

# Techniques and Software Architectures for Medical Visualisation and Image Processing

**About the cover**

The images on the front and back cover are different views of the same cadaveric scapula. The scapular surface was extracted from CT data with the DeVIDE software described in chapter 2 using the segmentation techniques described in chapter 4. It was subsequently rendered with the Renderman<sup>®</sup>-compliant photorealistic renderer AQISIS using the `dented displacement` shader and the `ivory` surface shader.

# **Techniques and Software Architectures for Medical Visualisation and Image Processing**

**Proefschrift**

ter verkrijging van de graad doctor  
aan de Technische Universiteit Delft,  
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,  
voorzitter van het College voor Promoties,  
in het openbaar te verdedigen op maandag 12 september 2005 om 10:30 uur

door

**Charl Pieter BOTHA**

Master of Science in Electronic Engineering  
Universiteit Stellenbosch, Zuid Afrika  
geboren te Worcester, Zuid Afrika.

Dit proefschrift is goedgekeurd door de promotor:  
Prof. dr. ir. F.W. Jansen

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter

Prof. dr. ir. F.W. Jansen, Technische Universiteit Delft, promotor

Ir. F.H. Post, Technische Universiteit Delft, toegevoegd promotor

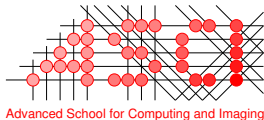
Prof. dr. A.M. Vossepoel, Erasmus Universiteit Rotterdam

Prof. dr. ir. F.C.T. van der Helm, Technische Universiteit Delft

Prof. dr. P.M. Rozing, Leids Universitair Medisch Centrum

Prof. dr. ir. B.M. ter Haar Romeny, Technische Universiteit Eindhoven

Dr. E. Gröller, Technische Universität Wien, Oostenrijk



This work was carried out in the ASCI graduate school.  
ASCI dissertation series number 117.

ISBN 90-8559-094-9

©2005, Charl Pieter Botha, Delft, All rights reserved.

<http://visualisation.tudelft.nl/~cpbotha/thesis/>

*To Barry.*



---

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Contributions . . . . .	4
1.3	Structure of this thesis . . . . .	5
<b>2</b>	<b>DeVIDE</b>	<b>7</b>
2.1	Introduction . . . . .	7
2.2	Defining DeVIDE . . . . .	8
2.2.1	Visualisation framework models . . . . .	8
2.2.2	Requirements . . . . .	9
2.2.3	DeVIDE . . . . .	10
2.3	Related work . . . . .	12
2.3.1	VTK and ITK . . . . .	12
2.3.2	AVS . . . . .	13
2.3.3	OpenDX . . . . .	13
2.3.4	SCIRun . . . . .	13
2.3.5	VISSION . . . . .	14
2.3.6	ViPEr . . . . .	15
2.3.7	Summary . . . . .	15
2.4	Making DeVIDE modules . . . . .	16
2.4.1	Requirements for integrating third party functionality . . . . .	17
2.4.2	In practice . . . . .	17
2.5	Architecture . . . . .	20
2.5.1	The Module and Module API . . . . .	23
2.5.2	Module Library . . . . .	23

2.5.3	Module Manager . . . . .	23
2.5.4	External Libraries . . . . .	25
2.5.5	Pervasive Interaction . . . . .	25
2.5.6	Graph Editor . . . . .	26
2.5.7	Mini Apps . . . . .	27
2.6	Design and Implementation . . . . .	27
2.6.1	Python as primary implementation language . . . . .	27
2.6.2	Module Application Program Interface . . . . .	29
2.6.3	Execution model . . . . .	33
2.6.4	Introspection . . . . .	34
2.6.5	Data types . . . . .	36
2.6.6	Module Library . . . . .	36
2.7	The prototyping process in DeVIDE . . . . .	40
2.8	Discussion . . . . .	41
<b>3</b>	<b>DeVIDE Applications</b>	<b>45</b>
3.1	Pre-operative planning for glenoid replacement . . . . .	45
3.1.1	Segmentation . . . . .	48
3.1.2	Planning . . . . .	49
3.1.3	The Glenoid Drill Guide . . . . .	51
3.1.4	Future work . . . . .	54
3.2	Visualisation of chorionic villi and their vasculature . . . . .	54
3.2.1	Method . . . . .	54
3.2.2	Results . . . . .	56
3.3	Pelvic floor displacement from MRI . . . . .	58
3.3.1	Surface derivation . . . . .	59
3.3.2	Measuring the displacement . . . . .	61
3.3.3	Results . . . . .	61
3.4	Conclusions . . . . .	62
<b>4</b>	<b>Segmentation of the shoulder skeleton</b>	<b>63</b>
4.1	Introduction . . . . .	63
4.2	Requirements and complications . . . . .	64
4.3	Related Work . . . . .	66
4.4	Method Overview . . . . .	67
4.5	Deriving Structure Masks: Method A . . . . .	68
4.5.1	Histogram Segmentation . . . . .	68
4.5.2	Connected Components Labelling and Selection . . . . .	71
4.5.3	Inverse Masking . . . . .	73
4.5.4	Modified Hole-filling . . . . .	74
4.6	Edge features and Initial Level Set . . . . .	77
4.7	Level Set Segmentation and Topology Preservation . . . . .	78
4.7.1	Curve Evolution by Level Sets . . . . .	80



4.7.2	Geometric Deformable Models . . . . .	80
4.7.3	Topology Preservation . . . . .	81
4.8	Deriving Structure Masks: Method B . . . . .	82
4.8.1	The Watershed Segmentation Algorithm . . . . .	83
4.8.2	Selection . . . . .	83
4.8.3	Refinement by level set-based deformable model . . . . .	84
4.9	Results . . . . .	84
4.10	Conclusions and Future Work . . . . .	86
<b>5</b>	<b>Transfer Function Specification for DVR</b>	<b>89</b>
5.1	Introduction . . . . .	89
5.2	Related Work . . . . .	91
5.3	Simple data-registered feedback . . . . .	92
5.4	Data-registered predictive volume rendering . . . . .	94
5.4.1	Mathematical Preliminaries . . . . .	96
5.4.2	Algorithm . . . . .	97
5.4.3	Implementation . . . . .	99
5.5	Results . . . . .	99
5.6	Conclusions . . . . .	102
<b>6</b>	<b>ShellSplatting</b>	<b>105</b>
	Abstract . . . . .	105
6.1	Introduction . . . . .	105
6.2	Related Work . . . . .	106
6.2.1	Splatting . . . . .	107
6.2.2	Shell Rendering . . . . .	108
6.3	The ShellSplatting Algorithm . . . . .	110
6.3.1	Calculation of splat polygon . . . . .	110
6.3.2	Back-to-front shell voxel traversal . . . . .	112
6.4	Results . . . . .	112
6.5	Conclusions and Future Work . . . . .	115
<b>7</b>	<b>Improved Perspective Visibility Ordering</b>	<b>117</b>
7.1	Introduction . . . . .	118
7.2	Previous Work . . . . .	120
7.3	PBTF . . . . .	123
7.4	IP-PBTF . . . . .	125
7.4.1	Constructing the IP-PBTF . . . . .	125
7.4.2	Analysis . . . . .	127
7.4.3	Implementing an interleaved split-dimension traversal . . . . .	130
7.4.4	Efficient interleaving with space skipping . . . . .	130
7.5	Results . . . . .	135
7.6	Conclusions . . . . .	137

<b>8</b>	<b>Conclusions and Future Work</b>	<b>139</b>
8.1	Conclusions . . . . .	139
8.1.1	Unifying visualisation and image processing . . . . .	140
8.1.2	Keeping the human in the loop . . . . .	140
8.1.3	Using a very high level interpreted language . . . . .	141
8.2	Future work . . . . .	141
8.2.1	Medical visualisation . . . . .	141
8.2.2	General visualisation techniques . . . . .	142
	<b>Colour Figures</b>	<b>145</b>
	<b>Bibliography</b>	<b>153</b>
	<b>List of Figures</b>	<b>163</b>
	<b>List of Tables</b>	<b>171</b>
	<b>Summary</b>	<b>173</b>
	<b>Samenvatting</b>	<b>175</b>
	<b>Curriculum Vitae</b>	<b>177</b>
	<b>Acknowledgements</b>	<b>179</b>

# CHAPTER 1

---

## Introduction

---

### 1.1 Motivation

A simple though accurate definition of scientific visualisation can be found in [73]:

*Scientific visualisation is the art of turning raw data into pretty pictures.*

Creating pretty pictures from raw data enables us to make use of the highly developed human visual system and its cognitive back-end to explore data, to find patterns, to test hypotheses and to discover new phenomena. In other words, scientific visualisation is a way of turning raw data into knowledge, or to gain insight from numbers. This is a pleasing realisation, as it reminds us of the famous quote by Richard Hamming:

*The purpose of computing is insight, not numbers.*

The road leading from raw data to visual representations is often described as being a pipeline consisting of three stages, namely *filtering*, *mapping* and *rendering* [27]. Raw data flows through this visualisation pipeline and is successively transformed until a suitable visual representation results. The *filtering* stage processes the data to prepare it for visualisation. Examples of filtering operations are interpolation, smoothing, segmentation and feature extraction. The *mapping* stage converts the filtered data into geometry that can be rendered. Examples of mapping operations are iso-surface extraction, particle advection, creating graphs and colour-coding. The *rendering* stage refers to the actual process of rasterising geometry to create images. Continuous interaction and feedback greatly improve the effectivity of any visualisation effort. Throughout the visualisation process, the user adjusts filtering, mapping and rendering parameters whilst monitoring the visual results of these changes.

When scientific visualisation techniques are adapted and applied to medical problems, we are dealing with a specialisation of scientific visualisation called medical visualisation. The work documented in this thesis focused on investigating scientific visualisation techniques for shoulder replacement (see section 3.1 for more information on shoulder replacement) and was performed as part of the DIPEX project.

DIPEX, or Development of Improved Prostheses for the upper EXtremities, was a research effort by the Delft University of Technology in cooperation with the Leiden University Medical Centre. Its aims were to improve the current state of the art in shoulder replacement by developing improved prostheses and also improved techniques for planning and performing shoulder replacements. The research programme consisted of six projects:

1. Task analysis of the surgical process.
2. Image processing, visualisation and computer aided surgery.
3. Functional assessment.
4. Fixation of the endoprosthesis.
5. Design of improved endoprosthesis.
6. Protocols and surgical instruments for placement.

Our work was performed as part of the second project.

Although much visualisation and image processing research has focused on replacement procedures for the other joints, there is almost no visualisation and image processing infrastructure available that targets shoulder replacement. A software platform was required to facilitate experimentation with various visualisation and image processing techniques in this context.

A powerful approach to this type of experimentation is offered by so-called data-flow application builders that allow flexible linking of functional modules to build up an image processing or visualisation pipeline from a library of standard modules. However, medical imaging problems often call for a combination of visualisation *and* image processing techniques in a single application. This combination was quite rare in the available application builders.

Also, existing application builders tend to require a high level of overhead effort from module developers. In other words, extending an application builder with a new functional module or algorithm requires significantly more effort than the implementation of the algorithm itself. Existing systems are more oriented towards the network builder than towards the module developer. During the research and development of new visualisation and image processing techniques, the researcher often takes on the roles of both network builder and module developer.

Once a new module has been created, experimenting with variations of the implemented algorithm or simply experimenting with algorithm parameters can be a tedious process with existing platforms. Modifying the implementation of the underlying algorithm often leads to

a recompile and re-link of the module. The ability of flexibly modifying any parameter or aspect, including the actual program code, of an algorithm at run-time, significantly speeds up experimentation.

In order to address these issues, we developed DeVIDE, or the Delft Visualisation and Image processing Development Environment. DeVIDE can be classified as a data-flow application builder, but, in a departure from the norm, it employs a very high level interpreted language as main implementation language and a traditional compiled language for the processor intensive aspects. Almost any aspect of the system can be modified, in various ways, at runtime and the results of these changes are immediately visible. Creating and refining new modules has been made as flexible and low-effort as possible. DeVIDE includes, through the libraries that are utilised, a wide range of visualisation and image processing functionality that can be flexibly combined in a single application.

In chapter 2 DeVIDE is described in more detail and in chapter 3 we document three applications of the software. One is shoulder-related, the other two demonstrate the system's wider applicability.

