in a collection.

Implementation

Although Java is not the optimal choice for doing vector math operations, for platform compatibility reasons, a pure Java version of the cross-correlation mechanism described has been implemented. It is, however, strongly suggested that the native version of the module is used when dealing with large amounts of data.

The two more interesting parts of the implemented code are:

1. The actual cross-correlation and maximum index selection implementation: "calcMaxValueIndexCrossCorrelation(**double**[] signal1, **double**[] signal2)" and
2. the place in the code, where the resulting XML vector is forged.

A dissection of the code begins, starting with the place where the cross-correlation vector is calculated from the two data signals, originating from rows in the RAW data, seen in source listing 19.

```
...
int beginOffset = 1-signal2.length;
int endOffset = signal1.length-1;

int crossIndex = 0;

for(int offset=beginOffset;offset<=endOffset;offset++){
        double crossValue = 0.0;

        for(int i=0;i<signal1.length;i++){
                int j = i + offset;
                if(j>=0 && j<signal2.length){
                        crossValue += signal1[i]*signal2[j];
                }
        }

        signalCross[crossIndex] = crossValue;
        crossIndex++;
}
...
```

Source 19, Code snippet showing the Java implementation of the cross-correlation function.

What happens, is that a data array (signalCross) is filled with values resulting from a summation of products (crossValue += signal1[i]*signal2[j]) between coherent pixel values of signal1 and signal2, where signal2 is phase shifted by an offset.

After calculating for all offsets (possible phase shift values), signalCross contains the cross-correlation between signals 1 and 2. When the cross-correlation between the two signals has finished, the index of the maximum value in the cross-correlation array is found and returned (see source listing 20). The length of signal2 minus one is subtracted from the result to compensate for border effects.

```
...
        double maxValue = signalCross[0];
        int maxIndex = -1;

        for(int i=0;i<signalCross.length;i++){
                if(signalCross[i] > maxValue){
                        maxValue = signalCross[i];
                        maxIndex = i;
                }
        }

        result = maxIndex + beginOffset;

        return result;
...
```

Source 20, Finding the maximum value in the cross-correlation array and returning the index.

Usage/Results

A simple sine wave pattern will be used to demonstrate the alignment functionality of the cross-correlation module.

Design a small algorithm consisting of:

- 1 raw data phantom module,
- 1 cross-correlation module,
- 1 row translation module and
- 1 visualization module.

Connect the modules as shown in figure 26. Then select the “Sine” pattern on the details page for the data phantom, then click on the cross-correlation module and choose to align rows within images (see figure 27).

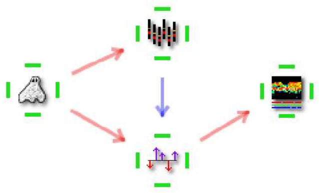
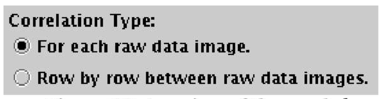


Figure 26, Aligning a phantom generated pattern, using the cross-correlation module.



Correlation Type:

● For each raw data image.

○ Row by row between raw data images.

Figure 27, A region of the module details panel for the Cross-Correlation module.

Use the visualization module to see the results by pressing “Consume”, then “Show” (see figures 28 and 29).

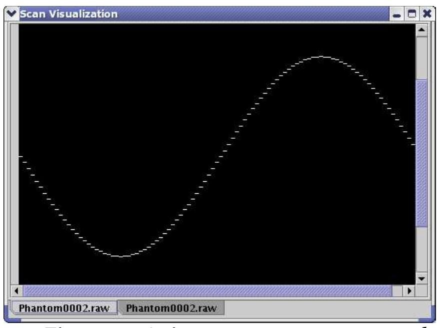


Figure 28, A sine wave pattern generated by the data phantom.

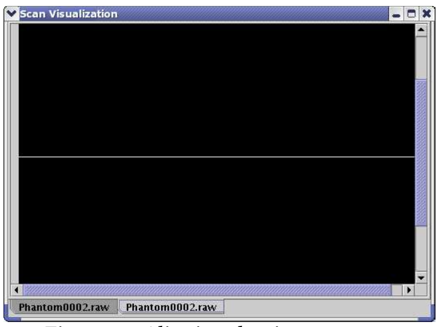


Figure 29, Aligning the sine wave pattern using the cross-correlation module results in a straight line.

Figure 28 shows the sine wave pattern, consisting of what would correspond to a series of unaligned A-scans. The result after alignment, seen in figure 29, is a straight line, as expected.

3.3 Cross-Correlate Native module

Name	Cross-Correlate Native
Graphical Icon	
Consumes messages of type	ScanDataCollection
Produces messages of type	Xml
Description	The functionality of this module is the same as for the “Cross-Correlate” module, with the exception that the core cross-correlation function is written in C++.

Table 10, Overview of the Cross-Correlate Native module.

Theory/background

See the theory section for the “Cross-Correlate module”.

Sometimes it can be a slow process doing vector multiplications in pure Java as each atomic operation will have to be converted between the virtual machine and the hardware running it. Implementing in C, however, has some advantages, as compiled code can be optimized, by the compiler, in a CPU specific way. In Java, it's possible to integrate with native code through the Java Native Interface (JNI).

Requirements

See the requirements section for the “Cross-Correlate module”.

The inner workings of the cross-correlation algorithm, implemented in this module, must be based on some C/C++ implementation, performing significantly better than its pure Java counterpart.

Implementation

Most of the code for the native cross-correlating module is very similar to that of the pure Java version, with one exception:

The function, that calculates the cross-correlation has been replaced with one that makes a call to an external library, containing a high performance version of the actual cross-correlation method (see source listing 21)

```
public static int calcMaxValueIndexCrossCorrelation(
        double[] signal1, double[] signal2){
        // create and init the variable to return
        int result = 0;

        // get an instance object of the CrossCorrelateWrapper
        CrossCorrelateWrapper wrap = CrossCorrelateWrapper.getInstance();

        // get the result from calling the native function
        result = wrap.findMaxIndexCorrelation(signal2,signal1);

        return result;
}
```

Source 21, Java code calling the optimized native function via a wrapper class.

All native method calls are accessed through wrapper classes to decouple the native interface from the core Java module even further, mainly to make it easier to change the underlying native libraries (see figure 30). Source listing 22 shows the accompanying C++ code produced. The main things in the code to take note of, is the places where arrays are being moved between Java and C++, as well as the Blitz++ functions, allowing very simple calls to be made, in order to perform the correlation.

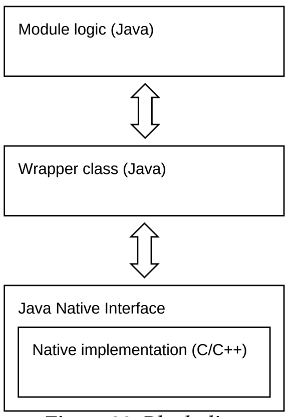


Figure 30, Block diagram of the wrapper functionality.

```
JNIEXPORT jint JNICALL
Java_com_dtu_oct_wrappers_CrossCorrelateWrapper_findMaxIndexCorrelation
 (JNIEnv *env, jobject obj, jdoubleArray signal1, jdoubleArray signal2)
 {
        // make the data and length of signal1 and signal2 accessible
        // (also locks the data, so Java won't modify it)
        jsize sig1_len = env->GetArrayLength(signal1);
        jdouble *sig1_data = env->GetDoubleArrayElements(signal1, 0);
        jsize sig2_len = env->GetArrayLength(signal2);
        jdouble *sig2_data = env->GetDoubleArrayElements(signal2, 0);

        // construct blitz++ arrays arround the signals
        blitz::Array<double,1>
                A(sig1_data, blitz::shape(sig1_len), blitz::neverDeleteData);
        blitz::Array<double,1>
                B(sig2_data, blitz::shape(sig2_len), blitz::neverDeleteData);

        // perform a convolution of signal2 reversed on signal1
        // corresponding to making a correlation of signal1 and signal2
        blitz::Array<double,1> C = blitz::convolve(A,B.reverse(0));

        // find the index of the maximum value in the correlated data
        // (the index is shifted to compensate for border effects of convolution)
        int maxIndex = blitz::maxIndex(C)[0] - (sig1_len-1);
        // unlock the data
        env->ReleaseDoubleArrayElements(signal1, sig1_data, 0);
        env->ReleaseDoubleArrayElements(signal2, sig2_data, 0);
        return maxIndex;
 }
```

Source 22, C++ (native) code performing a cross-correlation of two signals and returning the index of the maximum value found.

Usage/Results

For this module, the focus of testing will be on two things:

1. Does the module produce the same results as in the pure Java case and
2. How do the two modules compare in processing time.

Example 1: Repeating the alignment test

Design a small algorithm consisting of:

- 1 raw data phantom module,

- 1 native (C++) cross-correlation module
- 1 row translation module and
- 1 visualization module.

Connect the modules as shown in figure 31. Then select the "Sine" pattern on the details page for the data phantom, then click on the cross-correlation module and choose to align rows within images (see figure 32).

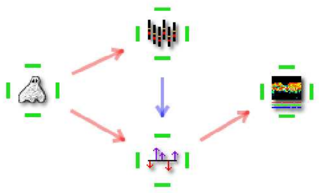


Figure 31, Aligning a phantom generated pattern, using the native (C++) cross-correlation module.

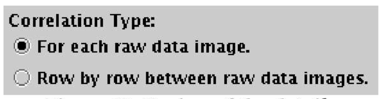

Correlation Type:
(● For each raw data image.)
○ Row by row between raw data images.

Figure 32, Region of the details panel of the Cross-Correlate Native module.

Use the visualization module to see the results, illustrated in figure 33, by pressing "Consume", then "Show". The resulting image is a straight line as in the example from using the pure Java cross-correlation module.

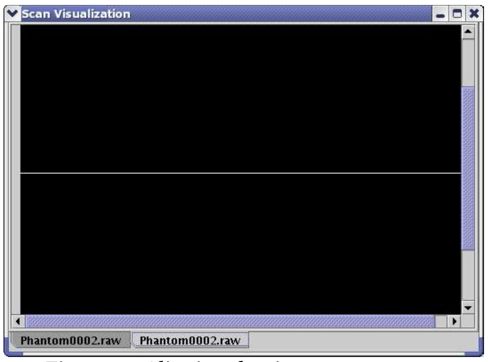


Figure 33, Aligning the sine wave pattern using the cross-correlation module results in a straight line.

Example 2: Comparing the two cross-correlation functions in a benchmark test

First, design an algorithm consisting of the following:

- 1 RAW data file import module,
- 1 Cross-correlation module and
- 1 Cross-correlation Native module.

Connect the modules as shown in figure 34.

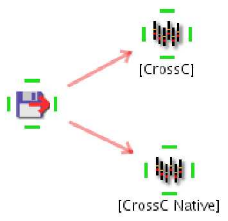


Figure 34, Diagram designed for benchmark.

Now, two different test cases are constructed:

- Case 1: 11 images from an OCT2 system being aligned by cross-correlation. Images dimensions are 100x500.
- Case 2: 5 images from an OCT3 system being aligned by cross-correlation. Image dimensions are 1024x512.

Duration in seconds	***Java version***	***C++ version***
Case 1	5.54	2.08
Case 2	51.27	17.1

Table 11, Benchmark results, comparing the C++ and Java implementations of the Cross-Correlation function.

The results, listed in table 11, clearly show a significant performance increase when using the C++ version.

3.4 Duplicate Data module

Name	DuplicateData
Graphical Icon	
Consumes messages of type	ScanDataCollection
Produces messages of type	ScanDataCollection
Description	Duplicates the incoming data for each instance of outgoing channels. Used to prevent data to be corrupted by parallel modifications in different paths.

Table 12, Overview of the Duplicate Data module.

Theory/background

To limit the amount of memory and CPU power consumed by the system while processing, the data contained in a ScanDataCollection message is not copied between modules. Instead, a reference to the same object is passed through from module to module. Although being very useful when trying to limit the amount of used resources, problems might occur when the same image data take more than one path through the constructed setup, where modifications done to the data in one path will affect the data in all other areas of the system. To work around this potential problem, a special module is needed to duplicate the data.

Requirements

The module should consume one ScanDataCollection message and produce copies of this to all connected consumers when requested.

Implementation

When data is consumed in the DuplicateData module, the reference to the data object is just stored like in most other modules. The main difference is in the function returning produced (see source listing 23).

```
public IScanDataCollectionMessage getScanDataCollectionMessage() {

        ScanDataCollection dataCol = null;

        // if there is data present to copy...
        if( null != m_dataCollection ){
                dataCol = new ScanDataCollection();

                // make a copy of the internal data collection
                Iterator it = m_dataCollection.getIterator();
                while(it.hasNext()){
                        ScanData data = (ScanData)it.next();
                        ScanData dataCopy = new ScanData(data);
                        dataCol.addData(dataCopy);
                }
        }

        // wrap the data copy in a message
        ScanDataCollectionMessage message = new
                ScanDataCollectionMessage(dataCol);

        return message;
}
```

Source 23, When the data duplication module is asked to deliver a ScanDataCollection message by the framework, a copy is returned instead, keeping subsequent modules from modifying the data contained.

Normally, the internally produced data is just returned as a reference pointer to the data object, wrapped in a ScanDataCollectionMessage to be sent out on a data channel. The data duplication module behaves differently by making a copy of the contained data to be returned instead. This way, all data consumers connected to the DuplicateData module will get a unique copy of the data, arriving at the incoming channel of the data duplication module.

Usage/Results

To show the effect of the data duplication module, and why it's needed, two examples are made. The first example shows how data can get corrupted by not using an the data duplication module, followed by an example of how to make it right.

Example 1: Results of NOT using data duplication

Design a small algorithm consisting of:

- 1 data phantom module,
- 1 filter convolution module and
- 1 visualization module to compare results.

Connect the modules as shown in figure 35.

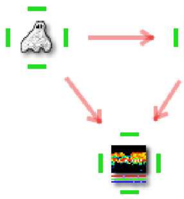


Figure 35, Without data duplication.

Select the “Checker Board” pattern for the phantom and the “Edge (width 5)” filter for the FilterConvolution module, and try to see the results by pressing “Consume” and “Show” in the visualization details panel.

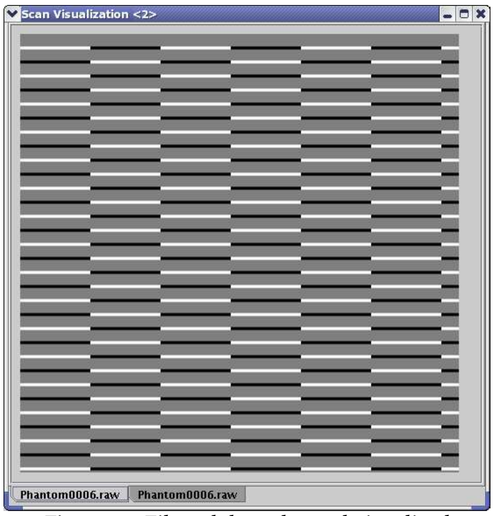


Figure 36, Filtered data channel visualized.

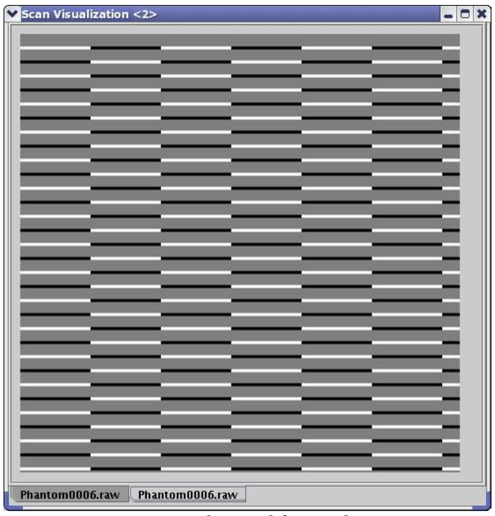


Figure 37, Data channel from phantom visualized.