Another important aspect of our DIPEX research, and one of the reasons for the creation of the DeVIDE software, was the investigation of pre-operative planning functionality for shoulder replacement. A crucial component of any pre-operative planning solution, as well as for any kind of measurement functionality or structural modelling, is the availability of patient-specific models of the relevant anatomical structures. These models are dependent on accurate segmentation of medical datasets. To our knowledge, there was no literature dealing with the segmentation of the skeletal structures of the shoulder from CT data. Due to the complex geometry of the shoulder joint and the fact that most shoulder replacement patients suffer from bone and cartilage altering diseases, this is a challenging problem. We have developed a shoulder segmentation approach that shows very promising results even on CT data of severely affected joints. This work is documented in chapter 4.

Volume visualisation is a scientific visualisation technique that is often applied to medical datasets. This technique can be performed in three ways: rendering two-dimensional slices of the volume, rendering surfaces that have been extracted from the volume and direct volume rendering [14]. In the case of shoulder CT data, all three are applicable, but for use by clinicians on their own PCs, existing DVR implementations are not always suitable.

A light-weight and interactive direct volume rendering method was required. By light-weight, we mean that it should run on ubiquitous and lower-end graphics hardware. Blending with traditional accelerated surface rendering is desirable: for example, a surface model of a prosthesis can then be visualised embedded in a direct volume rendering of a shoulder.

To fulfil this requirement, we developed a fast direct volume rendering method that requires very little graphics hardware support, is especially suited to the rendering of bony structures from CT data and supports blending with traditional accelerated surface rendering. This renderer is called the ShellSplatter and is documented in chapter 6. The ShellSplatter was based on existing algorithms for the visibility ordering of discrete voxels on regular grids during perspective projection. However, these existing algorithms show artefacts when using voxel splats, the default rendering mode employed by the ShellSplatter. In order to solve

this problem, we developed a new perspective mode ordering for discrete voxel splats. This ordering is detailed in chapter 7.

Another crucial component of an effective direct volume rendering is a suitable transfer function. Finding such a suitable transfer function is recognised as a challenging problem in volume visualisation. We came up with two simple but effective approaches to deriving suitable direct volume rendering transfer functions. These techniques are documented in chapter 5.

The goal of the research documented in this thesis was to develop visualisation and image processing tools and techniques for shoulder replacement surgery. However, the resultant tools and techniques have proven to be generically applicable to other medical visualisation and image processing problems as well.

## 1.2 Contributions

The work described by this thesis makes the following contributions to the medical visualisation field:

- A software platform that assists with the visualisation and image processing aspects of research efforts is presented. This software differs from other similar systems in two ways:
  1. Although it can be used as delivery vehicle for visualisation and image processing implementations, it focuses on speeding up the prototyping process for the algorithm developer.
  2. Only processor-intensive parts of the system are implemented in high-level languages such as C++ and Fortran. All other parts of the system are implemented in Python, a very high-level language. Chapter 2 gives a detailed motivation for this decision.

We also show how this software can be applied in the research process.

- A comprehensive approach to the segmentation of skeletal structures from CT data of the shoulder is presented. This approach also works in cases where the patient's shoulder skeleton has abnormal bone density and where joint space narrowing has taken place. These two symptoms are often associated with various types arthritis, one of the major reasons for joint replacement in the shoulder.
- We present two techniques whereby real-time visual feedback can be presented during the direct volume rendering transfer function specification process that greatly speeds up this traditionally difficult activity.
- We present a technique for the interactive direct volume rendering of anisotropic volumes that is particularly suited to the rendering of musculo-skeletal datasets.

- Finally, we present an improved visibility ordering for object-order perspective projection volume rendering.

## 1.3 Structure of this thesis

The remainder of this thesis is structured as follows: Chapter 2 describes DeVIDE, the Delft Visualisation and Image processing Environment, a software platform that we designed for the rapid creation, testing and application of modular image processing and visualisation algorithm ideas. Chapter 3 documents three applications of DeVIDE in respectively pre-operative planning, visualisation of chorionic villous vasculature and finally, the deformation of the female pelvic floor. In chapter 4 we present an approach for segmenting skeletal structures from CT images of the shoulder. Chapter 5 describes a method for generating meaningful visual feedback during the direct volume rendering transfer specification process. Chapter 6 presents an interactive direct volume rendering that is a combination of splatting and shell rendering. Chapter 7 details an improved back to front ordering for perspective object-order rendering. In chapter 8 we present our conclusions and discuss possible avenues for future research.





---

## DeVIDE: The Delft Visualisation and Image processing Development Environment

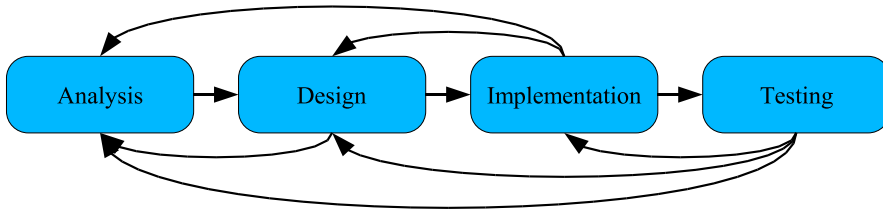
---

This chapter describes DeVIDE, the Delft Visualisation and Image processing Development Environment, a software platform that was created in order to prototype, test and deploy new medical visualisation and image processing algorithms and ideas.

### 2.1 Introduction

In the course of any informatics-related research, algorithms and ideas have to be implemented for prototyping, validation and eventually deployment. In this case, prototyping refers to the action of developing an idea by experimenting with an implementation. Here one could imagine creating a first implementation of an algorithm and subsequently experimenting with different combinations of parameters and algorithm adaptations until a suitable solution has been found. Validation refers to the phase where the discovered solution is rigorously tested under experimental conditions and its results are perhaps compared with a gold standard, if available. The validation determines whether the implementation is suitable for deployment or requires further modification or, in the worst case, is completely unsuitable to the problem. Deployment refers to the stage where an implementation is actually used in a practical situation.

During research, by far the most time is spent on the prototyping phase. The prototyping phase itself should be approached with something akin to the well known iterative software development model. This development process consists of a sequence of incremental iterations, where each iteration consists of most or all of the well-known analysis, design, imple-



**Figure 2.1:** The prototyping phase can be seen as an iterative and incremental ADIT trajectory: the problem is analysed and a solution is designed and subsequently implemented. If testing indicates that the solution is suitable, work stops. If not, we return to a previous stage to refine the process with newly-gained information. Such a return to a previous stage for refinement can happen at the design and implementation stages as well.

mentation and testing (ADIT) phases. Earlier iterations focus on analysis and design, whereas later iterations pay more attention to implementation and testing. Each iteration refines the prototype until a satisfactory result is attained.

Figure 2.1 serves as a framework for the following short explanation of the ADIT phases. During the analysis stage, the problem is identified and described. Requirements are specified for the solution. The design stage entails creating a solution based on the analysis of the problem. The resultant design is implemented and tested. The test stage indicates whether the implemented solution solves the problem. If this is the case, the ADIT is complete. If not, we return to a previous stage with the new information we have gathered during the design, implementation and testing stages. With this information, the ADIT sequence can be refined. This cyclical ADIT is repeated until a suitable solution has been found.

This cyclical prototyping phase is of central importance to many software-based research efforts and often represents a significant chunk of research time and effort. In the light of this, it makes sense to optimise the prototyping stage, as this would enable the researcher to investigate more possibilities in a shorter time. One method of optimisation is to make tools available that facilitate prototyping. This chapter describes one such a tool, called DeVIDE, or the Delft Visualisation and Image processing Development Environment.

## 2.2 Defining DeVIDE

In this section, we give a precise definition of the DeVIDE system. We start by presenting a brief overview of the different types of visualisation frameworks, followed by the list of design requirements we applied during the development of DeVIDE. We end with a concise description of our system.

### 2.2.1 Visualisation framework models

In [82], software frameworks for simulation and visualisation, also called *SimVis* frameworks, are classified as adhering to one or more of the following architectural models:

**Application Libraries** are repositories of software elements, or program code, that can be re-used via published application programming interfaces, or APIs. In practice, this means that the researcher writes new program code that makes use of an existing application library via method calls or object invocations.

**Turnkey Applications** are purpose-built stand-alone applications that can be used to experiment with a subset of the simulation and visualisation problem domain. The processing pipeline is usually fixed and the only factors that can be investigated are program parameters and input data sets.

**Data-flow Application Builders** are frameworks where software components with clearly determined inputs and outputs can be linked together to form larger functional networks that process incoming data to yield some desired derivative output data set. A graphical user interface is available that enables the user to manipulate iconic representations of the software components and to build and interact with the aforementioned functional networks.

Visualisation and image processing frameworks can naturally be described in the same way. DeVIDE mainly adheres to the *Data-flow Application Builders* architectural model, but can also be utilised as an *Application Library* or can be used to build *Turnkey Applications*.

## 2.2.2 Requirements

As mentioned in the introduction, the main and most important requirement for DeVIDE was that it should facilitate prototyping and experimentation with algorithm implementations. A secondary top-level requirement was that it should enable the delivery of code, i.e. once an algorithm has been implemented, it should be possible for a third party to make use of such an implementation. For example, if an image processing module were developed for use in finite element analysis, it should be possible for a cooperating engineer to test the new functionality within the DeVIDE infrastructure.

This section lists and defines the design requirements at a more fine-grained level. All of these fall under either or both of the top-level requirements mentioned before.

**Easy integration** The software should make the minimum of demands on the module developer. In other words, the advantages of integration should far outstrip the effort involved in integrating functionality with the framework.

**Pervasive Interaction** It should be possible to interact at all levels with all components of the software. Parameters can be changed, code can be added, removed or changed and there should be immediate feedback on the effect of these changes. This speeds up experimentation and the implicit convergence on a solution in the code and parameter domains.

**Short code-test iterations** A pattern often seen during implementation is a repetitive code modification, compile, link, run and test sequence. At every repetition, the researcher

tries a new set of parameters or code modifications that will eventually converge on the correct solution. It is desirable to speed up these iterations.

**Modularity** It is desirable to keep code as modular as possible and to keep the module interface as small as possible. This aids robustness as it is easier to pinpoint any errors. Re-use is facilitated.

**Code re-use** Being able to re-use research code that has been developed in-house and also being able to make that code available to third parties is a highly desirable characteristic. This speeds up subsequent research efforts that may make use of code written during previous attempts. It also enables the reproduction of results.

**Scalability** Regardless of the amount of functionality (number of modules) that is integrated, the framework should remain manageable and functional. With large software libraries and non-modular (monolithic) systems, the software can reach a critical mass of functionality. Adding more functionality past this point makes it increasingly difficult to manage and maintain the library and also to utilise it. In other words, there is a direct relationship between complexity and functionality. This relationship should be avoided.

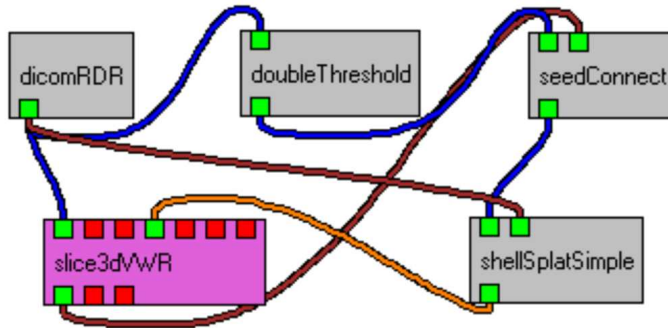
**Platform independence** The software should be portable.