As clearly seen by the results in figures 36 and 37, the two images are identical. That's because the underlying data for the two images is actually the same object! The next example shows how this effect can be eliminated quite easily.

Example 2: Doing it right

Now, try to cut away the channels coming from the phantom and include a data duplication module as shown in figure 38. Including the data duplication module, ensures that different paths of the data will contain unique copies of the incoming data chunks, hereby preventing the modifications on one path to affect any other area.

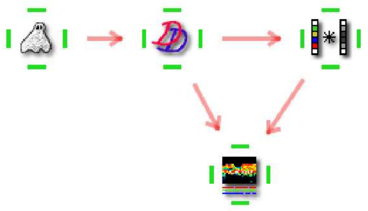


Figure 38, With data duplication.

Now, select the data phantom and press "Produce" to make a fresh image to work with, then select the visualization module and press “Consume” to start the processing chain. When done, press “Show” to see the result.

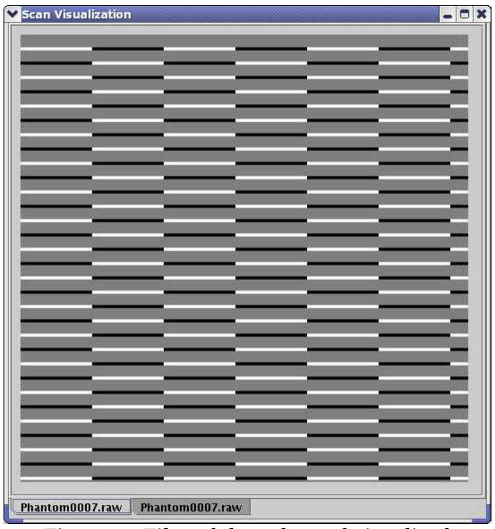


Figure 39, Filtered data channel visualized.

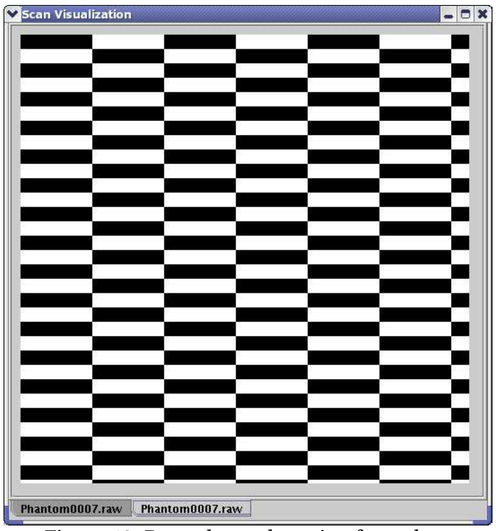


Figure 40, Data channel coming from the phantom visualized.

The result, seen in figure 40, clearly shows how the original image data has not been affected by the filter applied on the data, illustrated in figure 39.

3.5 Filter Convolution module

Name	FilterConvolution
Graphical Icon	▮✱▮
Consumes messages of type	ScanDataCollection
Produces messages of type	ScanDataCollection
Description	Applies a selectable filter to incoming data by convolving kernels with data rows.

Table 13, Overview of the Filter Convolution module.

Theory/background

In image processing, convolving images with a small signal, or kernel, can be used in preprocessing to enhance edges, low/hight frequencies, suppress certain structures, etc., using a small neighborhood of a pixel in an input image to produce a new brightness value in the output image [Sonka93, p. 67].

As we are working with 1-dimensional rows, corresponding to A-scans in the acquired OCT image, the kernels used are also 1-dimensional.

In this module, the filter kernel will be used directly as a mask, where it should not be reversed prior to processing, as otherwise traditionally done, when convolving two signals.

Requirements

This module should be able to make a convolution of the contents of an incoming ScanDataCollection with a selectable filter kernel. The filters should be applied in 1D for incoming A-scans. Required filters include:

- A “mean” filter, constructed as a flat filter ($y = 1$).
- A “gradient” or “edge” filter, designed as a straight line ($y = x$).

Both should be of a user selectable width.

Implementation

The filter mechanism is implemented as a light weight solution supporting only a few different static filter kernels, mainly to make the class easy to extend by limiting the amount of complexity in the code.

Mean and a gradient filters with kernel widths 3 and 5 are implemented as shown in source listing 24. More filters can easily be added this way, although a generic filtering module would be desirable sometime in the future, where the filter kernel would be dictated by an external XML source, enabling the user to produce filters from e.g. MatLab and import them into the system.

```
...
// Mean width 3
m_kernelVector.addElement( new double[]{0.3333,0.3333,0.3333});
m_kernelTextVector.addElement("Mean (width 3)");

// Mean width 5
m_kernelVector.addElement( new double[]{0.2,0.2,0.2,0.2,0.2});
m_kernelTextVector.addElement("Mean (width 5)");

// Gradient width 3
m_kernelVector.addElement( new double[]{-1.0,0.0,1.0});
m_kernelTextVector.addElement("Gradient (width 3)");

// Gradient width 5
m_kernelVector.addElement( new double[]{-2.0,-1.0,0.0,1.0,2.0});
m_kernelTextVector.addElement("Gradient (width 5)");
...
```

Source 24, Filter kernel definitions in the FilterConvolution module.

Usage/Results

In this example, we will try to apply two different filters to the same phantom data pattern.

Design an algorithm, containing the following modules:

- 1 data phantom module,
- 1 data duplication module,
- 2 filter convolution modules and
- 1 visualization module to view results.

Connect the modules as shown in figure 41, then select a sine wave for the phantom pattern (see figure 42), and do a quick visualization (see figure 43).

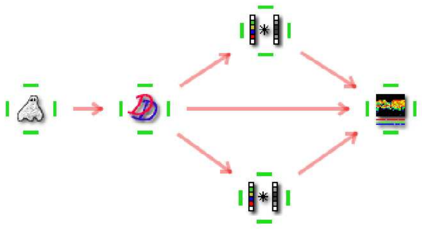


Figure 41, Algorithm showing phantom data being convolved with two different filters in parallel. Both results are visualized.

Phantom type: Sine

Figure 42, Selecting the "Sine" pattern in the module details panel for the raw data phantom.

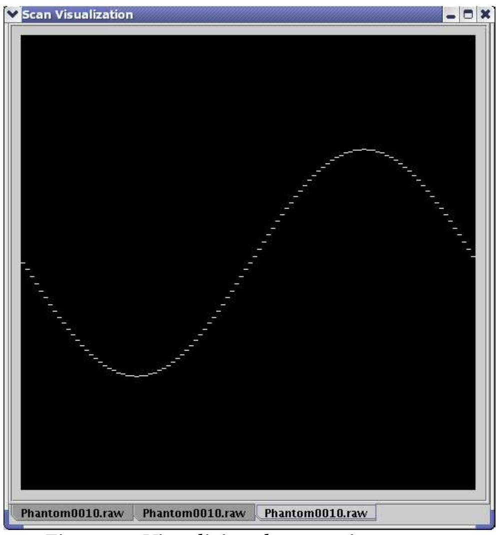


Figure 43, Visualizing the pure sine wave pattern, generated by the data phantom.

Now, set one of the filter convolution modules to “Mean (width 5)” and the other to “Edge (width 5)”, then select the visualization module, press “Consume”, and then “Show” to see the results. Figure 44 shows the image with an applied mean filter and figure 45 shows the same image with an applied gradient filter.

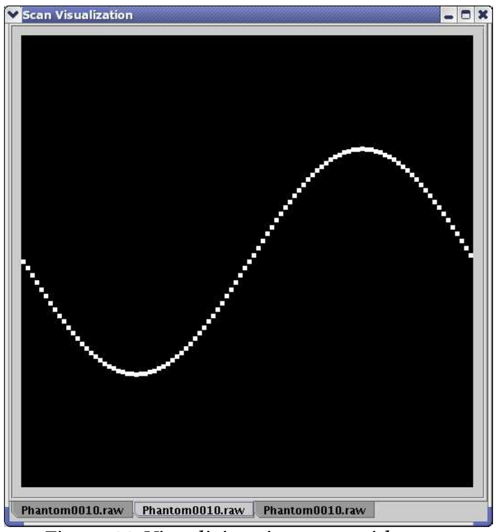


Figure 44, Visualizing sine wave with an applied mean filter.

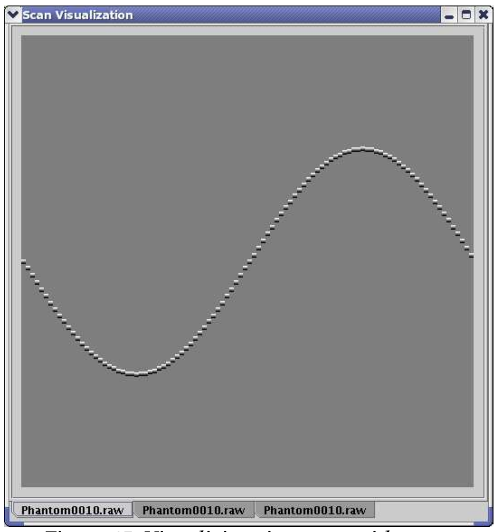


Figure 45, Visualizing sine wave with an applied edge detection filter.

3.6 Histogram Cutoff module

Name	HistogramCutoff
Graphical Icon	📊
Consumes messages of type	ScanDataCollection
Produces messages of type	ScanDataCollection
Description	Enables the user to cut away pixel brightness values outside a selectable range. A visualization of the histogram is provided to help with the selection.

Table 14, Overview of the Histogram Cutoff module.

Theory/background

Often, the static noise in an acquired image will have pixel values outside the range of where the actual signal data is present (see figure 46). If the noise can be identified to exist under some given threshold value, it should be possible to eliminate it (to some extent) by assigning all pixels with a value under it to that threshold value. This way, the threshold value will become the lowest value, thereby appearing as a “noise free” background [Teuber89, p. 158].

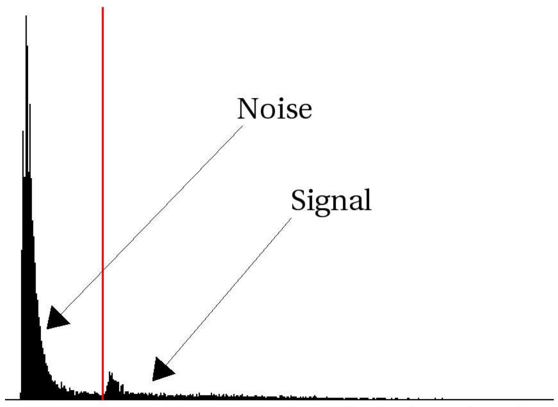


Figure 46, Histogram of image data, where the noise has been identified as the hill left of the red threshold separator.

Requirements

The program logic must be able to limit the data values for a ScanDataCollection by limiting individual A-scan data values to a specified range. It would also be desirable if the GUI provided a way of displaying the histogram with

Implementation

The histograms of the incoming data are calculated and placed in a HashMap relating to the name of the incoming image in the ScanData collection (see source listing 25). Histograms are contained in object instances of ScanDataHistogram, a class designed to contain and modify histogram data.

When the framework receives a “Produce” command, aimed at this module, a window in the brightness levels is “cut out” (see source listing 26), limiting lower and upper levels of the data according to values selected by the user:

```
private HashMap m_mapOfHistograms = new HashMap();

private void makeHistograms(){

        m_mapOfHistograms  = new HashMap();

        Iterator it = m_dataCollection.getIterator();

        System.out.print("Making histograms...");
        while(it.hasNext()){
                ScanData data = (ScanData)it.next();

                try {
                        String name = ((String)(data.getProperty("filename")));

                        m_mapOfHistograms.put(name,
                                new ScanDataHistogram(data));

                } catch (Exception e){
                        // TODO handle
                        System.out.println(""+e);
                }

        }

        System.out.println(" [done]");
}
```

Source 25, Upon data reception (where data is consumed on a ScanData collection channel), histogram data is produced for each image in the collection.

```
private static void cutWindow(
        double minValue, double maxValue, double[] signal){

        for(int i=0;i<signal.length;i++){
                if( signal[i] < minValue ){
                        signal[i] = minValue;
                } else if( signal[i] > maxValue ){
                        signal[i] = maxValue;
                }
        }
}
```

Source 26, The cutWindow function implemented to limit the lower and upper bounds of the brightness levels in a signal (raw data).

Usage/Results

Design an algorithm, consisting of the following modules:

- 1 raw data import module,
- 1 visualization module,
- 1 histogram cutoff component and
- 1 data duplicator to prevent mixing of filtered and unfiltered data.

Connect the modules as shown in figure 47. Then use the RAW data import module to import some real data. Select the Histogram Cutoff module and press "Consume" followed by "Show". The histogram for the loaded data should look similar to the one shown in figure 48.

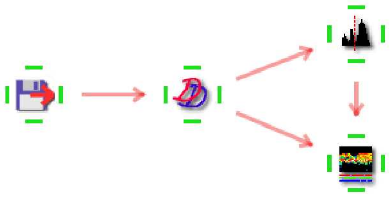


Figure 47, Visualization of data before and after applied cutoff threshold.

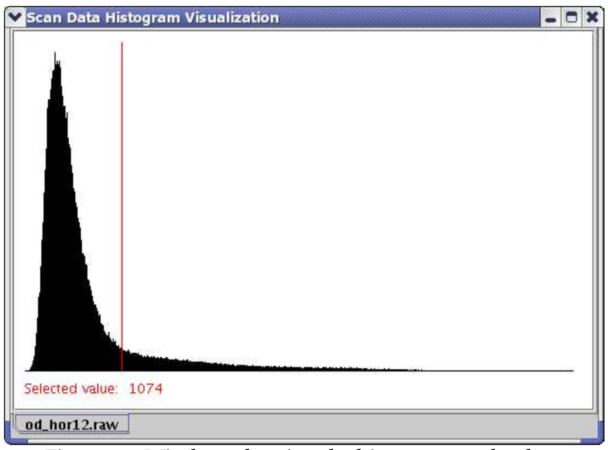


Figure 48, Window, showing the histogram and red selection line.

Using the mouse to point at the histogram, find a value separating the huge pile to the left from the rest. Read the value under the graph, this will be our chosen cutoff value.

Now, select the visualization module, press “Consume” and “Show” to compare the results.

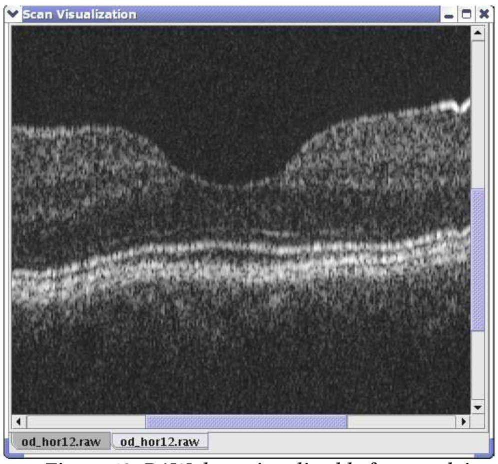


Figure 49, RAW data visualized before applying the histogram cutoff.

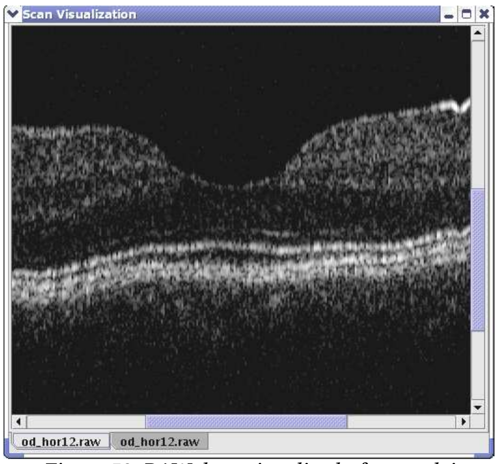


Figure 50, RAW data visualized after applying the histogram cutoff. Notice how much of the background noise has disappeared.

As seen in figure 50, much of the background noise has been eliminated from the image in figure 49.