### 2.2.3 DeVIDE

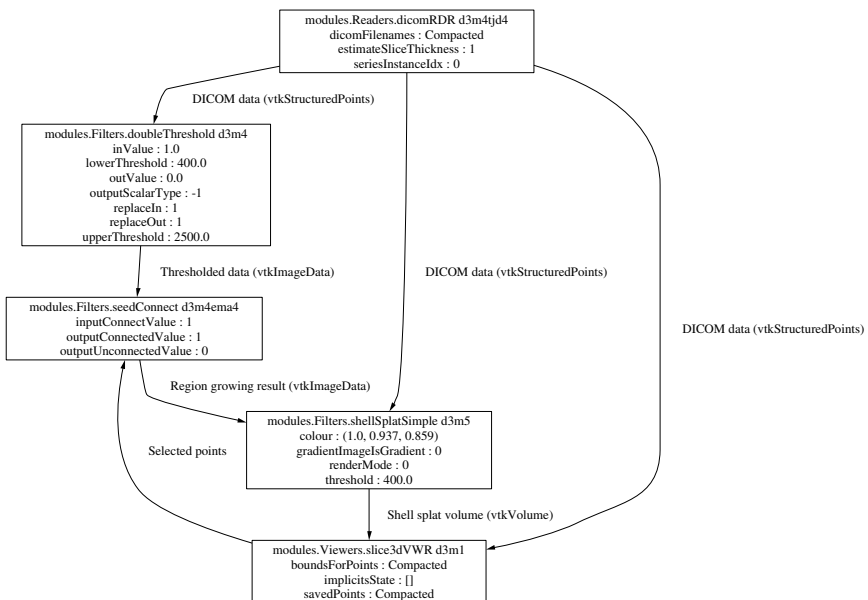
DeVIDE is a cross-platform software framework that provides infrastructure for the rapid creation, testing and application of modular image processing and visualisation algorithm implementations. Each implemented algorithm is represented by a module. Each algorithm, or module, can take input data and generate output data. The output data of one module can be connected to the inputs of one or more consumer modules. In this way, complex networks of functionality can be formed where data is successively transformed and processed by various algorithms. At any point, data can be visualised or written to permanent storage for later use.

These functional networks can be interacted with in order to experiment with parameter sets and algorithm modifications. Run-time interaction with all module and even system internals is possible, which greatly facilitates experimentation. This factor distinguishes our framework from many of the existing network-based solutions.

DeVIDE also provides an implementation of the visual programming user-interface idea, called the Graph Editor, so that the user can graphically interact with these networks of modules. Glyphs, or blocks representing modules, can be graphically placed and connected to each other. However, it is important to note that this is just one of the possibilities for interacting with the networks, i.e. DeVIDE was not designed only around this user interface element. Figure 2.2 shows an example of a simple network as rendered by the DeVIDE Graph Editor. Figure 2.3 shows a different representation of the same network where all algorithm parameters and output descriptions are also visible. In this case, DeVIDE generated a special network representation that could be used as input to external graph layout software.



**Figure 2.2:** Figure showing the DeVIDE visual programming interface to the internal representation of a simple network. DICOM data is being read by the “dicomRDR” module and then thresholded by the “doubleThreshold” module. A 3D region growing is subsequently performed on the thresholded data by the “seedConnect” module, starting from seed points indicated by the user. The result of this region growing is volume rendered with the “shellSplatSimple” module.



**Figure 2.3:** Alternative representation of the simple network shown in figure 2.2. All explicit algorithm parameters are also shown: this is useful when documenting experiments.

Because the main design requirement of DeVIDE was to optimise the prototyping process, much effort has been put into minimising the overhead required to integrate code into the framework. In spite of the fact that comparatively little effort is required to integrate with the framework, the advantages are quite significant.

## 2.3 Related work

Various systems that assist with the prototyping of and experimentation with visualisation and image processing algorithms are available. In this section, we discuss a selection of these solutions that offer functionality similar to that of DeVIDE.

In the following discussion, the terms *embedding* and *extending* will often be used when describing the integration of visualisation and image processing frameworks with language interpreters, such as Tcl or Python. When an interpreter is *embedded*, the main program is implemented in another language, for example C++ or Java, but includes an interpreter so that scripts can be executed at run-time. When an interpreter is *extended*, the main program itself is executed by the interpreter. The functionality of the interpreted language is extended by wrapping compiled code, such as C++, and making it available to the interpreter at run-time.

### 2.3.1 VTK and ITK

VTK [73], or the Visualization Toolkit, is an extensive object-oriented application library of C++ classes focused on visualisation and related data processing. Due to its native automatic wrapping system, VTK functionality can be invoked from the Python, Tcl or Java languages as well. ITK [32], or the Insight Segmentation and Registration Toolkit, is also an Application Library of C++ classes encapsulating a wide spectrum of image processing functionality. ITK makes use of a modified form of the SWIG [4] wrapping system to make available Python, Tcl and Java bindings.

Although both of these are strictly speaking application libraries, the fact that their functionality is completely addressable from interpreted languages such as Python and Tcl makes it possible to use these libraries, albeit in a limited fashion, as data-flow application builders. Both VTK and ITK employ data-flow-based processing, so an interpreted language wrapping can be used as a rudimentary text-based interface to connect and disconnect processing elements.

However, the most common use-case with VTK and ITK is that they are used to create new turnkey applications for visualisation and image processing. A turnkey application usually does not facilitate experimentation with its underlying functionality. DeVIDE integrates all functionality in both of these libraries and so makes all this functionality available through a data-flow application builder architectural model. It is possible to experiment with any combination of VTK and ITK elements at runtime. Effectively, one can experiment not only with parameters and input data, but also with different code paths.

### 2.3.2 AVS

The Advanced Visualization System, or AVS, is one of the first, if not the first, visualisation-oriented data-flow application builders [87]. AVS modules are written in the C language. This means that modules have to be compiled before they can be used and also after every change to the module code.

These modules contain a significant amount of overhead code, i.e. code that is not related to the actual algorithm implementation but to the module's integration with the AVS system. On average 50% of the module code is AVS system code [82].

AVS embeds a proprietary interpreted language called "cli" with which modules can be grouped together or other control functions can be performed.

Modules have to make use of the AVS data types for input and output data.

### 2.3.3 OpenDX

OpenDX, previously known as the IBM Visualization Data Explorer, is a data-flow application builder that focuses on data analysis and visualisation [1]. A distinguishing characteristic of this platform is its extremely generic, i.e. application-independent, data model. Roughly speaking, data is encapsulated in a data-structure called a *field*. A *field* can contain any number of named *components* that constitute the actual data storage. A *component* contains for example *positions*, i.e. the geometry of a dataset, or *connections*, i.e. the topology of a dataset, or attribute data that is associated with the geometry such as pressure or temperature.

In our experience, the consistency of this data-model speeds up the understanding of and writing new modules for OpenDX, although it is often not the most memory- or processor-efficient way to store and work with large datasets. In this sense, the more pragmatic approach of for instance VTK might be more suitable.

OpenDX modules are programmed in C and adhere quite strictly to an extensive module-programming API. The OpenDX Module Builder assists with this process by generating a module source code skeleton based on user input with regard to the desired characteristics of the new module. Modules have to be compiled before they can be used, but modified and newly compiled modules can be reloaded while the rest of the software remains running. Reloading is not always without complications: for example, if the number or type of module inputs has changed, it is often necessary to restart the server component of the software. All other runtime interaction takes place via the dedicated module user interfaces.

OpenDX expects far more effort from a module developer than a network builder. Because the module building and prototyping activity is a very important part of the complete algorithm prototyping process, we consider this and the relatively limited interaction possibilities to be weaknesses in the otherwise excellent OpenDX platform.

### 2.3.4 SCIRun

SCIRun [61] is a modern software package that focuses on providing functionality for computational steering in a visual programming setting, i.e. it is also a data-flow application

builder. This combination is called a PSE, or Problem Solving Environment. The design goal of this system was to provide scientists with a tool for creating new simulations, developing new algorithms and for coupling existing algorithms with visualisation tools.

SCIRun is a rather large and complex package that currently runs only on Unix systems. Writing a new SCIRun module entails writing a new C++ class that inherits from a special base class and exposes certain expected methods. A module optionally takes input and generates output data. All data objects are encapsulated by SCIRun-specific C++ classes. In selected cases, third party code, for example ITK objects, can be integrated as SCIRun modules by authoring a number of eXtensible Markup Language (XML) files that describe the code object that is to be wrapped, the SCIRun module that will represent it and optionally the user interface. SCIRun uses this XML description to build C++ code at a later stage.

In all cases, C++ code has to be compiled after every change to the module specification. Compiled code is dynamically loaded at run-time. SCIRun is able to compile changed module source code while the platform is running, but this obviously requires a configured and compatible build system on the host machine.

Tcl/Tk [60] was initially integrated into the system in order to act as graphical user interface layer, but it is also used for example for saving SCIRun networks and for evaluating expressions. It is also possible to interact with a running simulation via a Tcl shell window.

### 2.3.5 VISSION

VISSION [82], or VISualisation and SIMulation with Object-oriented Networks, is an interesting departure from the conventional C++-based data-flow application builder pattern. This work introduced the *Meta-C++* concept: VISSION integrates an embedded C++ interpreter, called CINT [68], that is used to interpret, at run-time, module specification files that are also written in C++.

This has several advantages: external compiled C++ libraries can be incorporated in VISSION modules without having to make any changes to the external code. The C++ interpreter is able to dynamically load and unload compiled code at run-time, which means that module specifications can be changed without having to restart the platform or recompile any program code.

This interpreter is also used as an extra interaction modality. In other words, the user can interact with the software via a text interface, by typing C++ statements that are immediately interpreted and executed by the interpreter.

Usually, a scripting language interpreter is embedded into an application written in some other compiled language in order to perform the necessary run-time parsing and execution. This is called a dual language framework. The author of VISSION argues that his single language approach for both the compiled and interpreted components is superior. Besides the relevant countering points we make in section 2.6.1, it is important to keep in mind that C++ was designed to be a compiled language, whereas there are other modern languages that have been designed and optimised for and take far more advantage of the dynamic interpreted setting in which they function.



Unlike SCIRun and OpenDX, VISSION does not require modules to make use of pre-determined types for input and output data, but instead relies on the typing of the underlying C++ code.

At the time of publication of [82], VISSION was only available for Unix-like platforms. A porting effort is underway in order to make VISSION available on Windows PCs as well.

### 2.3.6 ViPEr

ViPEr [72] is a data-flow application builder that is fundamentally the most similar to DeVIDE. Instead of embedding the interpreter in the application code, the interpreter acts as the main process and is extended with application code. For example, VISSION and SCIRun embed respectively the CINT and Tcl/Tk interpreters and in both cases the compiled C++ application code acts as the main process. In the case of ViPEr and DeVIDE however, the Python interpreter itself acts as the main process and is extended with application code that is implemented in Python and other compiled languages. This is quite an important distinction. The advantages associated with the latter philosophy will become clear throughout the rest of this chapter.

ViPEr focuses on molecular visualisation and modelling. It places no restrictions on data types, although these can be specified for documentation and interaction purposes. Modules, also called nodes in ViPEr parlance, can be constructed or modified at run-time and are specified as Python source files. The module specification framework, although facilitating the creation of modules, is not as flexible as its Python formulation allows.

Graphical interaction widgets are represented as a special kind of node that can be connected to processing nodes. Direct interaction with the Python interpreter is possible.

### 2.3.7 Summary

In this section we have briefly discussed a selection of visualisation and image processing frameworks with a clear focus on data-flow application builders. Table 2.1 summarises our findings. VTK and ITK, both application libraries, were also introduced because they constitute an important part of DeVIDE's built-in functionality. In addition, due to their Python and Tcl wrappings, they can be seen as examples of visualisation and image processing platforms that extend these languages.

OpenDX represents the more traditional approach to constructing a data-flow application builder: the whole system is implemented in a static, compiled language. Module specification takes place in the same way. There is no interpreter interface.

SCIRun embeds the Tcl interpreter and thus constitutes a traditional dual language framework. Module specification itself still takes place in C++ that has to be compiled. The module user interface is specified in Tcl. In selected cases, an XML module specification can be automatically translated into C++.

VISSION is an interesting development in that it embeds a C++ interpreter and thus offers all the advantages of an embedded interpreter, but in a single language. Modules are specified

Framework	Module Spec.	Typing	Interpreter	Platforms
VTK	C++	VTK-specific	Python/Tcl - ext	Unix/Win
ITK	C++	ITK-specific	Python/Tcl - ext	Unix/Win
AVS	C	AVS-specific	cli - emb	Unix
OpenDX	C++ / MDF	DX-specific	None	Unix/Win
SCIRun	C++ / XML / Tcl	SCIRun-specific	Tcl - emb	Unix
VISSION	Meta-C++	Underlying C++	Meta-C++ - emb	Unix
ViPEr	Python	Dynamic	Python - ext	Unix/Win
DeVIDE	Python	Dynamic	Python - ext	Unix/Win

**Table 2.1:** A selection of existing frameworks and some of their distinguishing characteristics. *Module Spec.* refers to module specification, and is an indication of how new modules are created for the relevant framework. In the *Interpreter* column, *ext* and *emb* signify *extended* and *embedded* respectively.

in Meta-C++, which is interpreted, i.e. not compiled, at run-time. This does simplify the process.

ViPEr extends Python with application code, and is the most similar to DeVIDE. Both offer extensive run-time interaction functionality via the extended interpreter.

The main differences between ViPEr and DeVIDE are:

- The DeVIDE module specification process is much more flexible than that of ViPEr. In ViPEr, a node is defined mostly in terms of a number of variables. The module execution code is defined as a text string. In DeVIDE, one defines a number of methods. In Python this is a smaller distinction than in other languages, but the approach taken in DeVIDE affords the programmer significantly more flexibility.
- DeVIDE offers more possibilities for interaction with the underlying logic.
- DeVIDE modules tend to encapsulate higher level functionality than ViPEr nodes. This is more a philosophical difference than a design limitation.
- DeVIDE is focused on biomedical visualisation and image processing, ViPEr specifically on molecular visualisation and modelling.

With regard to ease and speed of integration, i.e. new module specification, both ViPEr and DeVIDE constitute a significant improvement over any of the other mentioned frameworks.

## 2.4 Making DeVIDE modules

The module is a centrally important concept in the DeVIDE framework. Due to this, and to the fact that this facilitates the following detailed explanation of our system, we start our

exposition by explaining in practical terms how the module developer goes about creating and refining a new DeVIDE module.

First, the requirements for integrating external code as new modules are set out. Subsequently, we look at the actual practice of creating and refining a new module by making use of a simple example. This serves as a more high-level description of module integration. Sections 2.5 and 2.6 document respectively architectural and implementation details.

### 2.4.1 Requirements for integrating third party functionality

For any functionality to be integrated into DeVIDE in the form of modules, that functionality has to satisfy two requirements:

1. It must support data-flow-based processing. In other words: incoming data is copied, transformed and then made available at the output. In the large majority of cases, code that does not follow this model can simply be supplied with a wrapping that does, without changing the underlying code.
2. It must be callable from Python. This last requirement is usually straight-forward to satisfy. If Python bindings are not available, these are easily created manually or semi-automatically with packages such as SWIG [4] or Boost.Python<sup>1</sup>. There are various other ways to invoke external functionality from Python, such as for example making use of an operating system pipe.

The functionality available in the bundled external libraries as explained in section 2.5.4 is quite extensive and immediately available to new DeVIDE modules. This functionality already satisfies both these requirements.

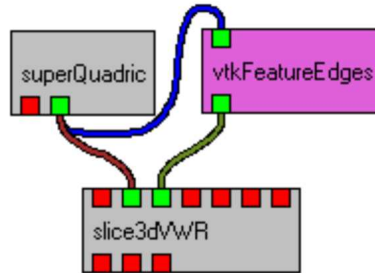
As an illustration of the flexibility of DeVIDE module integration, it is also possible to wrap independent external applications as DeVIDE modules. We have done this for instance with an existing implementation of an efficient closest point transform algorithm [50].

### 2.4.2 In practice

Creating a new DeVIDE module is a straight-forward process. For the sake of this exposition, we will make use of a simple example: a user is testing discrete curvature calculation methods on the surface mesh of a synthetic object, in this case a super-quadric toroid. Because the curvature visualisation shows discontinuities, the user suspects that the toroid mesh itself has discontinuities, i.e. its constituent triangles are not correctly connected, and so sets out to visualise these discontinuities. The user constructs a network, shown in figure 2.4 with the *superQuadric*, *slice3dVWR* and *vtkFeatureEdges* DeVIDE modules. The *vtkFeatureEdges* module is a simple VTK object wrapping, explained in section 2.6.6, that is able to extract several kinds of edges. In this case, the user configures the module to extract all boundary edges. A boundary edge is an edge that is used by a single polygon. The boundary edges will indicate discontinuities in the super-quadric tessellation.

---

<sup>1</sup><http://www.boost.org/libs/python/doc/>



**Figure 2.4:** DeVIDE network that extracts different types of edges from a super-quadric mesh. The mesh and the extracted edges are visualised together in the *slice3dVWR* module.

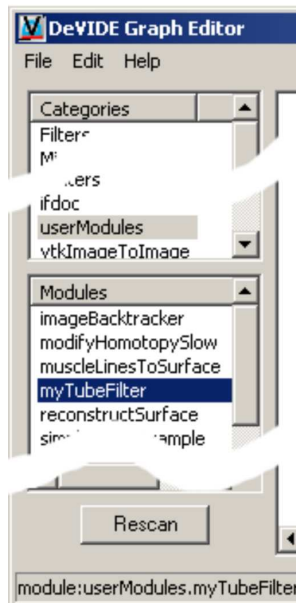
Visualising the extracted boundary edges and the super-quadric with the *slice3dVWR*, the user notes that the boundary edges are shown as very thin lines which are difficult to pick out. It is decided that it would be useful to create a module that builds polygonal tubes from thin lines.

With one of the packaged module behaviours in the Module Library (see section 2.6.6 for more about module behaviours) the user constructs a module that is based on a *vtkTubeFilter*. Depending on the behaviour that is used, this module specification can be written in less than ten lines of Python. The module specification has to conform to the Module API as documented in section 2.6.2.

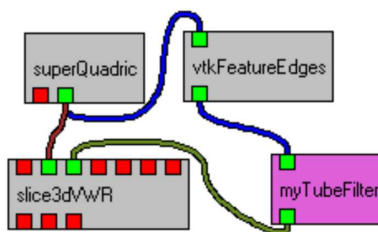
The user creates the Python module specification, in this case called *myTubeFilter.py*, in the DeVIDE user modules directory. Because this is a new module, the Module Manager has to be instructed to scan for and make available new modules. This is done by clicking on the Graph Editor's *Rescan* button, shown in figure 2.5. At this stage, the new module appears in the *userModules* category as shown in the figure. This can now be dragged and dropped onto the canvas to the right of the module palette. If there are no serious errors in the module code, a glyph representing the newly instantiated module will be created. If there are errors, the system will inform the user of these and also of precisely where in the code they occurred. The user can subsequently remedy these and simply retry instantiating the module, without recompiling or restarting any other modules or parts of the software.

Once successfully instantiated, the module's glyph can be connected with the existing glyphs. In this case, the output of *vtkFeatureEdges* is connected with the input of *myTubeFilter* and the output of *myTubeFilter* is connected to the visualisation module. Figure 2.6 shows the resultant network and figure 2.7 the resultant visualisation.

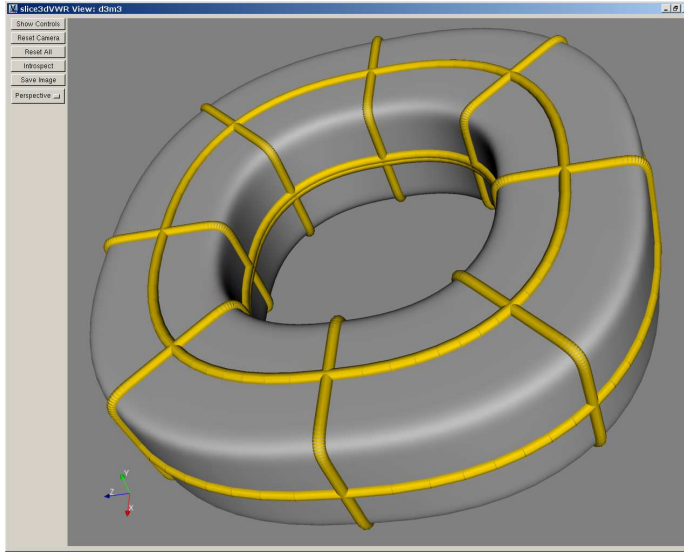
It is entirely possible that the module developer wishes to refine the module after seeing the results after the first successful integration and network execution. Experiments with different parameters or even a different implementation can be performed by making use of any of the pervasive interaction elements documented in section 2.5.5. For example, the user could make use of the introspection facilities to determine better default parameter values for



**Figure 2.5:** The Graph Editor's module palette (shortened) showing the module categories at the top, with the *userModules* category selected. Modules in the currently selected categories are shown in the bottom list. A module can belong to more than one category. Multiple categories can be selected.



**Figure 2.6:** The DeVIDE network shown in figure 2.4 with the *myTubeFilter* module integrated.



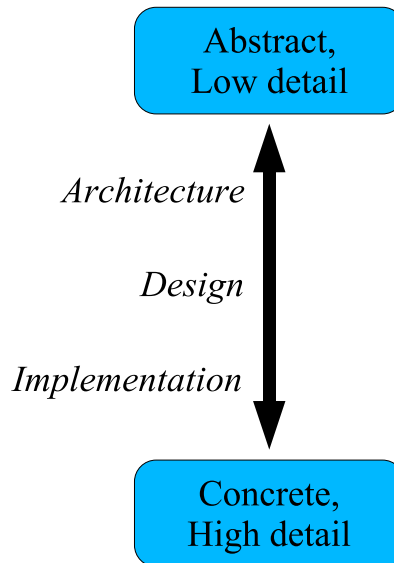
**Figure 2.7:** A visualisation that makes use of the newly-created *myTubeFilter* module to emphasise discontinuities in the sample mesh.

the new module. The visualisation and all other data and parameters throughout the system change immediately to reflect the modifications. Subsequently, the user could modify the module specification on disc and simply re-instantiate the module. At every instantiation, the latest version of the module code is automatically loaded. During this process of refinement and repeated module re-instantiation, the rest of the system and all modules remain up and running. This facilitates the rapid development and refinement of new modules.

In general, creating new DeVIDE modules is a relatively simple process. The system makes as few as possible demands on the module developer, but extra effort is rewarded with more functionality. In addition, the pervasively dynamic nature of DeVIDE that enables rapid refinement and experimentation with new modules helps to distinguish it from many similar problem solving environments.

## 2.5 Architecture

In the literature, the terms *architecture*, *design* and *implementation* are often used in order to describe software systems. However, the definitions of these terms are not straight-forward. Broadly speaking, these three terms refer to different levels of abstraction and detail, with *implementation* referring to the highest level of detail and concreteness and *architecture* referring to the lowest level of detail and the highest level of abstraction [22]. Figure 2.8 illustrates this continuum. In literature, no consistent distinction can be found.



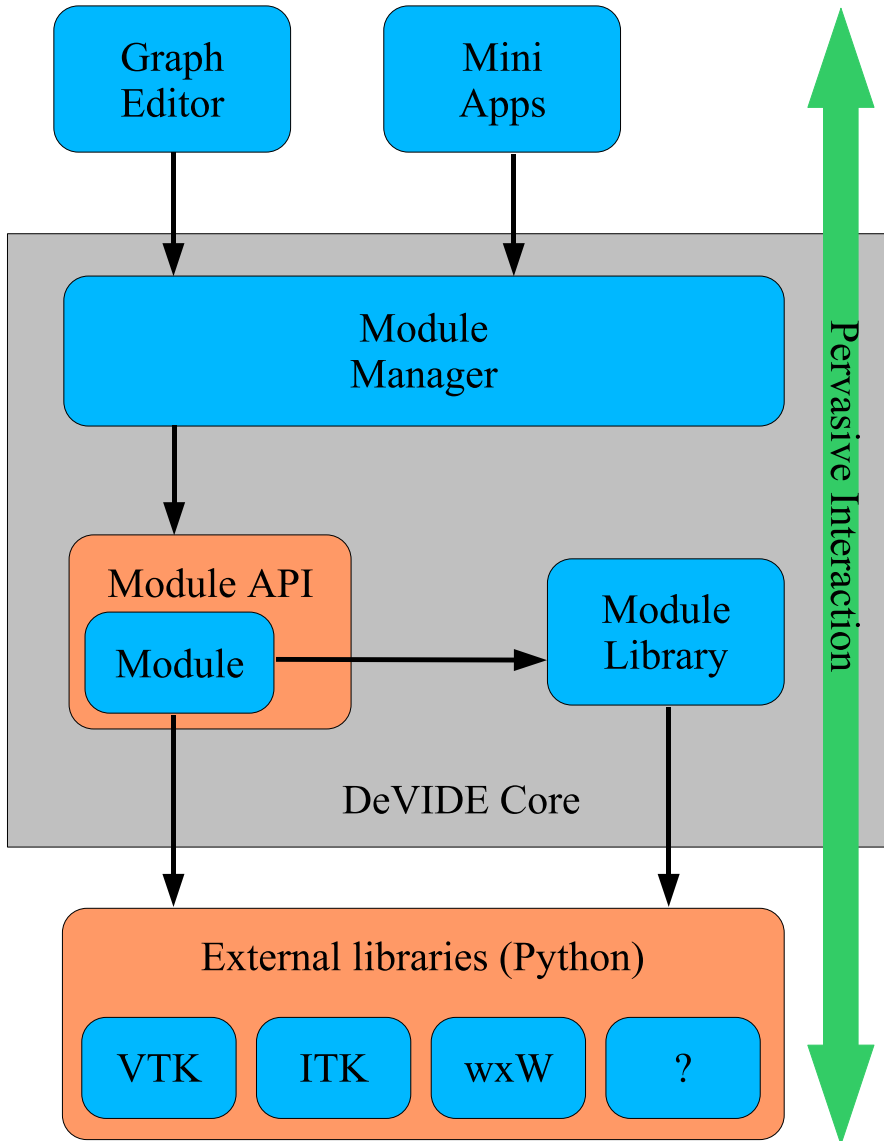
**Figure 2.8:** The architecture, design and implementation continuum: from concreteness and high-detail to abstraction and low-detail.