3.7 Median Filter module

Name	MedianFilter
Graphical Icon	
Consumes messages of type	ScanDataCollection
Produces messages of type	ScanDataCollection
Description	Applies a median filter with a selectable width to incoming data rows.

Table 15, Overview of the Median Filter module.

Theory/background

The main purpose of a median filter is to remove impulse noise from images. The filter works by replacing a pixel value with the median of it's neighbors [Niblack86, p.78]. The median value is found by placing all values, within a certain distance from the pixel in question, in a sorted 1D array, followed by a selection of the middle value in the array. The principle of the median filter is illustrated in figure 51.

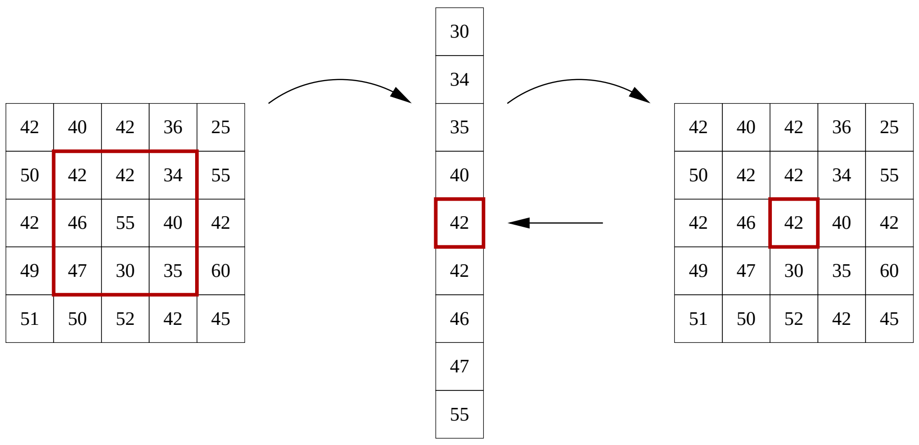


Figure 51, Principle of the median filtering mechanism illustrated. NOTE: This example shows a 2D neighborhood matrix for input for illustrative. purposes. When working with rows in the raw data, we will be using a 1D neighborhood function for selection.

In the case of this module, the median filter will be applied to images composed of multiple acquired - and possibly jumpy - A-scans next to each other. A one dimensional neighborhood selection will therefore be selected, for the initial array, as two dimensional selection would only make sense after the A-scans have been correctly aligned.

The main advantage of median filters over generic smoothing filters (see section on the Filter Convolution module), is that it eliminates impulsive noise quite well , while not blurring edges very much [Sonka93, p.74].

Requirements

The module must provide means to apply a median filter of varying width on incoming ScanDataCollections. The result is a ScanDataCollection containing all images (ScanData objects) affected by the median filter.

Implementation

The heart of the median filtering algorithm implementation, is relying on two very optimized functions built into the Java core (see source listing 27):

- System.arrayCopy(), doing low level memory duplication between two arrays of the same type, given offset and length.
- Arrays.sort(), implementing the very effective Quicksort [Cormen96, p.153], sorting the specified array into ascending numerical order.

```
…
for( int i=0; i<signal.length-width; i++ ){
        System.arraycopy(signal,i,tmpArray,0,tmpArray.length);
        Arrays.sort(tmpArray);
        result[i+halfwidth] = tmpArray[halfwidth];
}
…
```

Source 27, Using the built in Java functionality to sort arrays for median selection.

Usage/Results

Design an algorithm like the following, consisting of:

- 1 data phantom,
- 1 visualization module,
- 1 median filter component and
- 1 data duplicator to prevent mixing of filtered and unfiltered data.

Connect the modules as shown in figure 52 and select the median filter module.

Set the width to 5 (see figure 53), then visualize the results by clicking on the visualization module followed by a click on “Consume” and “Show”.

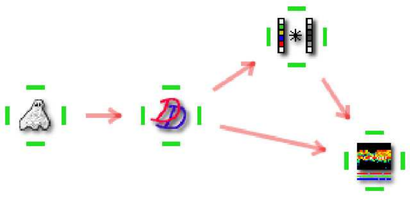
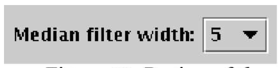


Figure 52Visualizing median filtered and raw data from a phantom.



Median filter width: 5

Figure 53, Region of the module details panel of the Median Filter module.

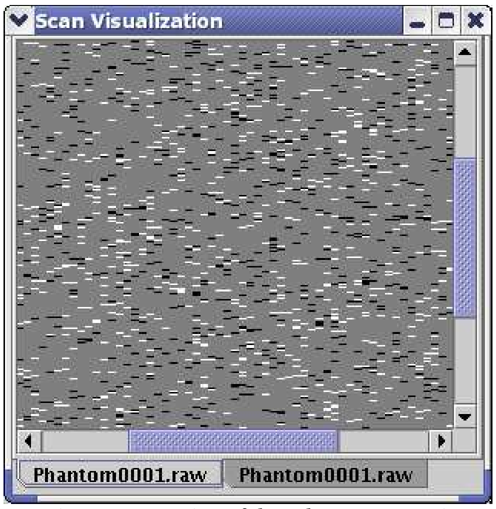


Figure 54, A region of the salt & pepper noise generated by the data phantom.

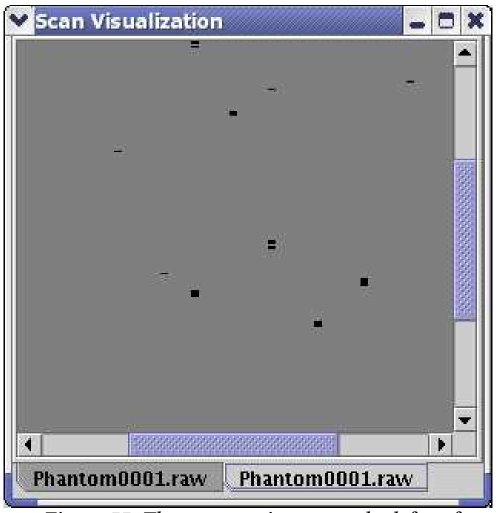


Figure 55, The same region as on the left - after applying the median filter.

Comparing the two images in figures 54 and 55, it is clear that the median filter has eliminated a lot of the salt & pepper noise.

3.8 Normalize module

Name	Normalize
Graphical Icon	
Consumes messages of type	ScanDataCollection
Produces messages of type	ScanDataCollection
Description	Scales pixel brightness values in the incoming data to fit within a specified range.

Table 16, Overview of the Normalize module.

Theory/background

Normalizing data can be useful before an addition or when comparing two sets of data, as the data might have values in totally different scales/ranges while representing the same properties. The standard normalization of data would transform all pixel values to fit between zero and one, e.g. the currently lowest valued pixels would be set to zero, the highest values set one, etc..

In this system, the mechanism will be extended to also include scaling and translation. The principle is illustrated in figure 56.

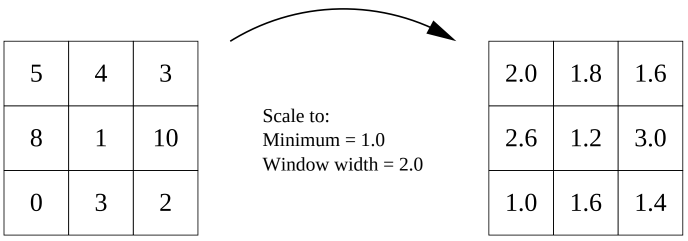


Figure 56, Scaling an array of pixels (brightness values) to fit within a window from 1.0 to 3.0.

Requirements

This module must provide the user with a way of specifying a new minimum and a data range value (a window) for the incoming ScanDataCollection. All ScanData elements (images) in the outgoing message should contain transformed version of the incoming according to these parameters.

Implementation

The core part of the scaling logic is in the normalize function illustrated in source listing 28, where all values are scaled to fit within the range defined by “newMax” and “newMin”.

```
...
private static void normalize( double[] signal,
        double oldMin, double oldMax,
        double newMin, double newMax){

        // calculate the old and the new brightness window size
        double oldDiff = oldMax-oldMin;
        double newDiff = newMax-newMin;

        // if the old image is not flat ( max = min )
        if(oldDiff > 0.0){
                for( int i=0; i<signal.length; i++ ){
                        // scale the pixel value
                        signal[i] = newMin + (((signal[i]-oldMin)/oldDiff)*newDiff);
                }
        }
}
...
```

Source 28, Scaling the ScanData values to fit within newMin and newMax.

Usage/Results

In this example, a composite image is constructed, consisting of a sine wave pattern with some background noise. The noise needs to be of a lower brightness level than the sine wave.

Design an algorithm, consisting of the following modules:

- 2 data phantoms,
- 1 visualization module,
- 2 normalization modules and

- 1 scan data collection module.

Connect the modules as shown in figure 57 and configure the modules according to the accompanying labels shown. Then select the visualization module, press “Consume” and “Show” to view the results seen in figures 58 and 59.

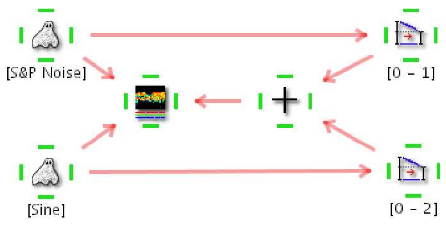


Figure 57, Algorithm to construct a composite pattern of a sine wave with noise in the background. Labels indicate module settings.

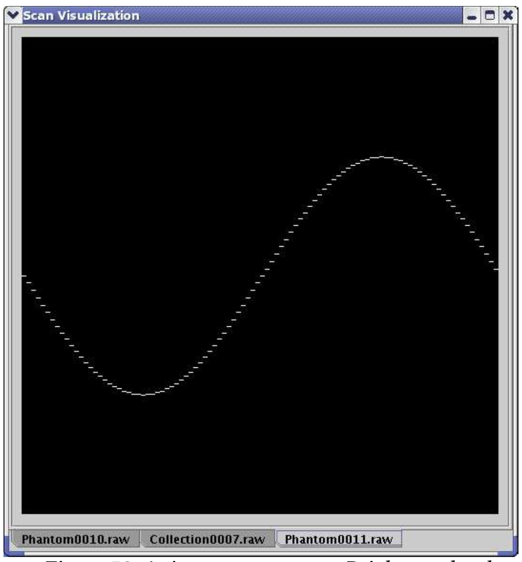


Figure 58, A sine wave pattern. Brightness levels are between 0 and 1.

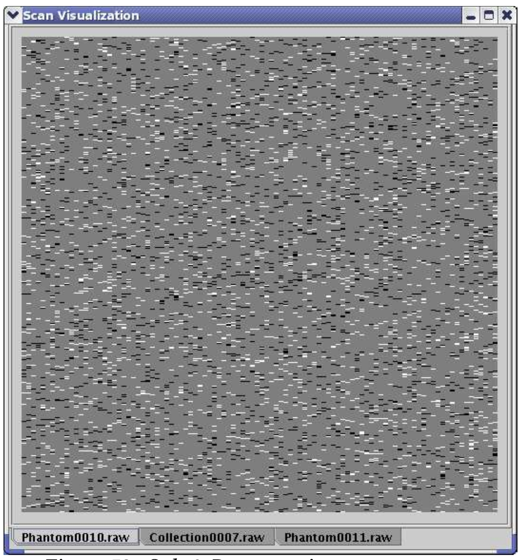


Figure 59, Salt & Pepper noise pattern. Brightness levels are between -1 and 1.

The brightness value of the sine wave pattern is scaled to fit between 0 and 2, while the noise is scaled to fit between 0 and 1. This makes the sine wave relatively more bright when combining the two patterns (see figure 60).

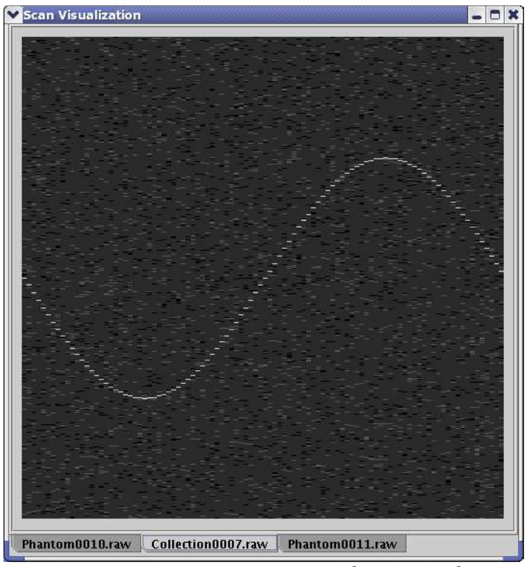


Figure 60, Sine wave pattern with noise in the background. Scaling of brightness levels prior to addition, makes the sine wave brighter than the background noise.

3.9 RAW Data File Import module

Name	RawDataFileImport
Graphical Icon	💾
Consumes messages of type	-
Produces messages of type	ScanDataCollection
Description	Imports RAW data generated and exported from Zeiss retinal OCT scanners.

Table 17, Overview of the RAW Data File Import module.

Theory/background

RAW data files exported from the ZEISS OCT systems contain the actual data values acquired throughout the scanning. At our disposal, we have had two different versions of the OCT retina scanner:

1. The Humphrey OCT scanner (Scanner A) capable of scanning 100x500 images.
2. A STRATUSOCT scanner (Scanner B) capable of scanning 512x1024 images.

After some experimentation, the following was discovered:

Data is stored as int16 values (Intel ordering) in files given the extension *.RAW when exporting data from the systems.

A-scans are stored as concatenated raw data chunks. For scanner A, each column is 500 rows x 2 bytes/pixel = 1000 bytes long. For scanner B it's 1024 rows x 2 bytes/pixel = 2048 bytes long.

Requirements

The internal image data format in the Java OCT Imaging System contains a collection of double arrays (multiple images) making it possible to perform image operations involving more than one image. A way of loading one or more raw data images into one internal ScanDataCollection is therefore needed.

Implementation

The RAW data images loaded from disk is converted to an internal double array and wrapped in a data container, ScanData (see source listing 29). The following properties are set in the data object:

- Rows: The number of rows in the image.
- Cols: The number of columns in the image.
- Maxvalue: The maximum pixel brightness value found.
- Minvalue: The minimum pixel value found.
- Filename: Name of the RAW data file.
- Ratio: A possible aspect ratio, used when visualizing the image.

```
...
if( 100000 == file.length()){ // 500*100*2 bytes
        int rows = 100;
        int cols = 500;

        byte[] rawbuffer = new byte[2*rows*cols];
        FileInputStream fp = new FileInputStream(file);
        fp.read(rawbuffer);
        fp.close();


        double[] rawData = new double[rows*cols];

        double minValue=999999;
        double maxValue=-999999;
        for(int i=0;i<rawData.length;i++){
                // convert the 2 bytes to a double value
                rawData[i] = (double)( ((int)rawbuffer[(i<<1)]&0xff) +
                        ( ((int)rawbuffer[(i<<1)+1]&0xff) <<8));
                // update the max/min values found
                if(rawData[i]>maxValue)maxValue=rawData[i];
                if(rawData[i]<minValue)minValue=rawData[i];
        }
        // Assign the data
        data = new ScanData(rawData);
        // Set data properties
        data.addProperty("rows",new Integer(rows));
        data.addProperty("cols",new Integer(cols));
        data.addProperty("maxvalue",new Double(maxValue));
        data.addProperty("minvalue",new Double(minValue));
        data.addProperty("filename",file.getName());
        data.addProperty("ratio", new Double(5.0));
}
...
```

Source 29, Loading and converting an image, exported from the Hymphrey OCT scanner.

Usage/Results

The two different RAW data files supported will be imported and visualized in this example. Design an algorithm like the following, consisting of:

- 1 RAW data file import module and
- 1 visualization module.

Connect the modules as shown in figure 61. In the file import module, load one RAW data file originating from the Zeiss Humphrey OCT scanner and one from the STRATUSOCTTM scanner. Select the visualization module, press "Consume" and "Show" to see the loaded images (see figures 62 and 63).