For the sake of this exposition, we will make use of a definition similar to that of Perry and Wolf [63]. Their definitions of architecture, design and implementation are as follows:

- *architecture* is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design;
- *design* is concerned with the modularisation and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements; and
- *implementation* is concerned with the representation, in a programming language, of the algorithms and data types that satisfy the design, architecture, and requirements.

We will document design and implementation together in section 2.6. In this section, we will be looking at the high-level structure of DeVIDE and focus on the main architectural elements, in other words, the building blocks of the DeVIDE framework.

Figure 2.9 shows a diagram of the high-level DeVIDE architecture. We will now proceed to document each of the illustrated components.



**Figure 2.9:** Diagram of the high-level DeVIDE architecture. The Module Manager is the main external interface.



### 2.5.1 The Module and Module API

The DeVIDE module is a central concept in the whole architecture. All algorithmic functionality is implemented and made available to the framework in the form of modules. Module functionality is exposed to the rest of the system via the Module API, which consists of a set of standard functions and behaviours. See section 2.6.2 for implementation details on this API.

Due to the module API, each module has zero or more input ports and zero or more output ports. Two modules can be connected by associating one or more of the first module's output ports to one or more of the second module's input ports. When this is done, data produced by the first module is available for further processing by the second module. By connecting more modules together in this way, large networks of functionality can be built.

Any object of functionality can be integrated as long as it can be made to satisfy the module API. Due to the choice of Python as module specification language (see section 2.6.1), a very large class of functional objects can be integrated with relatively little effort.

It is important to note that writing a DeVIDE module refers almost exclusively to creating the DeVIDE-specific specification that will make the encapsulated functionality available via the module API. This means that no modifications have to be made to the encapsulated implementation itself.

### 2.5.2 Module Library

The module library is a repository of functions and behaviours that can be used by module writers to satisfy the module API and integrate functionality with the DeVIDE framework. For example, by making use of a behaviour in the module library, a module writer can encapsulate the functionality of a VTK class with four or five lines of code. Section 2.6.6 has more detail on this.

### 2.5.3 Module Manager

All modules and module-related resources are controlled by the module manager. All access to modules and module resources has to take place via the module manager. This component has the following responsibilities:

**Cataloguing** The module manager searches permanent storage for valid modules and makes a list of such modules available to its clients, for example the Graph Editor. Since new modules can be added and modules can be removed at run-time, the module manager can be asked to update its list at any stage.

**Instantiation, reloading and destruction** Any number of instances of a specific module can be created. A module has to be instantiated before it can be used to process data. Each instantiation of a module makes use of the most current module specification, in other words, if changes are made to the specification, the next instance created by the module manager will include these changes.

If a module is no longer required, the module manager can be instructed to destroy it. During destruction, the module manager ensures that all producer modules, i.e. modules supplying data to the module that is about to be destroyed, and consumer modules, i.e. modules using data produced by the module that is about to be destroyed, are disconnected.

**Connection and disconnection** All module connections and disconnections are taken care of by the module manager. For example, if the Graph Editor is instructed by the user to connect two modules, the Graph Editor will first send the request to the Module Manager. The Module Manager will then perform the actual connection and report on its success to the Graph Editor. The nature of the actual connection itself is up to the participating modules. The Module Manager requests the modules, via the module API, to connect the relevant ports: if they do not protest, the connection is registered by the Module Manager as existing.

**Execution** Although DeVIDE makes use of a demand-driven execution model, i.e. a processing step only takes place if an explicit request for its result is made, it is possible to request explicitly the execution of a part of the network. These requests are controlled and acted upon by the module manager. DeVIDE's execution model is discussed in 2.6.3.

**Error and progress handling** During module instantiation, destruction, connection, disconnection or execution, errors may occur. The module manager performs error handling in this case. Error handling mostly entails notifying the user with an explanation of the error condition and its origin.

During module execution, explicitly or implicitly requested, the module can report on its progress to the module manager. The module manager will report on this progress to the user via a centralised interface.

**Serialisation and deserialisation** Serialisation refers to the action of representing a data-structure in a location-independent way so that it can be transported, for example over a network connection, or stored elsewhere, for example on disc. Deserialisation refers to the reverse operation, where the serialised representation is converted back to a data-structure.

As part of the module API, each module can be queried for internal state information. The module manager makes use of this state information as well as its own internal data structures with regard to the network topology, i.e. how modules are connected to form networks, in order to serialise the complete state of a whole network or a section of a whole network. In its serialised form, network state can be copied or saved to disc. Such a serialised network state can be deserialised, i.e. it can be used to recreate the original network, for example when loading a saved network from disc. A network can not be serialised during execution.

### 2.5.4 External Libraries

All modules have access to a set of external libraries, either directly or via calls in the module library. At the moment, the standard set of external libraries includes the Visualization Toolkit, or VTK [73], the Insight Segmentation and Registration Toolkit, or ITK [32], and wxWidgets/wxPython<sup>2</sup>. VTK is an extensive library of C++ classes focused on visualisation and related data processing. ITK is a library of C++ classes encapsulating a wide spectrum of image processing functionality. wxPython refers to the Python wrappings of wxWidgets, a library for creating cross-platform graphical user interfaces.

More libraries can be added: the only requirement is that the new functionality must be callable from Python.

### 2.5.5 Pervasive Interaction

A very important aspect of facilitating and speeding up the prototyping process involves enabling the user to interact with the system in as many ways as possible and as flexibly as possible. DeVIDE supports interaction at all levels, from interaction via graphical user interfaces created by the module developer to writing, loading and executing code at runtime. This interaction continuum can be roughly divided into four sections, ordered from high-level to low-level: interaction via graphical user interfaces, construction of functional networks, introspection and finally programming. Introspection will shortly be explained. Programming in this case refers to the action of writing program code, for example in the form of DeVIDE modules or in the form of a DeVIDE snippet<sup>3</sup>.

In addition to the interaction continuum, one could also imagine a continuum of user types. According to the nomenclature defined by Parsons *et al.* [62], ordered from low-level to high-level, we have: component developer, application developer and end user. We have omitted the framework developer as this is not pertinent to the current discussion. The component developer refers to a user that develops components that function as part of a software framework. In our case, this is the module developer. The application builder builds applications by making use of the developed components. In the context of DeVIDE, this refers to a user that constructs functional networks by configuring and connecting DeVIDE modules. The end user makes use of the constructed networks or applications.

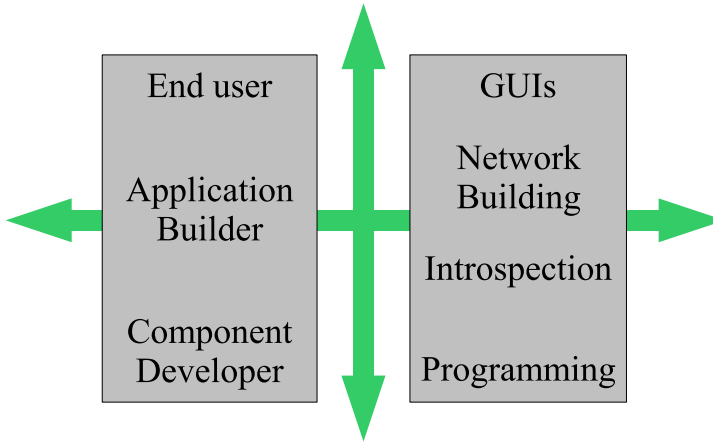
As illustrated in figure 2.10, the user type continuum runs in parallel to the interaction continuum: in general the end user is interacting via purpose-built graphical user interfaces, whilst the application builder is constructing networks and the component developer is making use of the introspection and programming facilities. However, no clear partitions need to be made between the various user and interaction types. For example, although the network building interaction modality is utilised primarily by the application builder, it is not inconceivable that the end user might change the topology of a pre-built functional network.

The continua show that interaction is possible at all levels of abstraction, i.e. there is an element of depth, but interaction at all points in the system, i.e. breadth, is also crucial for

---

<sup>2</sup><http://www.wxwidgets.org/> and <http://www.wxpython.org/>.

<sup>3</sup>See section 2.6.4 for more on snippets.



**Figure 2.10:** The user type and interaction continua. Interaction can take place at any level of abstraction. Each interaction type is more suitable to a specific user type, but no clear partitions can be made. The introspection interaction modality makes interaction possible at all points.

facilitating the prototyping process. In DeVIDE this interaction breadth is represented by the introspection functionality.

Introspection usually refers to the action of contemplating one’s own thoughts, of self-examination. In the computer science world, it is also known as *reflection* and refers to the action or ability of querying code objects at run-time about their characteristics, and also of making changes to these executing code objects.

In DeVIDE, *any* object or variable anywhere in the system can be examined and modified by the user at run-time. The effects of any changes are immediately visible. For example, if the output of a module is not as desired, a dialogue with the module internals can be initiated. During this dialogue, even unexposed variables can be changed and new variables and executable code can be added. This iterative process continues until the output is as desired.

The design and implementation issues of the system’s introspection functionality are discussed in section 2.6.4.

## 2.5.6 Graph Editor

The Graph Editor is an important client application of the Module Manager. This is probably the most flexible graphical user interface to the underlying ideas of creating, configuring and connecting modules to form functional networks. Modules are represented by glyphs, or boxes, with graphical representations of input and output ports. These ports can be graphically connected to the ports of other modules with wires.

In general, this is the interface that most users of the software will spend the most time working with. It is important to note, however, that this is one possible way of interfacing with the Module Manager and the underlying modules and networks. There is a very clear separation between the Graph Editor and the Module Manager. Other client applications are conceivable.

### 2.5.7 Mini Apps

In some situations, a simpler interface than the Graph Editor is required, for example if some subset of functionality is to be made available to a user. In this case, a mini application can be written that makes use of all DeVIDE functionality and all modules, but with a far simpler interface and method of operation. Such a mini application interfaces with the DeVIDE framework exclusively via the Module Manager.

## 2.6 Design and Implementation

In this section, we document more specific aspects of the DeVIDE framework. Where architecture is concerned with the architectural elements, or building blocks, of a system, design and implementation are concerned with *how* exactly these elements perform their functions.

### 2.6.1 Python as primary implementation language

Python is a high-level, interpreted, dynamically typed<sup>4</sup> and object-oriented programming language [92]. An important characteristic of the DeVIDE platform is that *all* high-level logic has been implemented in Python whereas processor-intensive logic has been mostly implemented in compiled C++ and Fortran, or other performance-oriented languages. This allows us to profit from the higher programmer productivity and software reuse offered by modern scripting languages [59, 67] but, partly because of the well-known rule that 20% of any program is responsible for 80% of the execution time [6], almost completely negates the normally associated performance impact.

Practically put, the code encapsulated by a DeVIDE module is generally compiled C++ whereas the module specification itself is Python. In addition, all framework code, e.g. the Module Manager, the Module Library and the Graph Editor, has been implemented in Python.