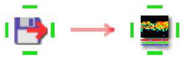


Figure 61,
Visualizing RAW data.

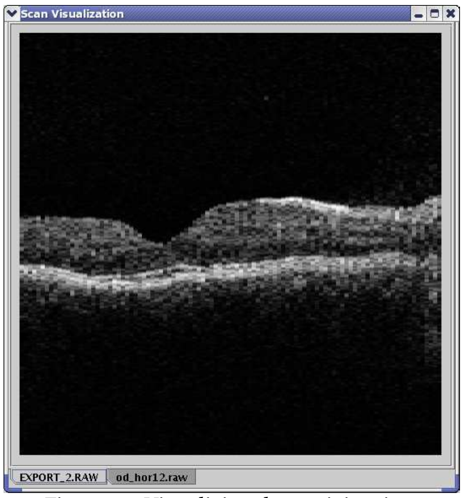


Figure 62, Visualizing data originating from the Zeiss Humphrey OCT retinal scanner.

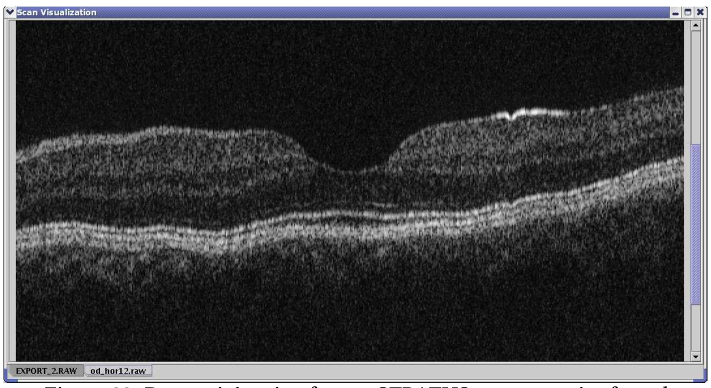


Figure 63, Data originating from a STRATUSoct system, is of much higher resolution.

3.10 RAW Data Phantom module

Name	RawDataPhantom
Graphical Icon	👻
Consumes messages of type	-
Produces messages of type	ScanDataCollection
Description	Produces a test pattern in a user selectable dimension. Patterns include “Checker Board”, “Salt & Pepper Noise” and “Sine” - new patterns are easily added.

Table 18, Overview of the RAW Data Phantom module.

Theory/background

When testing and calibrating measurement systems, especially in the world of medicine, it is a custom to use reference objects with well defined properties, covering all parameters measurable to the system, in a seem to be real life measurement[12]. For liquid measuring stations, like blood/gas measuring devices, these known as quality control fluids. Objects to be used with Computed Axial Tomography (CAT) or Nuclear Magnetic Resonance (NMR) scanners are volumetric objects, with well defined physical properties that lies in ranges close to those of humans [Cho93, p. 558-563]. These objects are called phantoms.

In a similar sense, phantoms in this system will be objects made to work as reference elements, useful for verifying and testing the capabilities of image processing modules, in the form of different reference patterns.

12 “What is a phantom?”, CIRS, http://www.cirsinc.com/overview.html

Requirements

The Phantom module should provide a way to produce test data using pluggable pattern generators. The following patterns are recommended for testing of other existing modules:

- Checker Board
- Salt & Pepper Noise
- Sine Wave

The patterns should be producible for all possible image sizes.

Implementation

The different selectable phantom pattern generators are implemented as separate classes (see figure 64), each containing logic to produce a specific pattern, given a predefined 'shell' in the form of a ScanData object. The ScanData object should be loaded with image 'rows' and 'cols', as these values will provide the base for the pattern generating algorithm to determine the size of the produced pattern.

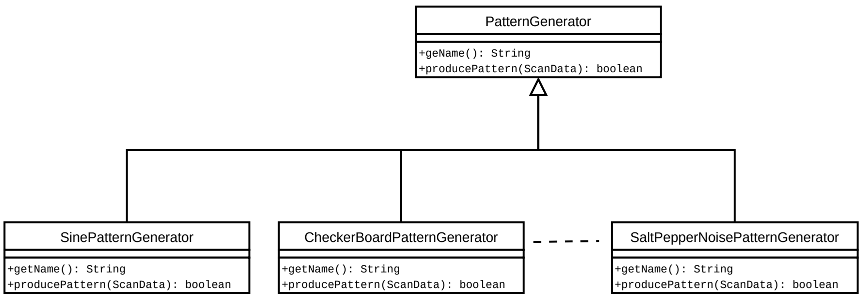

PatternGenerator
+geName(): String
+producePattern(ScanData): boolean
SinePatternGenerator
+getName(): String
+producePattern(ScanData): boolean
CheckerBoardPatternGenerator
+getName(): String
+producePattern(ScanData): boolean
SaltPepperNoisePatternGenerator
+getName(): String
+producePattern(ScanData): boolean

Figure 64, UML class diagram of the pattern generation implementation.

Usage/Results

Build an algorithm diagram consisting of the following modules:

- 3 raw data phantom modules and
- 1 data visualization module

Connect them as shown in figure 65, and select the phantom patterns as indicated by labels on the figure. Then select the visualization module, press “Consume”, then “Show” to see the three patterns illustrated in figures 66 to 68.

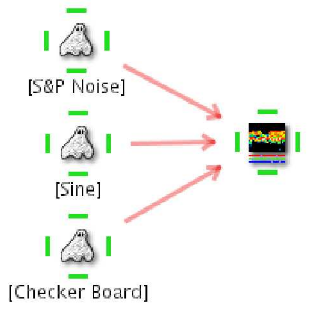


Figure 65, Visualization of three different raw data phantom patterns. Labels indicate phantom pattern settings.

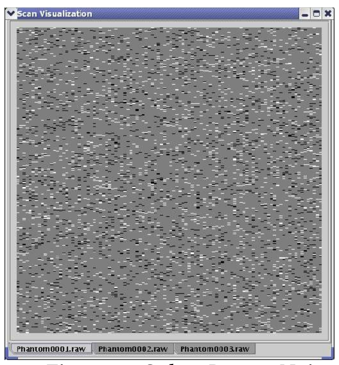


Figure 66, Salt & Pepper Noise

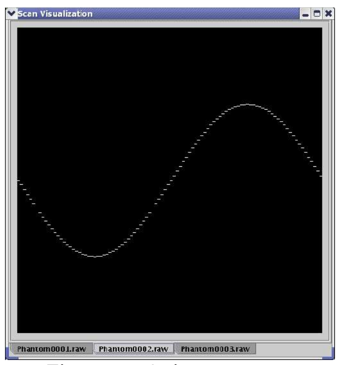


Figure 67, A sine wave pattern.

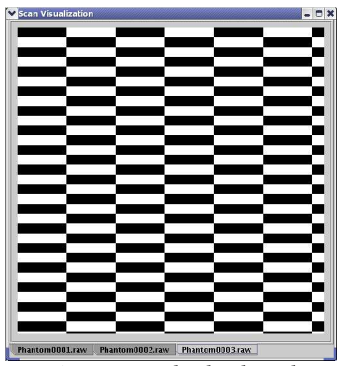


Figure 68, A checker board pattern.

3.11 Show XML Tree module

Name	ShowXmlTree
Graphical Icon	XML
Consumes messages of type	Xml
Produces messages of type	-
Description	Produces a 'pretty print' view of incoming XML messages.

Table 19, Overview of the Show XML Tree module.

Theory/background

Wherever XML is used in the system, it might be desirable to be able to visualize the data on the fly. The content might be originating from a file import, a cross-correlation or some other module, producing XML.

Requirements

This module must at least be able to visualize the contents of an XmlMessage in a multi-line text window. The logic should accept one incoming XmlMessage channel.

Implementation

The XML visualization will format the incoming message for easier readability, using the XmlWriteContext class (see source listing 30). One of the main advantages of using XML is that it is human readable and in most cases, easily editable without the use for anything but a simple text editor. One, however, should make sure that auto generated XML is converted to a pretty printed form before delivery (e.g. in a file, on screen, etc.).

```
…
public static void prettyPrint(Document xmlDoc, Writer out){
        try {
                XmlDocument doc = (XmlDocument)xmlDoc;

                XmlWriteContext xmlWriteContext =
                        doc.createWriteContext(out, 0);

                doc.writeXml(xmlWriteContext);

                out.flush();
        } catch (Exception e){
        }
}
…
```

Source 30, Pretty printing XML is done by a globally available function defined in the framework.

Usage/Results

Build a diagram (algorithm) consisting of the following modules:

- 1 raw data phantom module,
- 1 cross-correlate module and
- 1 XML visualization module

Connect them as shown in figure 69. The ShowXMLTree module will display the contents of an incoming XmlMessage in a multi-line text field in the details panel, illustrated in figure 70.

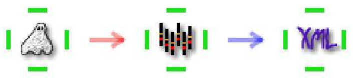
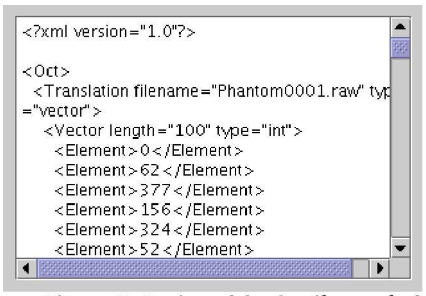


Figure 69, Minimalistic algorithm, constructed to show the XML generated by the Cross-Correlation module.



```
<?xml version="1.0"?>

<Oct>
  <Translation filename="Phantom0001.raw" typ
="vector">
    <Vector length="100" type="int">
      <Element>0</Element>
      <Element>62</Element>
      <Element>377</Element>
      <Element>156</Element>
      <Element>324</Element>
      <Element>52</Element>
```

Figure 70, Region of the details panel of the Show XML Tree module, containing the output vector from a Cross-Correlation.

3.12 Translate Rows module

Name	TranslateRows
Graphical Icon	
Consumes messages of type	Xml + ScanDataCollection
Produces messages of type	ScanDataCollection
Description	Translates rows in an incoming RAW data message according to related vectors in the incoming XML message.

Table 20, Overview of the Translate Rows module.

Theory/background

This module should translate rows in a scan data collection according to an incoming XML vector.

NOTE: As stated earlier in this report, the normal image representation, done by the VisualizeScans module, will display columns as rows and vice versa. This has nothing to do with the underlying model representation, where acquired A-scans are stored as rows in the RAW data.

Requirements

A module that combines a ScanDataCollection message and an XML message in such a way that rows in the incoming images (in the ScanDataCollection) are translated (shifted) according to vectors in the XML message.

Implementation

The row translation uses an externally generated XML vector to shift the offset of A-scan rows in a ScanData collection of images.

For example, a vector generated by running a phantom sine wave pattern through the Cross-Correlate module, can be seen in source listing 31.

```xml
<?xml version="1.0"?>

<Oct>
  <Translation filename="Phantom0011.raw"
type="vector">
    <Vector length="100" type="int">
      <Element>0</Element>
      <Element>7</Element>
      <Element>15</Element>
      <Element>23</Element>
      <Element>31</Element>
      <Element>38</Element>
      <Element>46</Element>
      <Element>53</Element>
...
      <Element>409</Element>
      <Element>415</Element>
      <Element>421</Element>
      <Element>427</Element>
      <Element>434</Element>
      <Element>440</Element>
      <Element>447</Element>
      <Element>454</Element>
      <Element>462</Element>
      <Element>469</Element>
      <Element>477</Element>
      <Element>485</Element>
      <Element>493</Element>
    </Vector>
  </Translation>
</Oct>
```

Source 31, Example output XML vector (shortened to fit here). The vector is a "translation" vector relating to the image with the name, "Phantom0011.raw".

Usage/Results

See example from the cross-correlation module.

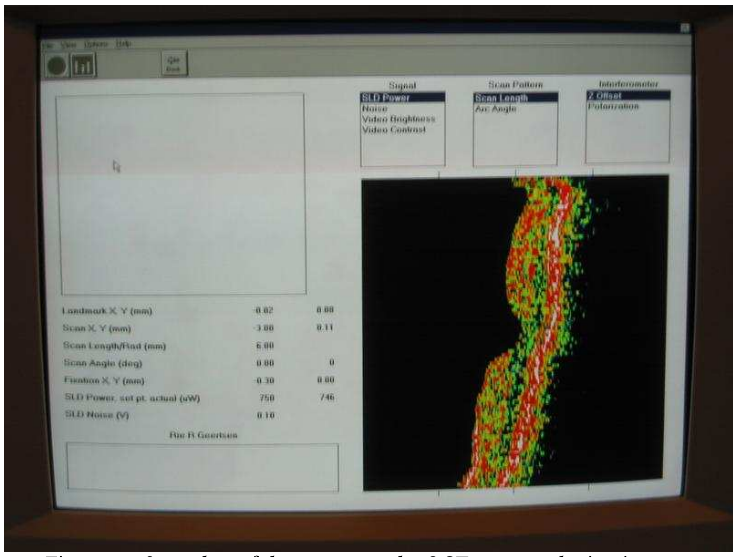
3.13 Visualize Scans (and Visualize Scans As Model) modules

Name	VisualizeScans (and VisualizeScansAsModel)
Graphical Icon	
Consumes messages of type	ScanDataCollection
Produces messages of type	-
Description	Visualizes RAW data from incoming channels as images (in a tabbed panel) in a separate window.

Table 21, Overview of the Visualize Scans module.

Theory/background

Data visualization is an essential part of this image processing system. In the current design, the only binary data routed through the system, consists of collections of acquired datasets, stored as 2-dimensional arrays of double precision values. Each dataset represents a series of acquired A-scans, originating from an OCT retina scanner. Figure 71 shows an image of the screen on the scanner during acquisition. The "live" scanning is visualized as a continuously updated set of pixel rows (A-scans) in the right half of the image.

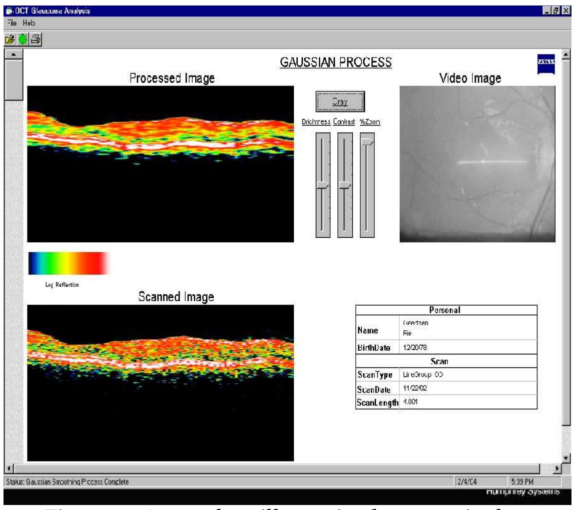


Landmark X Y (mm)	-0.02	0.00
Scan X Y (mm)	-3.00	0.11
Scan Length/Rad (mm)	6.00	
Scan Angle (deg)	0.00	0
Fixation X Y (mm)	-0.30	0.00
SLD Power, set pt. actual (uW)	750	746
SLD Noise (V)	0.10	

Rie R Geertsen

Figure 71, Snapshot of the screen on the OCT scanner during image acquisition.

When representing the scans, the data will be mirrored along a diagonal in the image, going through (0,0), prior to display, as the underlying data structure contains A-scans, stored as rows, while software on the Carl Zeiss retinal OCT scanners displays them as columns, as seen in figure 72. The mirroring is done to make the view more “compatible” visually, with that of the scanner software.



Personal	
Name	Geertsen Rie
BirthDate	12/20/78
Scan	
ScanType	LineGroup OD
ScanDate	11/22/02
ScanLength	4.001

Status: Gaussian Smoothing Process Complete 2/4/04 5:39 PM

Figure 72, A snapshot, illustrating how acquired scans are mirrored in the diagonal, when shown for analysis, in the software provided by Carl Zeiss.

Requirements

The module must be able to display a gray level image of any kind of ScanDataCollection. The images should be flipped, so rows will become cols, and vice versa. This is due to the fact that rows in the RAW data files corresponds to A-scans in the acquired data. These scans are normally displayed vertical (as columns).

For use in special cases, where it is desired to visualize data as it's stored in the underlying model (rows being rows and columns being columns), a special version of the visualization module should be made, called “VisualizeScansAsModel”.

Implementation

As the screen will only display gray levels between 0 and 255, brightness values in the ScanData collection should be scaled to fit this dynamic range. The code to achieve this, is shown in source listing 32.

```
...
double redRange[] = new double[]{ minValue, maxValue-minValue };
double greenRange[] = new double[]{ minValue, maxValue-minValue };
double blueRange[] = new double[]{ minValue, maxValue-minValue };
// mirror the image so the upper right corner becomes the lower left
// reason: the scanned data has swapped rows/columns
// compared to normal images
m_ratio = 1.0/((Double)(data.getProperty("ratio"))).doubleValue();
int row=0,col=0,destPtr=0;
for(int i=0;i<rawData.length;i++){

        int red = (int)(((rawData[i]-redRange[0])*255.0)/redRange[1]);
        if(red < 0)red=0;
        if(red > 255)red=0;
        int blue = (int)(((rawData[i]-blueRange[0])*255.0)/blueRange[1]);
        if(blue < 0)blue=0;
        if(blue > 255)blue=0;
        int green = (int)(((rawData[i]-greenRange[0])*255.0)/greenRange[1]);
        if(green < 0)green=0;
        if(green > 255)green=0;

        tmpData[destPtr] = 0xff000000 + (red<<16) + (green<<8) + blue;
        destPtr+=rows;
        col++;
        if(col>=cols){
                col=0;
                row++;
                destPtr=row;
        }
}
...
```

Source 32, Scaling the brightness values to fit within the dynamic range of the computer screen. The code is prepared to support color overlay windows by separating red, green and blue channels.

Usage/Results

Build a diagram (algorithm) consisting of the following modules:

- 1 raw data file import module,
- 1 visualize scans module and
- 1 visualize scans as model module

Connect them as shown in figure 73 load a raw data file using the RAW Data File Import module. Pressing “Consume” and “Show” for both of the visualization modules, will result in the images shown below. One, where data is visualized after mirroring, using the normal VisualizeScans module (figure 74), and one, showing the raw data as-is, using the VisualizeScansAsModel module (figure 75), where rows in the underlying data are visualized as rows in the displayed image.

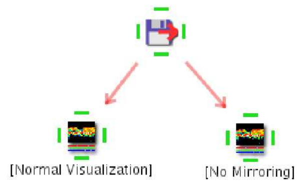


Figure 73, Diagram with two kinds of visualization: The normal one, and a modified version where the image is not mirrored.

Also, see one of the many other examples in this chapter, utilizing the scan visualization module.

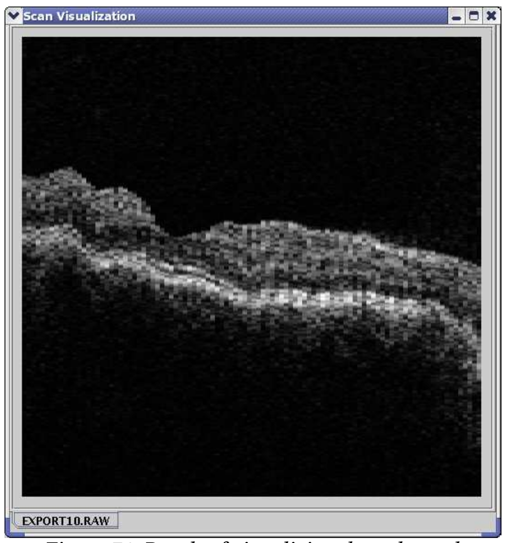


Figure 74, Result of visualizing data through the normal "VisualizeScans" module.

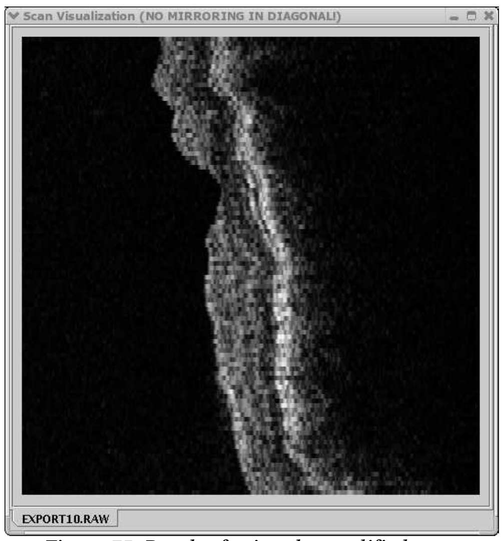


Figure 75, Result of using the modified "VisualizeScansAsModel" module.

3.14 XML File Import module

Name	XmlFileImport
Graphical Icon	
Consumes messages of type	-
Produces messages of type	Xml
Description	Imports XML from a file and wraps it in an XmlMessage.

Table 22, Overview of the Xml File Import module.

Theory/background

There is a need for the system to be able to import XML based data. This data could either be part of a previous processing (within the system), or from some external program. Providing this module would enable the user to use totally decoupled applications , e.g. MatLab, to produce XML data to be used as an input for operational components, such as the TranslateRows module.

Requirements

The module must provide an easy way to import XML data using a standard file chooser dialog. The filename should be saved as state information as part of the generic "save..." functionality for the system.

Implementation

Java has an excellent set of APIs for handling XML built in. When importing XML from a file, in three steps, it's possible to get from the flat file on disk to a complete document object model (DOM).

1. Get an instance of a document builder factory.
2. Use the retrieved factory to get a document builder.
3. Use the document builder to create a DOM, parsing a file containing valid XML.

The implemented Java code to handle can be seen in source listing 33.

```
public String loadXmlDocument(File file){
        String result = "(none)";

        //load the actual xml file
        try {
                DocumentBuilderFactory domFactory =
                        DocumentBuilderFactory.newInstance();

                DocumentBuilder domBuilder =
                        domFactory.newDocumentBuilder();

                m_document = domBuilder.parse(file);

                result = file.getName();
                System.out.println("'" + result + "' is well-formed.");
                setDataReady(true);
        } catch(Exception e) {
                System.err.println(e.toString());
        }
        return result;
}
```

Source 33, Loading an XML file, parsing it and storing the returned DOM in the member variable, m_document. If an error occurs, "(none)" is returned, indicating to the caller that no valid XML document was loaded.

Usage/Results

First, create a file, containing the following line of text:

"`<Hello><World>!</World></Hello>`".

Then build an algorithm diagram consisting of the following modules:

- 1 XML file import module and
- 1 XML visualization module.

Connect them as shown in figure 76 and open the file in the import module. Then select the visualization module and press "Consume" to see the result, illustrated in figure 77.

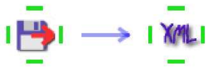
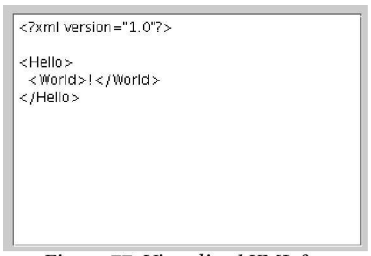


Figure 76, Algorithm to visualize an imported XML file.



```
<?xml version="1.0"?>

<Hello>
  <World>!</World>
</Hello>
```

Figure 77, Visualized XML from file.

Notice how the XML is "pretty printed", compared to the line in the file.

3.15 Summary

The modules listed in this chapter, form what could be considered a "core base" of functionality. However, enough modules are provided, to make the system useful in different real-life scenarios. Some of these will be covered in the following chapters, including a case, where a future user will test the algorithm design functionality of the system, combining modules in new ways.

4. Building algorithms

Now that a solid framework has been created, with a selection of different pluggable components, the next logical step is to actually get a feel of how the system would be used in practice.

In this chapter, two example algorithms are presented:

- "Enhancement by addition", constructed to enhance the image quality by adding a series of images being subsequent acquisitions of the same area in the retina and
- "Filtered phantoms", showing how combining phantom patterns, filters and pixel-wise addition, can produce somewhat realistic images for testing.

4.1 Enhancement by addition

Raw data acquired and exported from the Zeiss retina OCT scanners are generally filled with scatter noise and very jumpy. To deal with this, a method is presented here, that will take a series of images of the same data (subsequently acquired with a few seconds delay in between) and combine them to enhance the image quality by reducing the noise/signal ratio. The method is inspired by a paper on the subject, "Reducing speckle noise in retinal OCT images by aligning multiple B-scans" [Jørgensen04].

4.1.1 Theory

If we assume that each acquired pixel in the raw data follows this mathematical model:

$$g(x,y)=f(x,y)+e(x,y)$$

where g(x,y) is the sampled intensity, f(x,y) the true value and e(x,y) is assumed to be Gaussian noise (e.g. $e(x,y)\in N(0,\sigma^2)$), then by doing a pixel wise summation, we should be able to improve the signal/noise ratio by doing a pixel wise addition of acquired scans representing the same data [Carstensen97, p.40]. Adding n images and dividing by n we get:

$$\frac{1}{n}\sum_{n} g(x,y) = f(x,y) + \frac{1}{n}\sum_{n} e(x,y)$$

where $\frac{1}{n}\sum_{n} e(x,y) \in N(0,\frac{\sigma^2}{n})$, showing the noise being reduced with a factor of n.

Because of the high noise rate in raw data scans, it would be interesting to examine the possibility of reducing it in some way by exploiting the possibility of subsequently acquiring image data from the same place in the retina, and then do a pixel-wise addition of the data.

4.1.2 Construction

An algorithm is constructed, reflecting the overall concept of the idea described above. The first step is to drag a RawDataFileImport module to the testbed area, capable of importing the series of images required for the addition.

As multiple images are being added pixel-wise, it is important that the images in the set will be aligned with one another prior to addition, as the data will be present in different places due to small eye movements during data acquisition. For that, a Cross-Correlation module, a TranslateRows module followed by a CollectScans module is needed.

The one image resulting from the pixel-wise addition, should be less noisy, due to the fact explained above (in the Theory section), but because we have only aligned the images to match each other, and not caring about aligning subsequent rows in each image, the result would still appear jumpy. To correct this, yet another Cross-Correlation module, and its TranslateRows counterpart, is required.

Adding a VisualizeScans module after the last TranslateRows module, should provide means of visualizing the resulting image.

The completed algorithm, would look something like the one, illustrated in figure 78.

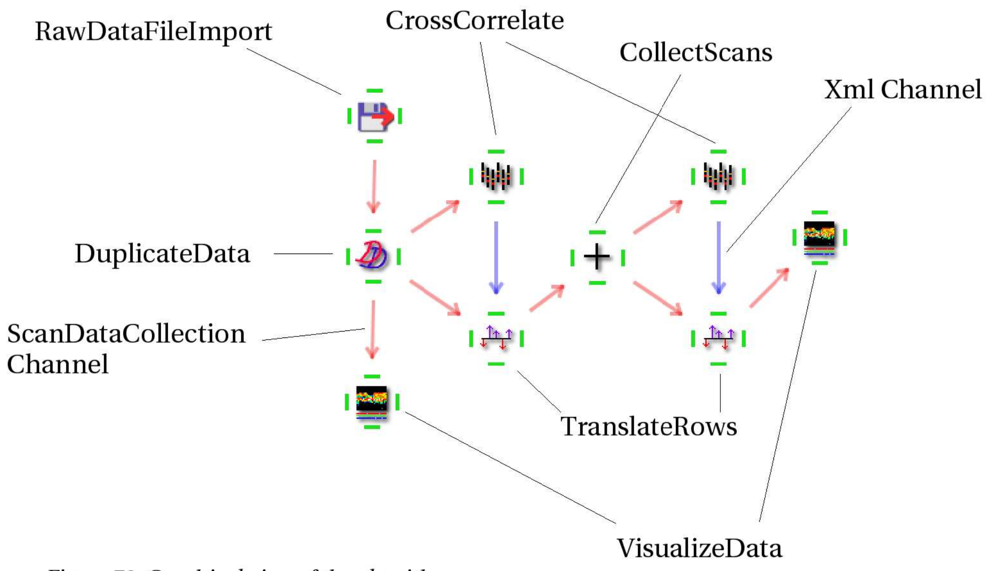


Figure 78, Graphical view of the algorithm.

An extra visualization module has been added to be able to visually inspect the incoming data before processing. The red arrows represent RAW data going in the direction of the arrow. The blue ones represent XML messages, which in this case would contain the translation vectors, resulting from the "best fit" alignment algorithm, implemented in the Cross-Correlation module.

The initial DuplicateData module is there to make sure, we preserve images for the first visualization.

4.1.3 Step-by-step execution and Results

Select the Raw data import module as shown in figure 79. This should bring up the details panel for the module, where we then open a set of files, acquired in a series, representing the same data (see figure 80).

Then select the first visualization module (see figure 81) and press “Consume” to start loading images, followed by a click on “Show”.The result of this, should be a series of visualized raw data files, as shown in figure 82.

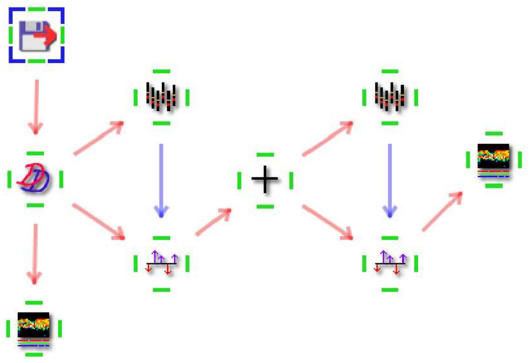
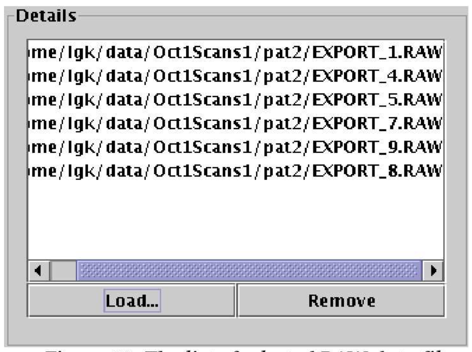


Figure 79, Selecting the RAW data import module as the first step of using the algorithm.



Details

```
me/lgk/data/Oct1Scans1/pat2/EXPORT_1.RAW
me/lgk/data/Oct1Scans1/pat2/EXPORT_4.RAW
me/lgk/data/Oct1Scans1/pat2/EXPORT_5.RAW
me/lgk/data/Oct1Scans1/pat2/EXPORT_7.RAW
me/lgk/data/Oct1Scans1/pat2/EXPORT_9.RAW
me/lgk/data/Oct1Scans1/pat2/EXPORT_8.RAW
```

Figure 80, The list of selected RAW data files for import.

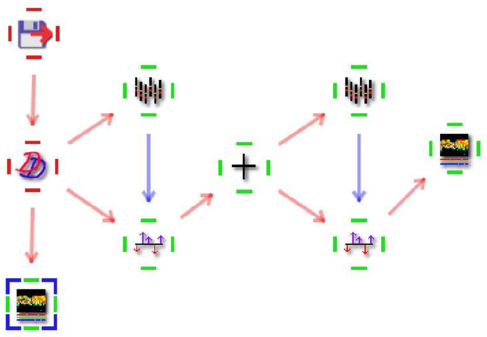


Figure 81, Algorithm diagram with the left most visualization module selected.