This could be seen as an instance of the dual language framework concept described by Parsons *et al.* [62] and by Telea [82]. In the mentioned publications, the dual language concept is presented as a way of creating frameworks where modules, or components, can be dynamically loaded and utilised in an already running framework. This is in contrast with compiled frameworks, where the utilised modules and network topology have to be determined at compile time. The dual language approach is criticised for two reasons:

---

<sup>4</sup>Dynamically typed implies that variable types are checked at run-time, and not at compile-time, as is the case with statically typed languages.

1. Two languages have to be learned in the cases when a user of the framework acts as both module developer (“component developer”) and network builder (“application designer”).
2. The mapping of abstractions from the compiled language to the interpreted language are complex or impossible, as often the compiled language sports more flexible concepts than the interpreted language. This entails that mapping from low-level to high-level enforces certain restrictions on the resultant functionality.

In the case of an advanced interpreted and syntactically clear language such as Python, the permanently increased programmer productivity far outstrips the relatively short time that has to be spent learning the language. This overcomes the first point of criticism.

Parsons and Telea describe a setup where the interpreted language is used solely for the run-time flexibility its interpreted nature offers, i.e. in order to be able to instantiate modules and build networks at run-time. In our approach, the interpreted language is used wherever possible, whereas the use of the compiled language is limited to processor-intensive tasks, such as image processing algorithm implementations. This change in philosophy entails that relatively few abstractions have to be mapped from the compiled language to the interpreted language. In addition, we believe that the abstractions made possible by Python are definitely more powerful than those offered by C++ or Fortran. These factors negate the second point of criticism.

The use of Python as specification and main implementation language of DeVIDE yields several advantages:

**No compile-link cycle:** Because Python is an interpreted language, no compile-link cycle is required for the Python components of the system. This has significant impact on development time, as the effects of changes to program code are immediately visible.

**Real dynamic loading:** We are able to load and *reload* arbitrary code modules at run-time. This is standard Python functionality and requires no system-dependent hacks.

**Run-time introspection:** Any existing program entity, code or data, can be inspected and modified at run-time. This makes for interesting development and debugging possibilities and is used extensively as part of the DeVIDE “pervasive interaction” architectural component.

**Robust code:** Garbage collection, bounds checking, advanced exception handling and other similar conveniences, help to eliminate basic errors which occur far more easily in low-level languages.

**Programmer productivity:** Python language features and the availability of an extensive collection of third party libraries speed up the development of new functionality.

**Listing 2.1:** Example of extended DeVIDE module specification with method bodies removed

---

```

class skeletonModule:
2     def __init__(self, moduleManager):
     def close(self):
4     def getInputDescriptions(self):
     def setInput(self, idx, inputStream):
6     def getOutputDescriptions(self):
     def getOutput(self, idx):
8     def getConfig(self):
     def setConfig(self, config):
10    def logicToConfig(self):
     def configToLogic(self):
12    def viewToConfig(self):
     def configToView(self):
14    def executeModule(self):
     def view(self, parentWindow=None):

```

---

## 2.6.2 Module Application Program Interface

The DeVIDE module application program interface, or API, refers to the standard interface via which the Module Manager communicates with and manages DeVIDE modules. This interface consists of a number of expected methods and calling conventions.

Creating the Python code that implements these methods according to the API can be seen as a form of module specification. However, the module specification is not limited to simple specifications of the required methods, but can be as extensive as the module programmer wishes and Python allows.

In the following subsections, the methods and calling conventions comprising the API are discussed in more detail. If a module is created from scratch, all these methods must at the very least be declared with an empty message body. In Python, this can be done by using the **pass** keyword. However, in general a module developer will make use of one or more of the ready-made module behaviours, or mixins, discussed in section 2.6.6 and will consequently only have to implement those methods that differ from the used mixins.

Throughout the following exposition, we will refer to the example in listing 2.1.

### Basic module behaviour

Each DeVIDE module is represented by a Python class, or object specification. Each module is contained in a separate file on disc, but each file may contain other class declarations besides that of the module class.

Line 1 of listing 2.1 shows a standard class declaration used for starting a module specification.

### Initialisation and finalisation

`__init__()` is the constructor of the module class and will be called when the Module Manager instantiates an object of this class, i.e. when this DeVIDE module is created. A binding to the Module Manager instance is passed to the constructor. The module developer can optionally store a copy of this binding in the module instance if the module has to access functionality made available by the Module Manager.

`close()` is called by the Module Manager when the module is destroyed. This method should perform all the necessary cleanup actions, such as deleting any bindings to objects instantiated or bound by the module and shutting down any user interface functionality. Before `close` is called, the Module Manager will automatically disconnect all other modules. Note that `close()` is not the class destructor.

### Input and output

The input and output ports are specified by the four methods in lines 4 to 7 of listing 2.1. `getInputDescriptions()` and `getOutputDescriptions()` return lists with descriptions for the input and output ports respectively. Each element in a list is used purely as documentation for that port, i.e. the user, via for example the Graph Editor, can get module developer-provided information describing the nature of a specific port. These descriptions are optional. However, the length of the list is crucial, as the system uses that to determine the number of input or output ports.

The method `getOutput(self, idx)` returns the data associated with the `idx`'th output port. `setInput(self, idx, inputStream)` associates the data bound to `inputStream` to the `idx`'th input. These methods are called by the system when two modules are connected. A module can prevent a connection by raising an exception within one of these two methods.

Note that the module API has been designed so that no type information is explicitly specified. Python's run-time typing system and interpreted nature enable much more advanced type handling at connection time. See section 2.6.5 for more details on this.

### User interfacing

Each module has the option of communicating with the user via a non-modal graphical user interface, i.e. it is allowed to create and show its interface, consisting of one or more windows, but it is not allowed to restrict all application input to its own windows. A module may show its interface immediately after it is created. This is the norm for viewer-oriented modules. In general, however, a module does not show its interface until it is asked by the Module Manager to do so. Such a request will be made via the `view()` method.

User interfaces can be created by the module developer in one or more of the following ways:

- by making use of a graphical interface designing tool such as wxGlade<sup>5</sup>,

---

<sup>5</sup><http://wxglade.sourceforge.net/>.



- programmatically, i.e. by making the various library calls to build up a complete user interface, or
- by making use of automatic interface creators in the module library.

Section 2.6.6 documents the automatic interface creators in more detail. The only constraint is that these interfaces integrate with the main wxPython event loop. In practice, this means that most interfaces are built with wxPython widgets, although it is conceivable that other interfaces can be used, as long as they periodically call into the wxPython event loop.

The module developer can also choose not to create a purpose-built interface and to have the user make use of the built-in pervasive interaction abilities of the DeVIDE platform. As is the case with many other aspects of the platform, no constraints are put on the module developer by for example requiring that a certain rigid interface API is adhered to. The developer is free to create interfaces of arbitrary complexity. At the same time, infrastructure is provided in those cases where the module developer wants to get a user interface up and running with minimal effort.

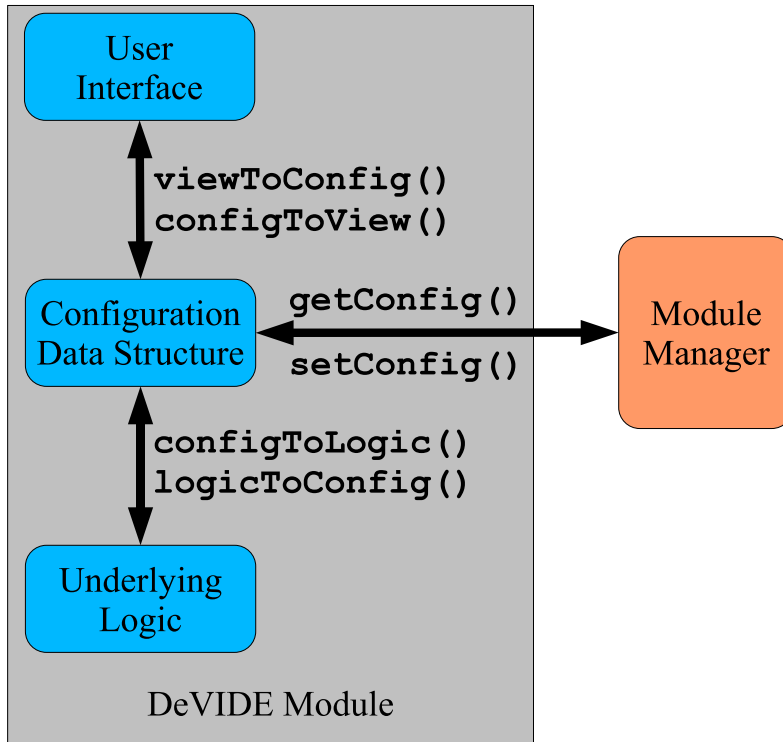
The module API also assists with the transfer of information between the module user interface, module state and the underlying code. The following section has more detail on this.

### Configuration handling

A very important aspect of the Module Manager's functionality is its ability to serialise and deserialise network state (see section 2.5.2). By default it has access to all topological information, but it also needs some mechanism to query and to set the configuration, or state information, of each module. The configuration calls in lines 8 to 13 facilitate this process for both the Module Manager and the module developer.

`getConfig()` should return an arbitrary Python data-structure that encapsulates the state of the module. This data-structure is defined as being such that if it is passed to `setConfig()`, the module will be returned to exactly the same state it was when the `getConfig()` was called that returned that data structure. The module developer is free to determine the completeness of this state, e.g. she may decide that only some variable values have to be restored. Henceforth, we will refer to this as the module *configuration data structure*. These are the only two methods that are required if the module developer wishes the module to support serialisation. If the user does not implement this support, either directly or via a module behaviour that includes this support, only the module's connections with other modules can be serialised. In many cases, this is sufficient.

However, many of the packaged higher-level module behaviours that are user-interface related expect that the module conforms to a certain model of configuration information storage and flow. This model entails that a module maintains an explicit internal configuration data structure and that the system has some say in when the state information is passed from the module user interface to its configuration data structure, and from the configuration data structure to the underlying algorithm. The diagram in figure 2.11 illustrates this model of



**Figure 2.11:** A model of the flow of module state information between the user interface, the module configuration data structure and the underlying logic, i.e. the actual implementation of the module algorithm. Communication with the Module Manager is also shown. The module API methods that drive the flow are also indicated.

state information flow. The methods that are expected are listed in lines 10 to 13 of listing 2.1 and are also shown in the diagram.

For example, all modules that make use of the `createECASButtons()` Module Library call, or any modules that make use of a module behaviour that makes use of this call, have the following buttons as part of their user-interfaces: *Execute*, *Close*, *Apply*, *Sync*, *Help*. Figure 2.14 shows an example of such an interface with the mentioned buttons at the bottom. When the user clicks on *Sync*, or “Synchronise dialogue with configuration of underlying system”, the system calls `logicToConfig()`, followed by `configToView()`, thus causing the state information to be extracted from the underlying logic and reflected in the user-interface. When the user clicks on *Apply*, or “Modify configuration of underlying system as specified by this user-interface”, the state information is transferred from the user-interface to the underlying algorithm by the system via calls to the module’s `viewToConfig()` and `configToLogic()` methods. Directly afterwards, these two methods are called again, but in reverse sequence, so that the state information percolates back up from the underlying al-

gorithm to the user interface. This is so that the algorithm also has the opportunity to reject invalid state information and to notify the user.

If any module that implements this configuration flow model is serialised by the system, its serialised state will reflect the module state that was last applied by the user.

This type of consistency and expected behaviour facilitate the creation of new modules as well as the interaction of the user with the software. The module developer is free to implement these methods without maintaining an explicit instance of an internal configuration data structure, as long as the methods put the module in the expected post-invocation state.

## Execution

The `executeModule()` method, shown in line 14 of listing 2.1, is called by the system when the user indicates that an explicit execution of a module is required. However, this method is not used as part of the demand-driven execution model of the DeVIDE software. As explained in section 2.6.3, the software requires that each module output object supports an `Update()` call that guarantees that all previous processing is up to date. Due to this, in most cases the `executeModule()` method simply calls `Update()` on the module outputs.