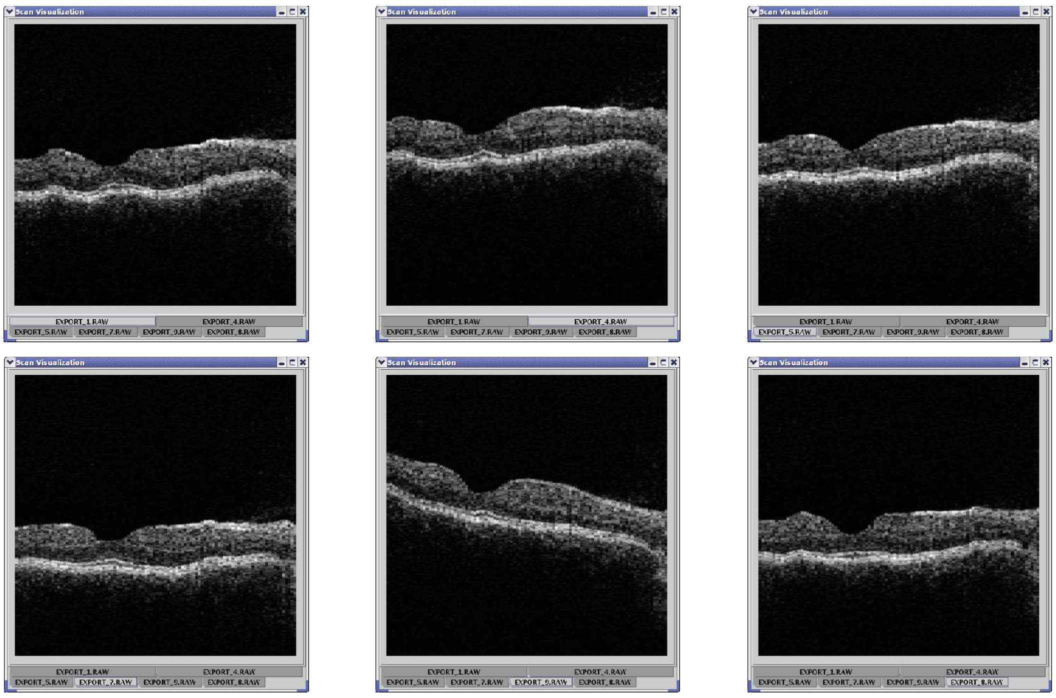


Figure 82, The six unprocessed raw data scans.

A few more visualization modules are inserted to be able to visually track the changes happening to the RAW data, as illustrated in figure 83. The next visualization module is used to verify that the other images (after the first in the collection) are a aligned nicely with the first image (see figure 84).

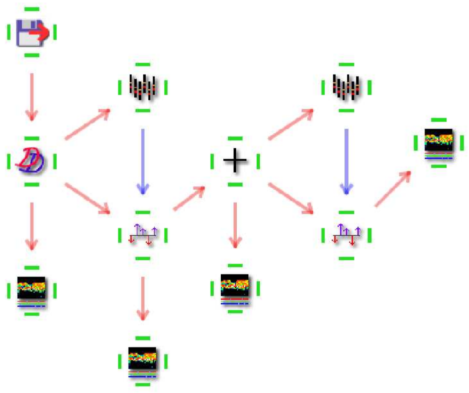


Figure 83, Algorithm diagram with extra visualization modules inserted.

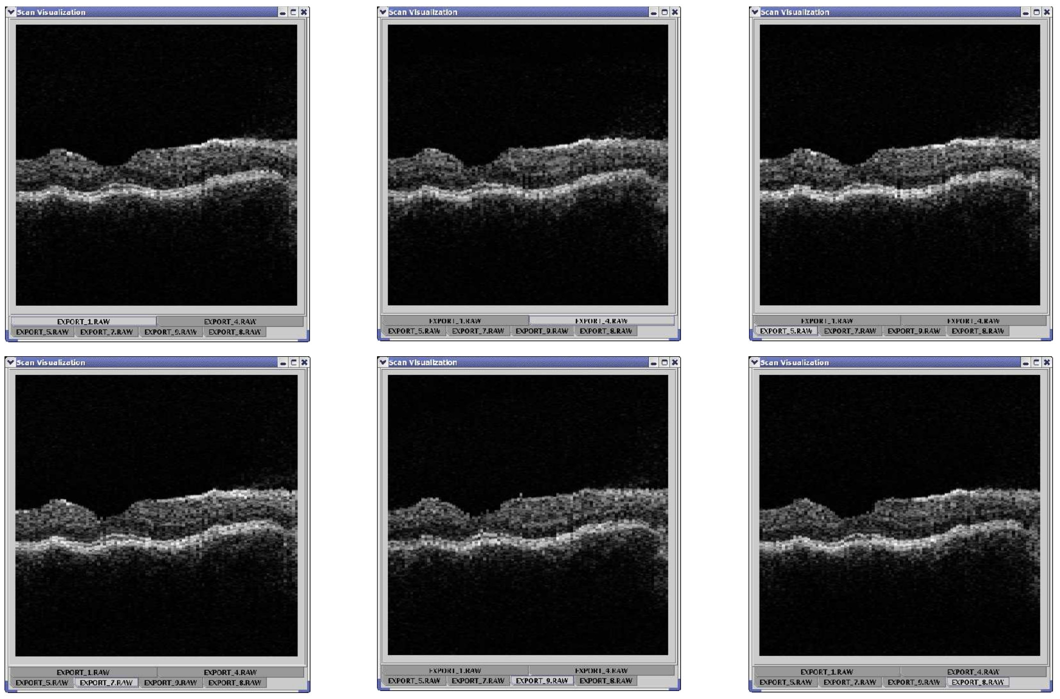


Figure 84, Images 2-6 are aligned with the first image. This is a necessary step before pixel-wise addition can be performed.

Using the visualization module below the CollectScans module, a view of the image resulting from doing pixel-wise addition of the six aligned images can be shown (see figure 85). The last step, is to use a "Cross-Correlation" module to straighten out the final image, by aligning subsequent rows in the data to one another. Selecting the final (rightmost) visualization module, pressing "Consume", then "Show" should result in the display, shown in figure 86.

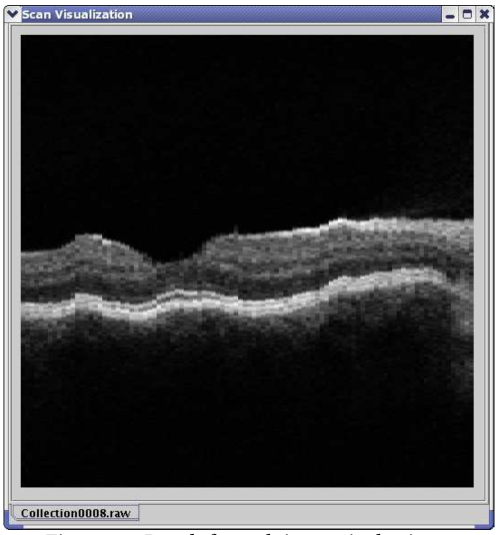


Figure 85, Result from doing a pixel-wise addition of the six aligned images. There is a clear improvement of the signal/noise ratio compared to the original images.

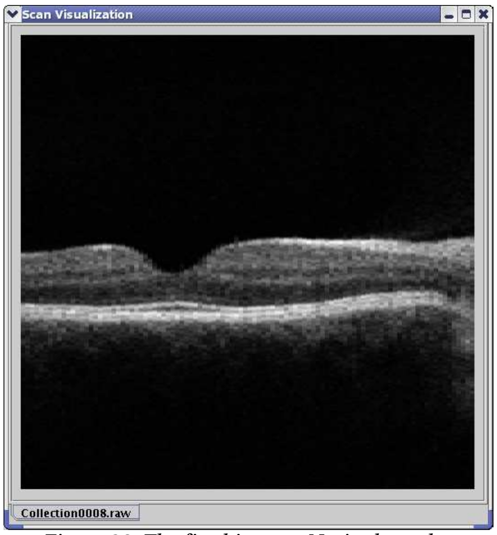


Figure 86, The final image. Notice how the noise is suppressed and the finer details in the retina begin to appear.

Looking at the final image produced (in figure 86), it is clear to see that the signal/noise ratio has been successfully improved, compared to the incoming images in figure 82. The image is aligned and finer details are now visible, providing a better visual feedback to the user. Ultimately, this could possibly help in the detection of some diseases, too hard to find in the original images.

4.2 Filtered phantoms

When the standard phantom patterns become insufficient for testing of a certain modules capabilities, experimenting with phantom images mixed with different filters might produce what we are looking for.

In this section, a test scenario will be made, that would be useful in testing a module produced during the project, by a potential future developer and user of the system. The module is called SumOfSquaredDiff and described in detail in section 5.1 on page 121.

Let's start with an idea of the kind of phantom data pattern that would be desirable for the task:

- As the SumOfSquaredDiff module is a module, that seeks to align data, the pattern should include some sort of wavy line to be aligned.
- Noise must be added to make the scenario more realistic.
- The pure phantom patterns are too “nice and clean”. Therefore, a combination of patterns and image filter should be made, to make a “rough” effect.

Converting the loose requirements stated above into something producible by the existing system modules, the resulting pattern could be described along the lines of: “A sine wave pattern, combined with a salt & pepper noise pattern. Both modified prior to addition by a selection of mean and gradient filters, creating the effect of a blurry wavy line on a blurry noisy background”.

A bit of experimentation, lead to to the following constellation of modules:

- 2 RawDataPhantom modules,
- 5 FilterConvolution modules,
- 2 Normalize modules,
- 1 CollectScans module.
- 4 VisualizeScans modules,
- 2 TranslateRows modules,

- 1 DuplicateData module,
- 1 HistogramCutoff module and
- 2 SumOfSquaredDiff modules (see section 5.1 on page 121).

Connect the modules as shown in figure 87.

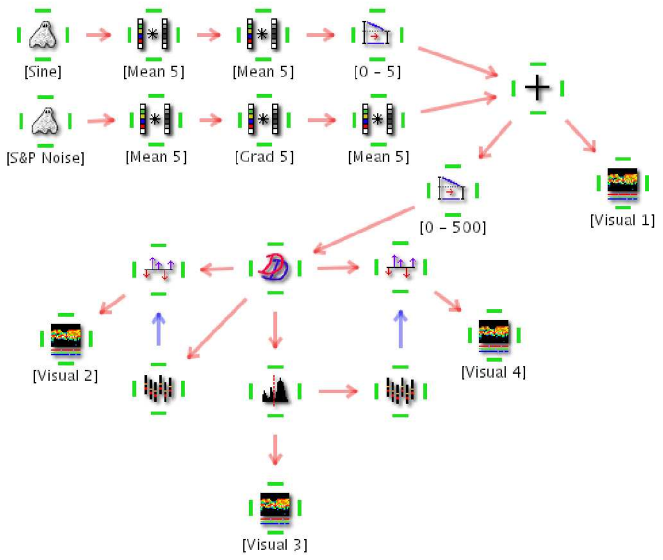


Figure 87, Diagram of the algorithm, constructed to test the SumOfSquaredDiff module. The module labels indicate settings for phantoms, filters and normalization modules.

The four visualization modules (labeled in the diagram) should show the following images:

1. Result of adding the two phantoms
2. The result of trying to align a noisy images
3. The same image as in [Visual 1], but with noise reduced by a histogram cutoff
4. Final image, alignment based on noise reduced image

The phantoms and filters are configured as indicated by the labels in figure 87.

Selecting [Visual 1], followed by a press “Consume” and “Show” will result in the noisy sine wave pattern[13] in figure 88. [Visual 2] is used to see the results (see figure 89) of trying to directly align this image.

It's clearly visible, that the alignment method has failed. The noise must be eliminated by using the histogram cutoff module. A separation between noise and signal values is found, illustrated in figure 90. [Visual 3] is used to see how well the noise has been removed (see figure 91).

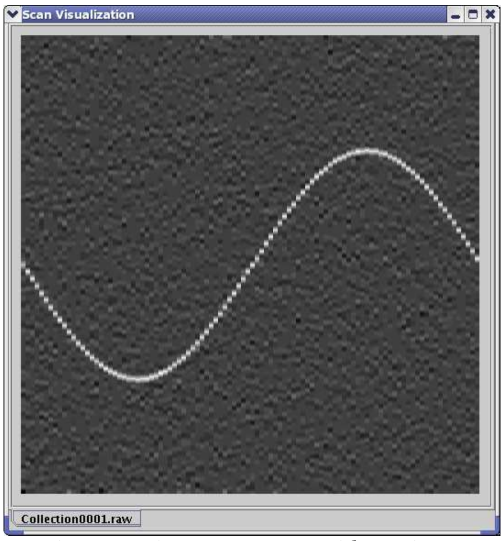


Figure 88, Sine wave pattern with a noisy background.

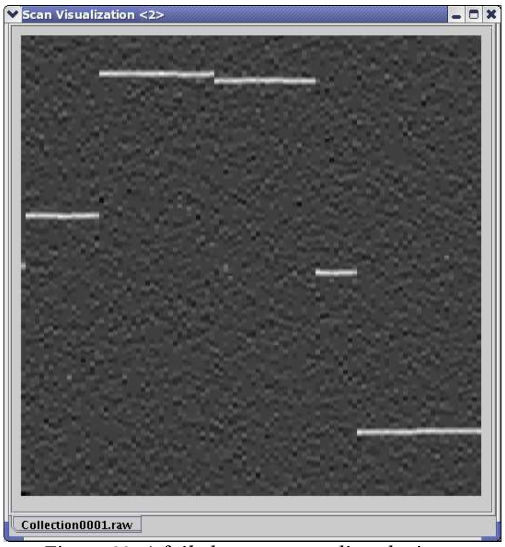


Figure 89, A failed attempt to align the image.

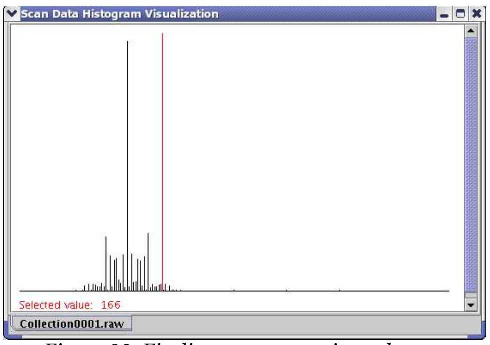


Figure 90, Finding an appropriate place to set the threshold.

13 Results will vary, as the random number generator used in constructing the Salt & Pepper noise pattern is based on a “current time” seed.

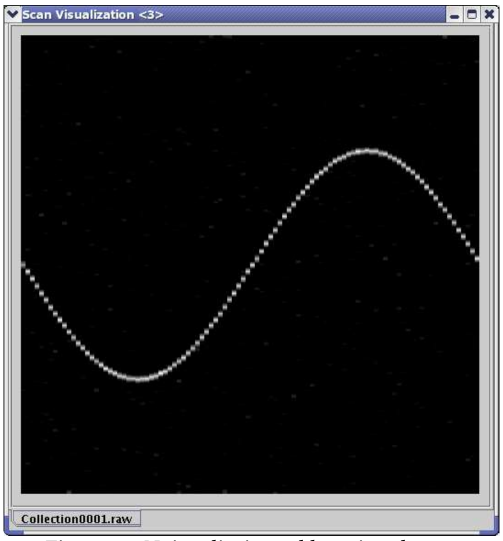


Figure 91, Noise eliminated by using the histogram cutoff module.

Finally, an alignment vector based on the image in figure 91, is calculated and used on the image in figure 88. The resulting image, illustrated in figure 92, is visualized, using [Visual 4]. This example, shows one of the powerful features of the system, where it is possible to keep the original image information intact (figure 88) when aligning based on a modified version of the data (figure 91).

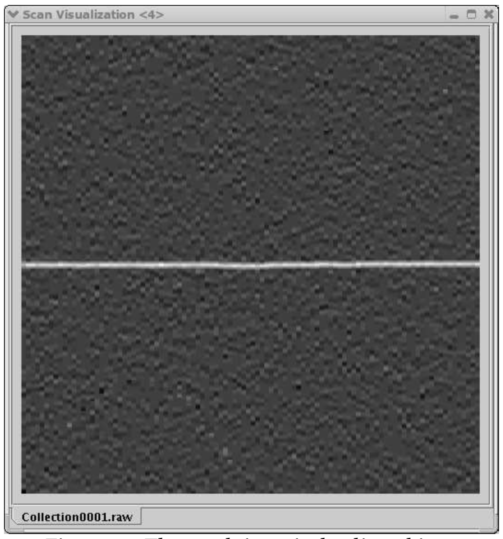


Figure 92, The result is a nicely aligned image.