### 2.6.3 Execution model

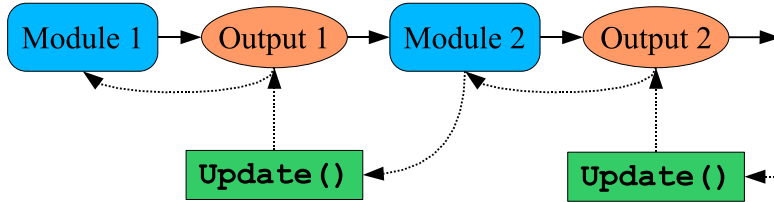
The DeVIDE framework is based on a demand-driven execution model. In other words, network execution only takes place when output data is requested by the user, for example through a viewer or a writer module.

In general, it is required that all module output objects have an `Update()` method that can be invoked by connected consumer modules. Calling this method should guarantee that the output object is up to date, implying that all required input data is also up to date and that the module in question has performed all relevant processing. If all modules follow this convention, update requests propagate backwards to the required source modules and processing subsequently propagates forwards through the relevant parts of the network.

We illustrate the working of this execution model with the example shown in figure 2.12.

When a consumer module requires up-to-date output data, it calls the `Update()` of the encapsulating output object, in this example called *Output 2*. The output object then notifies its owner module, *Module 2*, which in turn requests its input objects (the output objects of its producer module, i.e. *Output 1* in the diagram) to update before it can perform its own processing. This update request propagates all the way back through any given network until it reaches the source module or modules, in this case *Module 1*. The backwards propagation is shown by the dashed arrows. The actual processing then starts at the first module, or modules, and propagates forwards through the network until the final required data object is up to date. This forward propagation is shown by the non-dashed arrows.

This simple convention, also followed by VTK [73] and ITK [32], facilitates transparent execution of arbitrarily complex network topologies. For instance, modules can have any number of inputs. If the convention is followed, module execution will be correctly scheduled



**Figure 2.12:** The DeVIDE demand-driven execution model. Due to the simple update convention, output data update requests propagate backwards through any given network until the source module is reached. Processing then propagates forwards until it reaches the final output object.

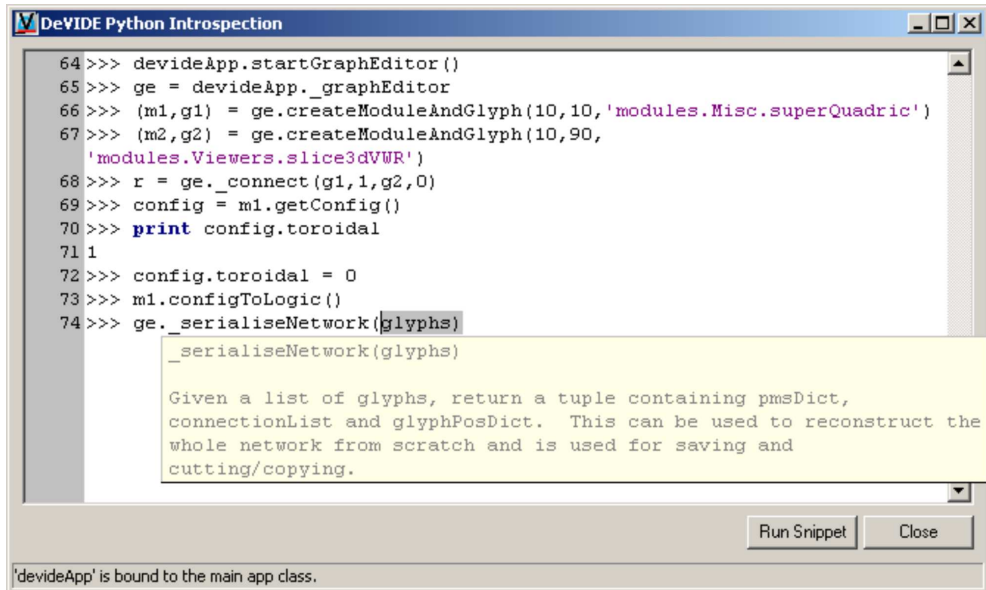
and only the necessary parts of the network will actually execute. Output objects usually implement update caching, i.e. if none of the required input objects have changed since the last update request, no processing has to be performed. In this way, network execution is kept to the minimum required to generate a valid output.

It is important to note that modules are not required to follow this convention, as long as a similar local convention is adhered to or as long as a module can guarantee that its output is *always* up to date. By a local convention, we mean that if a particular producer module can only be connected to a particular set of consumer modules, and there exists some agreement amongst these modules with regard to ensuring that the producer module's output data is up to date when any of the consumer modules require this, the execution model still holds. Similarly, certain modules have always-up-to-date output data, for example modules that handle user interaction and immediately copy the results of the interaction to their outputs. In this case, the update convention is not required, but correct network execution is still guaranteed.

## 2.6.4 Introspection

As explained in section 2.5.5, introspection allows one to examine and change any object at run-time. An object can be a simple variable, a more complex object containing variables and executable code, or executable code itself. Due to the use of Python as the primary development language, DeVIDE can offer different forms of introspection with which the user is able to examine and change any aspect of the system at run-time.

Figure 2.13 shows a sample session with the central introspection interface. In this example, the user, after making sure that the Graph Editor has been started up, creates two modules in the Graph Editor: one to generate a super-quadric isosurface and another to visualise that surface. Subsequently the user changes the parameters of the super-quadric function in order to generate a new surface. This is done via the configuration data structure interface as explained in section 2.6.2. Finally, the user investigates ways to serialise the network, i.e. to convert the network topology and module state information into a data structure that can be replicated or re-used. Remember that the actual state of the code is updated continuously as changes are made, so these changes will for instance be immediately reflected in any running



```

64 >>> devideApp.startGraphEditor()
65 >>> ge = devideApp._graphEditor
66 >>> (m1,g1) = ge.createModuleAndGlyph(10,10,'modules.Misc.superQuadric')
67 >>> (m2,g2) = ge.createModuleAndGlyph(10,90,
        'modules.Viewers.slice3dVWR')
68 >>> r = ge._connect(g1,1,g2,0)
69 >>> config = m1.getConfig()
70 >>> print config.toroidal
71 1
72 >>> config.toroidal = 0
73 >>> m1.configToLogic()
74 >>> ge._serialiseNetwork(glyphs)
    
```

`_serialiseNetwork(glyphs)`  
 Given a list of glyphs, return a tuple containing pmsDict, connectionList and glyphPosDict. This can be used to reconstruct the whole network from scratch and is used for saving and cutting/copying.

Run Snippet    Close

'devideApp' is bound to the main app class.

Figure 2.13: A sample introspection session with the main DeVIDE Python Introspection window.

visualisations or algorithm outputs. This kind of interaction and real-time feedback make for effective prototyping.

During such a session, the system attempts to assist the dialogue by showing help in the form of tool-tips wherever applicable. This help is extracted from the running code. An example of this is shown in figure 2.13. Possible completions of statements are also suggested by querying the member methods and variables of objects. An example is shown in figure 2.15.

At many points in the DeVIDE user interface there are possibilities to start an introspection session, for example via the module graphical user interface if the module developer has made use of the packaged introspection behaviours discussed in section 2.6.6. However, via the main introspection interface *all* internal objects are always available. Modules can also be interactively “marked” and named by the user via the Graph Editor. A list of marked modules can be requested from the Module Manager via any of the introspection interfaces.

## Snippets

Snippets are an interesting hybrid of introspection and batch programming that consist of useful sequences of introspection commands and can be stored on disc as DeVIDE “snippets”. These snippets can be executed via the “Run Snippet” button shown in figure 2.13, also available in other introspection interfaces. When such a snippet is executed, the instructions are

sequentially fed to the introspection interface from which the “Run Snippet” function was invoked.

Snippets are often used in cases where a module would be less appropriate. For example, DeVIDE ships with snippets for calculating the surface area of all selected 3D objects in a marked 3D object viewer module, or for making animations of deforming surfaces. One could also see the DeVIDE snippet as a kind of plugin that has access to *all* system internals and does not have to make use of a fixed plugin API.

### 2.6.5 Data types

As explained in section 2.6.2, the module API has been designed so that no type information has to be specified for module inputs or outputs. Instead of explicit type checking, module developers are free to implement “functional” checking. By making use of introspection functionality, a data type can be queried at runtime to determine whether it is suitable for the processing that the module wants to perform. Even if a data object is of a different class or type than the data object that a module usually operates on, if it has the required methods or shows the required behaviour, it counts as valid input data. This type of checking is more flexible than traditional rigid type checking.

Even if a developer decides not to implement any kind of checking, the Python run-time exception system will ensure that, in the case of an unusable data object, the DeVIDE error reporting system can trap and report the error to the user.

### 2.6.6 Module Library

The DeVIDE Module Library is an extensive collection of utility program code that can be used by module developers to create new modules. This library consists of module behaviours and discrete methods that can be used directly or indirectly, i.e. via a module behaviour, by modules. The discrete calls include functionality for logging and explicit error reporting<sup>6</sup>, module user interface creation and module interaction through introspection. In general, these are made available through the various module behaviours.

A “behaviour” refers to a module template that can be used by the module developer. For example, if a developer wishes to create a module for importing a new type of dataset, there is an existing “data reader” behaviour that can be used, thus minimising programming effort. Most module behaviours are implemented as mixin classes. A mixin class, as the name indicates, can be “mixed in” with, or added to, a module specification. It should not be seen as a base class. A module developer can make use of any number of mixin classes to add each mixin’s encapsulated functionality to the module that is being created. Some more complex module behaviours are implemented as real base classes and are built up out of a number of mixin classes. As explained in section 2.6.2, the complete list of module methods only have to be implemented if a module is written from scratch, i.e. without making use of any packaged module behaviours. Usually, however, a module makes use of packaged module

---

<sup>6</sup>Most errors are automatically detected by the system and reported to the user.

behaviours and only a small number of the module API methods have to be implemented. Some of the often-used behaviours are discussed in more detail in this section.

### Module introspection

Introspection functionality can be added to modules by making direct or indirect use of the module introspection mixin. Note that modules that do *not* make use of this mixin can also be introspected via any of the other interfaces offered by the system.

The user interface widgets that can be seen in the second to last widget row in figure 2.14 are usually automatically created and configured by the system when requested by the module developer. The introspection functionality that they invoke is encapsulated by the introspection mixin.

The drop-down box on the left of the second to last row contains a list of objects, supplied by the module developer, that can be examined by the user. In general, these are objects contained by the current module, but often include an instance of the module itself. If any of these objects are selected and they are VTK objects, a special interface such as the one in figure 2.15 is automatically created and shown. At run-time, the VTK object is queried and a list of its callable methods is created. On the grounds of this list, a graphical user interface is created. Each of the tabs contains a different set of controls.

In this example, the user is experimenting with a *vtkXMLImageDataReader* VTK object. Note that the interface suggests possible member variables and functions based on the user's input. If the selected object is not a VTK object, a simpler interface, similar to the one shown in figure 2.13, is shown. It is of course possible to create more specialised interface creation logic for any other type of object. If the "Pipeline" button is clicked and there are VTK objects in the object list, a representation of the linkages between the underlying VTK objects is shown. Objects can be selected from this graphical representation and interacted with. Our VTK object and pipeline parsing implementation is based on an existing library<sup>7</sup>.

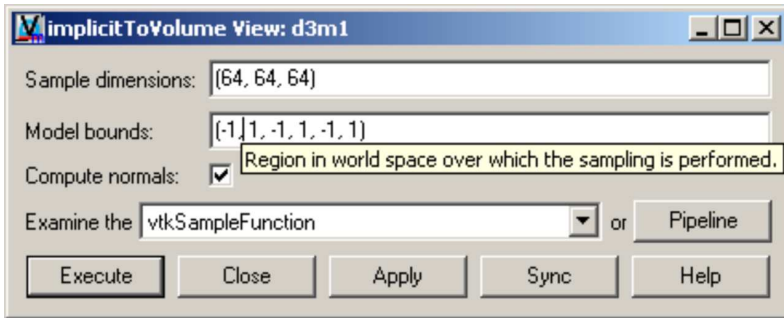
### Scripted interface creation

As explained in section 2.6.2, the user has a choice of creating a user interface with a graphical interface design tool, creating the interface programmatically or by making use of an automated interface creator. The scripted interface creation mixin is an example of such an interface creator.

When making use of this mixin, the module developer can specify a number of variable names and variable and type descriptions, and the system will automatically create an applicable user interface window. The complete interface shown in figure 2.14 is an example of such a window. All logic to perform input sanity checking is supplied by the system. For example, the value being edited in the "Model bounds" text box has been specified to be a six-element tuple of floats. When the user attempts to apply a new value, it will be automatically validated by the system and corrected if it is not a valid six-element tuple of floats. The validated value is added to the module configuration data structure.