4.3 Summary

Constructing algorithms using the framework and modules provided is fairly easy. After getting an idea of how a certain algorithm should be constructed, it doesn't take very long to get results.

In this chapter, two different example algorithms have been presented.

1. An example of how collecting scans can drastically improve image quality
2. An all phantom generated image being used to verify functionality of several modules.

Each of the algorithms only took a few minutes to construct, showing how simple it is to test new modules and to use the system for everyday processing of patient data.

5. User experiences

Until now, the system has only been used and tested in a local development environment. This is about to change, when the framework will be tested in two key areas by future users of the system:

- First, a developer will try to create a new module, based on a different alignment technique than the one present in the “Cross-Correlation” module.
- Second, one of the potential users of the system, creates an algorithm to achieve an enhancement of the edges in a collection of RAW data scans.

The contents of this chapter can be seen as a sort of customer approval of what has been delivered.

5.1 A developer's module

As the framework and several modules were in place, it was decided to test how easily someone else's methods and algorithms could be implemented in custom modules using the guidelines provided. Thomas Martini Jørgensen was a perfect candidate for this test, as he already had developed some of his own software in C++ for different handling of the Raw data originating from both of the OCT retina scanners. One piece of software implemented, was closely related to a paper, he had worked on [Jørgensen04], and it was decided to see if this was transferable to the framework developed in this project.

After some investigation, it was decided that one of the key methods used in his algorithm, could be implemented as a module. The method was closely related to that of the cross-correlation module but with a difference. Instead of aligning by finding the translation resulting in the maximum value, when summing pixel-wise multiplications, his method was based on finding the minimum, when summing the pixel-wise squared difference.

The result was a module, called SumOfSquaredDiff, which was added to the general module list. Integration and test of this module can be found in section 4.2 on page 116.

5.2 A user's algorithm

Now that the software is functional, it will be put to the test by letting a future user of the system design a new algorithm, using existing module components.

A meeting with Birgit Sander, M.Sc., Ph.d. and Head of Laboratory at Herlev Hospital, was arranged. The plan was to let Birgit design an algorithm, satisfying needs, not possible to achieve on the existing systems.

To begin with, Birgit had to get a feel of the system, which after a short while seemed to go quite nicely. She then decided to build an algorithm, that would take a series of images, collect them and then try to enhance the edges, using an edge detection filter, to make it easier to see and measure differences in the retinal thickness.

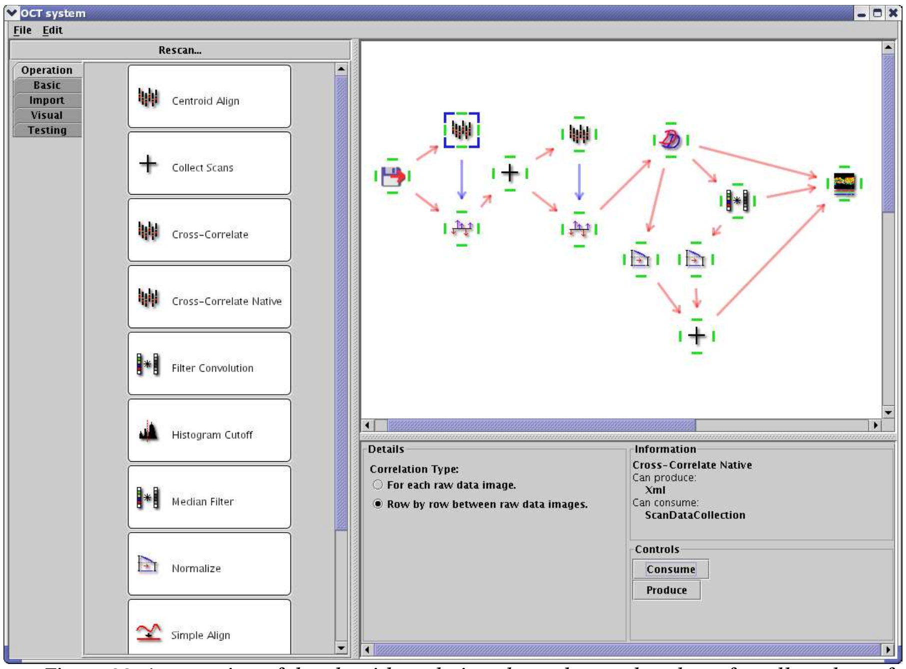
The first part of the algorithm would follow a design similar to the one described in section 4.1, where the signal/noise ratio is enhanced by pixel-wise addition. This is then extended with an edge detection filter (FilterColvolution module), two normalization modules and an extra CollectScans module, to combine the filtered edge detection image with the enhanced image from before. A view of the full workspace with the designed algorithm ready for use, can be seen in figure 93.

As an edge detection filter, “Gradient (width 3)” is chosen, and for both normalization modules, the following parameters are set (see figure 94):

- Minimum value = 0.0
- Range width value = 1.0

The same series of images, as in the case of the algorithm in the previous chapter, “Enhancement by addition”, were selected to be processed. The visualization module was selected, and a press on “Consume” initiated the back-propagating process up the diagram graph, where all modules connected are forced to produce data in an iterative fashion. After completion, three images were produced:

1. The result from collecting the scans (figure 95), where edges were a bit blurry, but definitely presenting a better result than what was exported from the OCT scanner,
2. an edge enhanced image (figure 96), resulting from running the collected image through the FilterConvolution module and
3. the combined image (figure 97), produced by adding images 1 and 2 after normalizing the brightness levels to be in the same range.

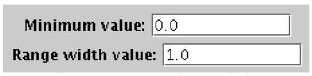


Correlation Type:
( ) For each raw data image.
(●) Row by row between raw data images.

Cross-Correlate Native
Can produce:
Xml
Can consume:
ScanDataCollection

Figure 93, An overview of the algorithm, designed to enhance the edges of a collected set of RAW data scans. The first cross-correlation module is selected, and the details are shown.

Birgit Sander was pleased with the result, as seen in her statement below (see text listing 2).



Minimum value: 0.0

Range width value: 1.0

Figure 94, Region of the details panel of the Normalize module.

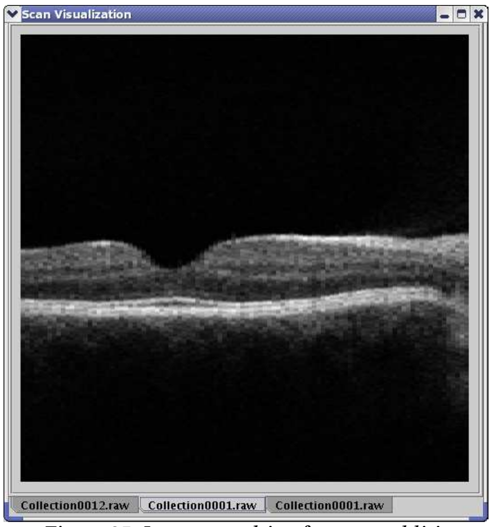


Figure 95, Image, resulting from an addition of six RAW data scans.

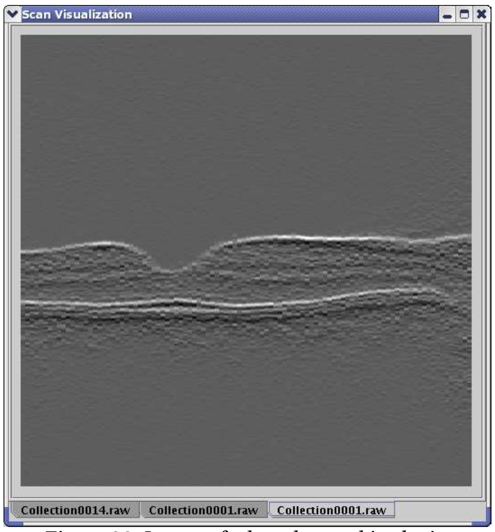


Figure 96, Image of edges detected in the image to the left (figure 95).

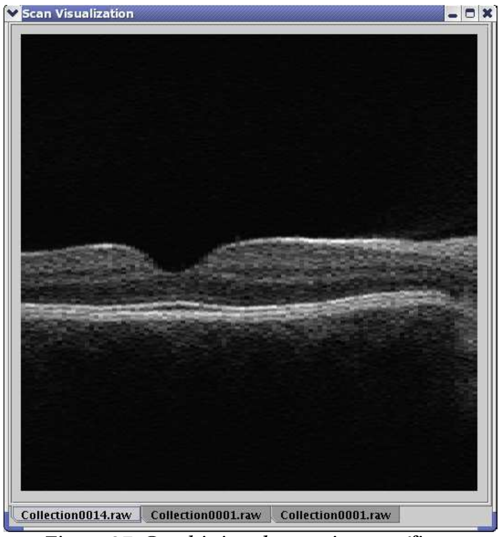


Figure 97, Combining the two images (figures 95 and 96) above to enhance the edges.

What Birgit Sander had to say after trying out the system (in Danish):

Anvendelse af OCT til klinisk brug er stærkt stigende og dermed også udviklingen af metoder, der udnytter potentialet fuldt ud. I denne forbindelse er det både den billedmæssige behandling og den kvantitative beregning, der er interessant. Det program, der nu er udviklet, giver mulighed for at bearbejde billederne med metoder, der ikke tidligere har været tilgængelige i forbindelse med OCT, idet der nu er mulighed for at anvende billedbearbejdnings-teknikker kendt fra andre områder og særligt udviklede programmer til OCT. Særligt billedsammenlægningen giver et fremragende resultat. Den store fleksibilitet og brugervenlighed gør det muligt at bruge programmet i hverdagen og at undersøge og fastlægge programsekvenser til forskellige patienttyper, hvor billedkvaliteten kan variere, og hvor de detaljer, der ønskes undersøgt, kan være lokaliseret til forskellige dele af scannet. Ved analyse af prøveeksempler i programmet var det indlysende, at der på basis af det udviklede program er mulighed for på sigt at komme videre med en kvantitativ behandling af data, som er meget brugbart ved længerevarende studier. Vi ser derfor frem til at kunne tage dette nye værktøj i brug i den daglige klinik og i forskningsmæssig sammenhæng.

- Birgit Sander, M.Sci., Ph.D.
Head of Laboratory, Herlev Hospital

Text 2, Statement made by the "customer" about the potential and usefulness of the developed system.

5.3 Summary

A new module was created, based on the design of the Cross-Correlation module, changing the way two signals are compared, to find a translation that would align them in an optimal way. The new module, bases its comparison on a pixel-wise sum of squared differences between the signals, where the old one bases it on a traditional cross-correlation. Implementation and testing of the module went fairly smooth, and the module integrated flawlessly with existing logic.

The flexible way of designing algorithms, was tested by a future user of the system, resulting in a method that would enhance the edges of collected scans, thereby making it easier to measure changes in retinal thickness. Birgit Sander, the person testing it, was very pleased with the result of the project work.

These user tests show, that developing processing modules and using the system to design new algorithms, is not limited to be done by the developer of the system alone.

Judging from the results of the tests, it's fair to say, that the initial requirements, stating that the system should be easy to extend and to use, have been met.

6. Plans for the future

Some features of the system were left out, mainly because time did not permit them to be implemented during the project period. Plans are made, however, for the author (and developer) of this project to work closely with Herlev Hospital, implementing more features after the time of writing. In this chapter, we will go through some of the more interesting planned features in detail.

As the system is based on independent modules, capable of containing virtually any sort of logic, the developer could want to implement, it would also be possible to make far more complex modules, directly communicating with external systems. This could be anything, ranging from communicating with the OCT scanners to letting a module function as a server, delivering results to a thin client requesting data, e.g. a web-browser.

6.1 Integration with other systems

One major improvement to the system will be in the form of modules integrating directly with other systems, e.g. integrating with the OCT scanners directly would eliminate the need for a user of the system to export and import data via raw data files. Modules providing this sort of feature, could possibly be developed as a joint effort between developers of the system and different manufacturers of medical equipment.

6.2 Print support

The possibility for print support in the following areas:

- Diagram views: When designing large algorithm diagrams, it might be desirable to get a hard copy of the flow of the diagram, showing the constructed algorithm. The graphical representation of the diagram, as well as a human readable version could be produced for documentation.
- Module settings: As individual module configurations might hard to overview when looking at larger algorithm designs, it would be useful to have a complete

list of all the settings for all the modules, collected on paper.

- Processed images: When applying algorithms to different RAW data images, the possibility of printing reports of the results, including the produced images, would be useful for documentation and research.

6.3 A common database between workstations

Currently, images are loaded from a disk through the RawDataFileImport module. If a group of people would like to access the same patient data (RAW data files) for processing, the introduction of a common data store could be the answer. There would be some advantages of this approach:

- **Data storage and retrieval:** Distributed information retrieval is a very nice to have feature, as it would provide the users a way for them to access and possibly process data from remote. In most cases, this would require a centralized data store, e.g. a database, being accessible over the network.
- **Possibility for an off-line processing unit:** When user client machines are too slow to handle processing of large data arrays, a powerful centralized mainframe computer could take most of the load, allowing the client to handle other tasks.
- **Web interface:** Creating a web interface for the data store, could allow restricted access to patient images, using a standard web browser

7. Conclusion

The primary goal of this project was to produce a robust and flexible image processing framework for ophthalmology researchers at Herlev Hospital, dealing with images originating from a Carl Zeiss Optical Coherence Tomography retinal scanner. Judging from the customers statement about the system, the mission of the project seems to have been successful (parts of the text translated to English):

"The use of OCT for clinical purposes is rising fast and therefore also the development of methods, exploiting the potential to its fullest… …The software developed, provides an opportunity to process the images using methods, not previously available in relation with OCT, as it is now possible to use image processing techniques known from other areas… …The great flexibility and usability makes it possible to use the software i every day use and to examine and settle on program sequences [algorithms] for different types of patients, where the image quality can vary, and where the details of interest, can be located in different places in the scan… …We therefore look forward be able to use this new tool in the daily clinic, as well as in research related work." (Birgit Sander)

Many steps are involved in making such a system, and it has been a very educational process to investigate and actually try out what is demanded of a software developer, when set out to produce systems, that are meant to be used outside the university classroom.

Making the system requirements alone was a long process, involving many iterations before settling on the result provided in Appendix A. This, however, proved to be very useful for the rest of the project, especially as they would always be there as overall guidelines, when focus was lost, due to all kinds of exciting possibilities, just waiting to be explored in the system.

Before starting any development, it was important to search for tools, capable of providing the functionality needed for the project, as there is no reason to reinvent the wheel. After doing a wide search for similar image processing frameworks, only finding a few commercial tools that matched, it became clear, that if the system *was* to be built as extensions to existing software, the licensing costs alone would be much too high for the system to be widely adopted. This meant, that the framework would have to be built as part of the project.

It was not easy, making the decision to scrap all the C++ code developed after two months into the project. At that time, a minimalistic - but functional - graphical user interface, as well as some core processing part were already in place. Serious considerations were made on how to reuse as much as possible of the existing code, but being based so heavily on the use of template libraries in C++, the integration task seemed almost impossible. The decision, to use Java all the way, proved to be right in several ways:

1. Java, being multi-platform by nature, allowed the development to be focused on building the actual framework and processing modules, instead of spending valuable time, just to make the system compile on different operating systems.
2. Many technologies used, are shipped with the standard Java SDK, including XML processing, GUI functionality, basic image input/output (e.g. used to load icons), etc.. This meant, that future developers of the system would not have to worry about compilation and installation of several 3rd party tools.
3. As Eclipse, the powerful IDE used, as well as the Java SDK itself being freely available for many popular operating systems, the costs put on users and developers is limited to the cost of the hardware used

Designing the system, so it would easily be integrated with legacy software, *did* at first seem like a very overwhelming task, as it would be almost impossible to make something that would integrate with every thinkable piece of code out there. After giving it some thought though, a solution began to appear. Using a combination of making everything communicate using XML *and* make all the processing logic in the system be based on small modular components, would enable users, wanting to integrate the framework with other systems, to either make their existing systems communicate through the use of XML – or to make custom designed modules, that could directly connect with their legacy systems, using any protocol they wanted.

This solution got an extra dimension to it, after swapping to Java, as developers would now have the choice of using Java, C or C++ when integrating, because of JNI, the Java Native Interface.

In the case of the Cross-Correlation module, two implementations were made. One in pure Java and one using JNI to make a bridge to a native library file, utilizing the very optimized Blitz++ library. Benchmarks showed, that significant performance improvements could be made.

Using XML provides other benefits as well. For example, when extending the system modules, or the framework itself, with new functionality involving changes in state information needed to be stored and retrieved from the workspace file, the system would in most cases be fully capable of loading files, stored with different versions of the framework. This is due to the fact that XML is capable of storing data in an unstructured way, where each component parsing the document, can concentrate on the parts of the XML it knows - not affecting and not *being* affected by the rest.

A fair deal of pluggable modules were created during the project. These, combined with the underlying framework, provide a solid base for researchers to construct algorithms, useful in their daily work with retinal OCT images.

This, however, does not mean, the system is limited to processing images of this type only. With very little effort, new import modules could be constructed to extend the systems use to other areas.

Successful integration of the code and methods, developed by Thomas Martini Jørgensen, gave extra confidence to the fact, that the system actually works.

It was always the intention, that this report should not only function as the product of a master thesis, to be put on a shelf and be forgotten, but also work as a reference manual for developers, as well as a users guide for researchers constructing algorithms. Major parts of this document will be moved to a project web page, where the continuously growing list of modules, including source code base and detailed descriptions, is going to reside as well.

Throughout the development process, I've tried to focus a lot on “the bottom of the iceberg”, and not just choosing the easy way out. This has shown to pay off, as I now feel, I can honestly say, the system produced really lives up to expectations of being robust, easy to use and designed for change.

Appendix A: System requirements

The requirements are split into three sections:

- **Framework:** The overall system and underlying framework requirements
- **Image processing:** Image handling methods and image processing techniques required to be implemented in the system.
- **GUI:** Visualization requirements, as well as providing a basic set of guidelines for what actions should be possible through user interaction

Framework requirements

Table explanation
Requirement ID: A traceable ID.
Priority: Must, Should, Could, Won't [MoSCoW model]

Requirement ID	***Priority***	***Description***
REQ-FRA-001	Should	Build on available 3rd party components where in-house development seems unreasonable.
REQ-FRA-002	Must	Run on multiple platforms, with Windows and Linux as the two primary targets.
REQ-FRA-003	Should	Be easy to integrate with existing software.
REQ-FRA-004	Could	Provide a text based interface to the system.
REQ-FRA-005	Must	Provide a graphical interface to the system.
REQ-FRA-006	Should	Minimize the amount of coding needed, when extending the system.
REQ-FRA-007	Must	Provide a way of saving and loading the workspace and its settings.
REQ-FRA-008	Should	Create a framework where future research/algorithms can be integrated as pluggable components (e.g. Using *.so files [*.dll for windows])

Image processing requirements

Requirement ID	*Priority*	*Description*
REQ-IMG-001	Must	Import RAW data files generated by the Zeiss Humphrey OCT system.
REQ-IMG-002	Must	Be possible to perform visual alignment of RAW data files, to compensate for errors caused by eye movements, etc..
REQ-IMG-003	Must	Be possible to make a combined alignment and addition of a series of RAW data files to enhance the signal/noise ratio in data.
REQ-IMG-004	Should	Include functionality for doing histogram related operations.
REQ-IMG-005	Should	Include functionality for processing images, using filter kernel convolution.
REQ-IMG-006	Should	Include a median filter.
REQ-IMG-007	Should	Support floating point pixel types.

GUI requirements

Requirement ID	*Priority*	*Description*
REQ-GUI-001	Must	Visualization of RAW data files generated by the Humphrey system.
REQ-GUI-002	Must	Ability to control image processing features of the underlying system/libraries.
REQ-GUI-003	Must	Run on a recent version of Windows.
REQ-GUI-004	Must	Run on a recent release of RedHat Linux.
REQ-GUI-005	Should	Run on generic Unix systems.
REQ-GUI-006	Should	Provide an organized view over available OCT data available on disk.
REQ-GUI-007	Could	Provide an interface to a PostgreSQL (or MySQL) database for remote data storage.

Appendix B: Functional specifications

The functional specification lists minimalistic design considerations to each of the system requirements (listed in Appendix A).

Framework

Requirement ID	***Description and suggestion for solution***
REQ-FRA-001	"Build on available 3rd party components where in-house development seems unreasonable." *Solution:* A market analysis must be performed, trying to uncover if existing tools can be used to build upon. These would preferably be distributed under an open-source licence.
REQ-FRA-002	"Run on multiple platforms, with Windows and Linux as the two primary targets." *Solution:* This prevents the use of operating specific APIs, including that of window/GUI functionality. A survey should be done to find multi-platform libraries for GUI, math, XML, etc.. All code should be possible to compile and run under Windows and Linux.
REQ-FRA-003	"Be easy to integrate with existing software." *Solution:* Using XML for all non-binary communication, the system would be easy to integrate with.
REQ-FRA-004	"Provide a text based interface to the system." *Solution:* The system must be based on a Model-View-Controller architecture, so the view may be changed from graphical to text mode with as little effort as possible.
REQ-FRA-005	"Provide a graphical interface to the system." *Solution:* A 3rd party GUI toolkit will be used to create the user interface.

Requirement ID	***Description and suggestion for solution***
REQ-FRA-006	"Minimize the amount of coding needed, when extending the system." *Solution:* The system should be split into two parts: 1. A framework, providing the core functionality 2. Pluggable modules, where all the statistical implementations reside. When new methods needs to be implemented, only the code for a new module needs to be made.
REQ-FRA-007	"Provide a way of saving and loading the workspace and its settings." *Solution:* The workspace should be saved in an XML format, including all state information of all modules.
REQ-FRA-008	"Future research/algorithms can be integrated as pluggable components (e.g. Using *.so files [*.dll for windows])" *Solution:* A plug-in mechanism must be implemented, that will integrate easily with existing libraries.

Image processing

Requirement ID	***Description and suggestion for solution***
REQ-IMG-001	"Import RAW data files generated by the Zeiss Humphrey OCT system." *Solution:* The RAW data format must be reverse engineered, and imported into the system in a double precision format.
REQ-IMG-002	"Be possible to perform visual alignment of RAW data files, to compensate for errors caused by eye movements, etc.." *Solution:* An alignment method, based on cross-correlation, must be implemented.
REQ-IMG-003	"Be possible to make a combined alignment and addition of a series of RAW data files to enhance the signal/noise ratio in data." *Solution:* An alignment method, based on cross-correlation, must be implemented.

Requirement ID	***Description and suggestion for solution***
REQ-IMG-004	"Include functionality for doing histogram related operations." *Solution:* Implement basic functionality for working with image histograms must be implemented.
REQ-IMG-005	"Include functionality for processing images, using filter kernel convolution." *Solution:* A generic filter kernel convolution mechanism should be implemented as a module, where different kernels can easily be created and selected.
REQ-IMG-006	"Include a median filter." *Solution:* The median filter should be implemented as a separate module.
REQ-IMG-007	"Support floating point pixel types." *Solution:* Internal data structures, as well as the operations applied on them, should be based on double precision floating point values.

GUI

Requirement ID	***Description and suggestion for solution***
REQ-GUI-001	"Visualization of RAW data files generated by the Humphrey system." *Solution:* The visualization mechanism, should be able to display the contents of RAW data files as images with brightness levels scaled to fit within 0 and 255 (maximum range of the screen).
REQ-GUI-002	"Ability to control image processing features of the underlying system/libraries." *Solution:* Where needed, the GUI should provide suitable interfaces to affect the underlying model, containing and manipulating data.
REQ-GUI-003	"Run on a recent version of Windows." *Solution:* Make sure that the 3rd party GUI library used, will compile and run under windows.

Requirement ID	***Description and suggestion for solution***
REQ-GUI-004	"Run on a recent release of RedHat Linux. " *Solution:* Make sure that the 3rd party GUI library used, will compile and run under X windows on Linux.
REQ-GUI-005	"Run on generic Unix systems." *Solution:* Make sure that the 3rd party GUI library used, will compile and run under X windows on Unix in general.
REQ-GUI-006	"Provide an organized view over available OCT data available on disk." *Solution:* A standard file chooser would function as the minimal solution to this.
REQ-GUI-007	"Provide an interface to a PostgreSQL (or MySQL) database for remote data storage." *Solution:* Integrated in a module, database access would be encapsulated and hidden from the rest of the system. The data storage interface would need to use ODBC (C++) or JDBC (Java) for communication in order to be supported on multiple platforms.

Acknowledgments

Bjarne Ersbøll (Associate Professor of Statistical Image Analysis, IMM, DTU), for great guidance and for keeping me focused on the job at hand when more interesting things caught my attention.

Thomas Martini Jørgensen (Senior Scientist, Risø National Laboratory), for introducing me to the field of optical coherence tomography and the need for better software at Herlev Hospital.

Birgit Sander (M.Sci., Ph.D., Head of Laboratory, Herlev Hospital), for being very supportive and helpful in passing on an interesting part of her research at Herlev Hospital to my knowledge. It would be interesting to carry on the collaboration, on the work we've done during the project.

Anders Hybertz Jensen, who has been a great help, guiding me away from major disasters, designing and developing in C++ - as well as putting up with all of my silly questions in the process.

Philip Anthony Nash, for his charismatic way of bringing me into his world of pure programming excellence, as well as being there as a friend when needed most – thanx Phil!

Luca Passani, who educated me about usability and its importance for success - or failure when neglected. Working with him has taught me to moderate my natural tendency to make things more complex then necessary.

Grzegorz Ciepiel, for opening my eyes to the world of patterns and good development practices.

Jens Erik Pontoppidan Larsen, for being there for me as a friend and sparring partner.

Bronagh Hannah McElduff, for helping me with the quirks of English grammar ;-)

My family for being supportive in any way they could.

Nanna, for inspiration.

Glossary

Throughout this thesis, many terms are used, that might meet the readers eye for the first time. It was therefore decided that a glossary of some of the more uncommon terms and abbreviations was provided.

API	Application Programming Interface. An API is a series of functions that programmers can use to make the underlying layers do their dirty work. Using Javas API, for example, a program can open windows, files, and message boxes, as well as perform more complicated tasks, by passing a single instruction.
Class	A fundamental building block in object-oriented languages. The class encapsulates programming logic and functions as a sort of template for its instantiations (objects). Class functionality may be copied and extended to other classes through inheritance.
Design	The activity performed by a software developer to reach to the architecture of the system to be produced. *The Design* also refers to the product of this activity.
GCC	The GNU Compiler Collection, which currently contains front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj,...). URL: http://gcc.gnu.org
GUI	Graphical User Interface. This could be considered "the tip of the iceberg" of the application.
HTML	HyperText Markup Language. HTML is the language used for publishing hypertext on the World Wide Web. It is a non-proprietary format based upon SGML, and can be created and processed by a wide range of tools, from simple plain text editors - you type it in from scratch- to sophisticated WYSIWYG authoring tools. HTML uses tags such as `<h1>` and `</h1>` to structure text into headings, paragraphs, lists, hypertext links etc.
JDBC	Java Database Connectivity. The API used in Java to connect to databases.
JNI	Java Native Interface.

LGPL	Lesser General Public License (http://www.gnu.org/copyleft/lesser.html).
Linchpin	A central cohesive element, e.g. *Reduced spending is the linchpin of their economic program.*
OCT	Optical Coherence Tomography.
Ophthalmology	The art and science of eye medicine.
PHP	"PHP" is a recursive acronym for "PHP: Hypertext Preprocessor". PHP is a server side scripting language, usually integrated closely with a web-server.
Refactoring	A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality – simplicity, flexibility, understandability, performance.
SDK	Software Development Kit.
Thick Client	Business and presentation logic resides on the client side, while data is accessed on the server. Example of a thick client: A Java applet, with all the business logic implemented, using JDBC to access a remote database.
Thin Client	The client only has presentation logic, while data access and business logic resides on the server side. Example of a thin client: A web-browser, where data would be accessed through server side business logic, e.g. a PHP page.
Third-party software	Software coming from other entities than the two parties developing a system. E.g. in the case of the system produced in this project, Blitz++ is coming from a 3rd party.
UML	Unified Modeling Language. Used to describes software models in diagrams. A model plays the analogous role in software development that blueprints and other plans (site maps, elevations, physical models) play in the building of a skyscraper.
VM	Virtual Machine. In common usage, "virtual machine" usually refers to a piece of software that provides an implementation of a virtual instruction set/virtual CPU that runs byte codes other than that of the native CPU.

Wrapper class	Typically, this would be a class encapsulating lower level functionality, e.g. a class with methods interfacing with some low level protocol, while keeping a nice high level interaction layer (public methods).
WYSIWYG	Short for ***w**hat **y**ou **s**ee **i**s **w**hat **y**ou **g**et.*
XML	Extensible Markup Language. XML derived from the Standard Generalized Markup Language (SGML). You can use both XML and SGML to create self-describing documents. Both languages use textual markup (tags) to describe data so that other applications or tools (like an SGML or XML parser) can correctly read the information and then do interesting things with it. XML is a simplified version of SGML, more suitable for use on the Web.

Bibliography

Grand98: Grand, Mark, "Patterns in Java, Volume 1", 1998, ISBN 0-471-25839-3, Publisher: "John Wiley & Sons"

Buschmann96: Buschmann, Frank, "Pattern-Oriented Software Architecture: A System of Patterns", 1996, ISBN 0-471-95869-7, Publisher: "John Wiley & Sons"

Geary99: Geary, David M., "Graphic Java 2, Mastering the JFC", 1999, ISBN 0-13-079667-0, Publisher: "Sun Microsystems"

Arehart00: Arehart, Charles; Passani, Luca; (many others in unreferenced chapters.), "Professional WAP", 2000, ISBN 1861004044, Publisher: "Wrox Press Inc."

Berg99: Berg, Clifford J., "Advanced Java 2 development for enterprise applications", 1999, ISBN 0-13-084875-1, Publisher: "Prentice-Hall, Inc."

Josuttis02: Vandevoorde, David; Josuttis, Nicolai M., "C++ Templates: The Complete Guide", 2002, ISBN 0-201-73484-2, Publisher: "Addison-Wesley"

Beck00: Kent Beck, "Extreme Programming Explained: Embrace Change", 2000, ISBN 201-61641-6, Publisher: "Addison-Wesley"

Carstensen97: Carstensen, Jens Michael, "Digital Image Processing", 1997

Ifeachor98: Ifeachor, Emmanuel C.; Jervis, Barrie W., "Digital Signal Processing: A Practical Approach", 1998, ISBN 0-201-54413-X, Publisher: "Addison-Wesley"

Sonka93: Sonka, Milan; Hlavac, Vaclav; Boyle, Roger, "Image Processing, Analysis and Machine Vision", 1993, ISBN 0-412-45570-6, Publisher: "Chapman & Hall Computing"

Teuber89: Teuber, Jan, "Digital Billedbehandling", 1989, ISBN 87-571-1060-3, Publisher: "Teknisk Forlag"

Niblack86: Niblack Wayne, "An Introduction to Digital Image Processing", 1986, ISBN 0-13-480600-X, Publisher: "Prentice/Hall International"

Cormen96: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L., "Introduction to Algorithms", 1996, ISBN 0-262-03141-8, Publisher: "The MIT Press"

Cho93: Cho, Z. H.; Jones, Joie P.; Singh, Manbir, "Foundations of Medical Imaging", 1993, ISBN 0-471-54573-2, Publisher: "John Wiley & Sons"

Jørgensen04: Jørgensen, Thomas Martini; Ersbøll, Bjarne; Sander, Birgit; Larsen, Michael, "Reducing speckle noise in retinal OCT images by aligning multiple B-scans", 2004