---

<sup>7</sup><http://mayavi.sourceforge.net/vtkPipeline/>.



**Figure 2.14:** Screen-shot of an example dialog created by the scripted configuration module `mixin`. An example tool-tip is shown as the mouse pointer hovers over the “Model bounds” text input widget. The standard introspection widgets can be seen on the second row from the bottom. The standard module action buttons are on the bottom row.

The module developer has to specify how the validated values are transferred between the configuration data structure and the underlying logic, i.e. the `configToLogic()` and `logicToConfig()` methods have to be specified as explained in section 2.6.2. In general, these are straight-forward, but the module developer has the option of performing more specialised processing.

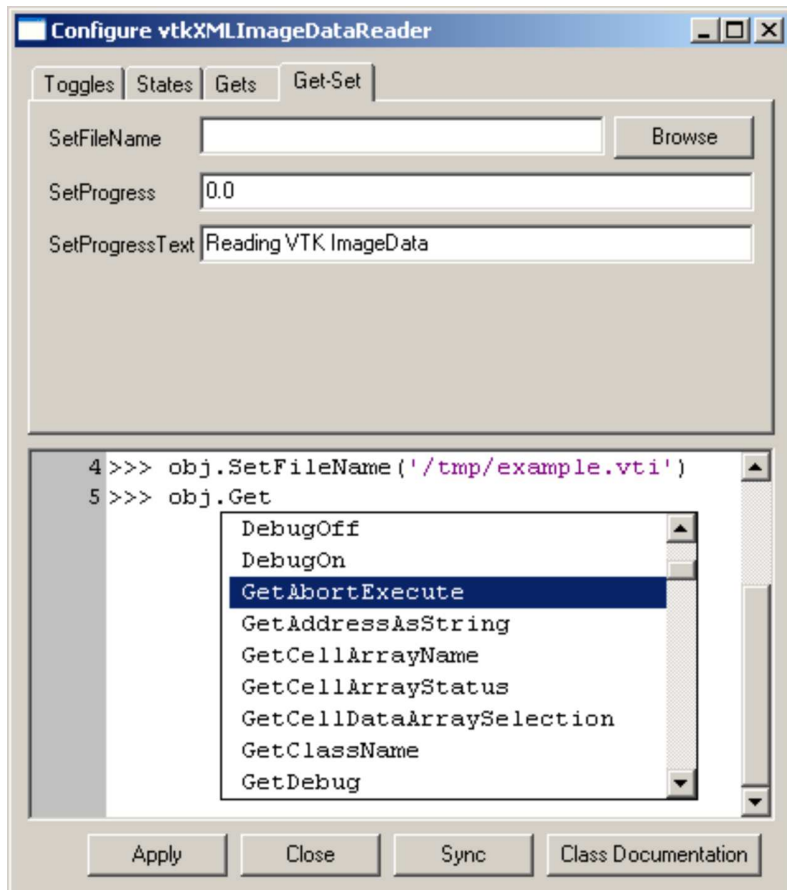
### Simple VTK object wrapping

Modules in DeVIDE are in general more high-level and are built up out of two or more VTK, ITK or other objects, but it is often useful to be able to wrap and experiment with discrete VTK objects. For this purpose, a simple VTK object wrapping base class is available. Listing 2.2 shows an example of the use of this class to wrap the `vtkTubeFilter` VTK class.

The resultant module has a complete user interface that is automatically generated by analysing the VTK object, is able to display user help by querying the VTK object documentation and is completely serialisable. By this last observation, we mean that any changes made by the user via the generated interface are saved and restored when the containing network is saved or restored. In other words, it is a rather complete DeVIDE module.

A relatively simple script was created to wrap a few hundred VTK objects in this way. The remaining objects weren’t wrapped because determining their parameters, such as number of inputs and outputs, could not be automated. These could also be wrapped with relatively little manual intervention. DeVIDE can thus also be used as a graphical front-end for experimenting with various VTK objects. Complete networks of VTK objects can be configured and stored for later use. This makes for a useful experimental and learning tool.





**Figure 2.15:** Screen-shot of sample introspection session with a VTK object after it has been selected from the object drop-down list on the second to last row of the dialogue box in figure 2.14.

Listing 2.2: Example of a complete module wrapping of a simple VTK object

---

```

# class generated by DeVIDE::createDeVIDEModuleFromVTKObject
2 from moduleMixins import simpleVTKClassModuleBase
import vtk
4
class vtkTubeFilter(simpleVTKClassModuleBase):
6     def __init__(self, moduleManager):
            simpleVTKClassModuleBase.__init__(
8                 self, moduleManager,
                vtk.vtkTubeFilter(), 'Processing.',
10                ('vtkPolyData',), ('vtkPolyData',),
                replaceDoc=True,
12                inputFunctions=None, outputFunctions=None)

```

---

## 2.7 The prototyping process in DeVIDE

The algorithm prototyping process can be seen as an optimisation problem. The error that is being minimised is the difference between the hypothetical perfect solution and the current solution. The independent variable axes span a space that consists conceptually of three possibly overlapping subspaces: structure subspace, program code subspace and parameter subspace. Structure refers to pre-packaged processing that is required, for example simple pre-processing and analysis of input data. Program code refers to the actual lines of program code that are being written and modified. Parameters are the various algorithm-specific variables that can be changed to modify the conditions of that algorithm.

Finding a suitable algorithm implies converging simultaneously through the three subspaces until a local minimum error is found that satisfies the design requirements. In other words, various permutations of structure, program code and parameters are tested until the resultant algorithm is considered to be a good solution.

In order to converge efficiently to a suitable solution via prototyping, it should be possible to search both rapidly and in a convergent fashion through the solution space described above. DeVIDE attempts to help the user satisfy both of these requirements.

The first requirement, rapid searching, is facilitated by the pervasive interaction architectural element of the system: structure space is traversed by reconfiguring the topology of the current functional module network and by instantiating pre-packaged modules. Program code space is traversed by creating and modifying modules and snippets and via the introspection functionality. Parameter space is searched via interaction with the various module-specific graphical user interfaces and also with the introspection functionality.

The second requirement, i.e. that the search through solution space should be convergent, is facilitated by the fact that the system gives immediate feedback on the smallest step made in structure, program code or parameter space. For example, if a single parameter is changed

via introspection, the resultant output data or visualisation is immediately available at any point in the network that is being used. This helps the user to iterate incrementally towards an improved permutation, i.e. towards a lower error in the solution space.

Very importantly, all iterative steps in the structure, program code and parameter spaces can be taken at run-time, without having to restart the platform or compile, link and load new program code. For example, after creating a new module or making changes to an existing module, only that module has to be re-instantiated for the changes to take effect. The rest of the system and the complete functional module network remains at its previous state. This factor speeds up prototyping to a great extent and distinguishes DeVIDE from many similar problem solving environments.

This optimisation problem view of prototyping is an alternative but compatible formulation to the iterative software development model view proposed in section 2.1.

## 2.8 Discussion

In this chapter we presented the Delft Visualisation and Image processing Development Environment, or DeVIDE, a software platform for the rapid creation, testing and application of modular image processing and visualisation algorithm implementations. DeVIDE focuses on facilitating and speeding up the prototyping aspects of visualisation and image processing related research. It does this by enabling a rapid convergence to a suitable problem solution through the structure, program code and parameter subspaces.

DeVIDE has the following features:

**Pervasive Interaction** As explained in section 2.5.5, it is possible to interact, at run-time, with the platform at any level and at all points. This characteristic differentiates the platform from many similar systems.

**Easy integration** DeVIDE attempts to make it as easy as possible to create new algorithm modules. Several characteristics aid in this goal:

1. There are only two requirements for code that is to be implemented in DeVIDE: it should support data-flow-based processing and it should be callable from Python. In most cases, it is straight-forward to satisfy these requirements.
2. The module specification language is Python. The advantages of this language are discussed in section 2.6.1.
3. The module API easily scales upwards and downwards. In other words, a module developer is free to implement the complete module API but, in cases where not all infrastructure-related functionality is required, only has to implement the relevant parts. In addition, many of the module API calls scale in a similar way, i.e. the developer can choose different levels of implementation detail.
4. The pervasive interaction feature mentioned above also assists the module developer in the module creation process.

**Short code-test iterations** In DeVIDE, all explicit parameters can be changed. Only the required sections of the processing pipeline re-execute so that the user can immediately examine the new results. This is a feature available in most good visualisation and image processing packages.

**Modularity** In DeVIDE, all algorithmic code is integrated in the form of extremely strictly partitioned modules. The interface between module and the rest of the framework consists of less than ten methods. Isolating code in this way facilitates cooperation between module developers, as they don't have to deal with the internals of external code.

**Facilitate re-use** Encapsulating implementations in modules keeps code separate but allows cooperation due to the common module API. This allows researchers to share implementations in the form of modules and to reuse old implementations in new experiments. If the code were monolithic, this would be prohibitively difficult.

**Scalability** As more functionality is added over time, monolithic (i.e. non-modular) systems become more difficult to develop and use. Modular data-flow systems, such as DeVIDE, generally do not suffer from this problem, as functionality is added in the form of modules. The system kernel remains compact and maintainable. Modules remain relatively simple and strictly isolated from each other. The fact that the number of modules increases has no impact on manageability. In this way, complexity and functionality have been decoupled.

**Documentation and reproduction** In addition to the fact that complete networks can be saved and restored, human-readable representations of the networks and their parameters, such as the example shown in figure 2.3, can also be produced. This plays an important role in the documentation of complete algorithms and methods.

**Platform independence** All the software components of DeVIDE were selected or designed to be platform-independent. This means that DeVIDE, in theory, can run on all modern operating systems. DeVIDE development takes place both on Windows and Linux. Binaries for both these platforms are available.

DeVIDE, unlike many similar systems, attempts to keep the abstraction layer between the module developer and the underlying logic as thin and as simple as possible. This approach yields the greatest flexibility and performance. The flexibility manifests itself in that modules can be created with the least possible effort and that the module API is relatively unrestrictive. The performance is attributed to the fact that underlying logic does not have to be adapted to adhere to a DeVIDE specific data-model, but continues to make use of its native data-model. For example, connecting two VTK-based modules generally results in the underlying VTK objects being natively connected.

The trade-off for this flexibility and performance is of course robustness. With DeVIDE, we have made the design decision to give the module developer as much freedom as possible, as our main goal is to facilitate the prototyping and development of new algorithms. This

also means that it is entirely possible for underlying misbehaving module code, e.g. written in C++, to cause the platform to crash. In our experience, this is a worthwhile trade-off.

Chapter 3 presents applications that demonstrate how DeVIDE functions in the algorithm development process. In addition, all the work in chapter 4 was performed within the DeVIDE framework. In our experience and that of our collaborators, the system fulfils a valuable role in visualisation and image processing related research.

We plan to extend the system with more modules and especially more interaction possibilities. The next large architectural change is adapting the platform to allow for transparent distributed processing. After the planned changes, the module manager will be able to execute module code on remote machines, completely transparently to the user. The changes we have in mind increase the already present separation between the front-end and the processing components. As an added advantage, this will also increase the robustness of DeVIDE. Also, as the number of modules and module categories grows, it becomes increasingly difficult to find the correct module or sequence of modules to solve a particular problem, in spite of the available documentation. To address this, we plan to implement functionality whereby the system can recommend suitable modules or module sequences given a description of the problem or the input data. Initially, this would be similar in functionality to the SMARTLINK approach [81], but we plan to investigate other possibilities as well.



# CHAPTER 3

---

## DeVIDE Applications

---

In this chapter, we present three applications, or case studies, of DeVIDE. In the first, we present DeVIDE functionality and interfaces for the pre-operative planning of shoulder replacement operations. The second application involves the registration and reconstruction of two-dimensional microscopic sections of the female human placenta. Finally, we discuss how DeVIDE was used as part of a study on the displacement of the female human pelvic floor muscles.

### 3.1 Pre-operative planning for glenoid replacement

In the Netherlands, 1.5 million people suffer from chronic arthritis. Every year, this group grows with 100000 new patients<sup>1</sup>. Chronic arthritis can lead to irreversible joint damage and disability.

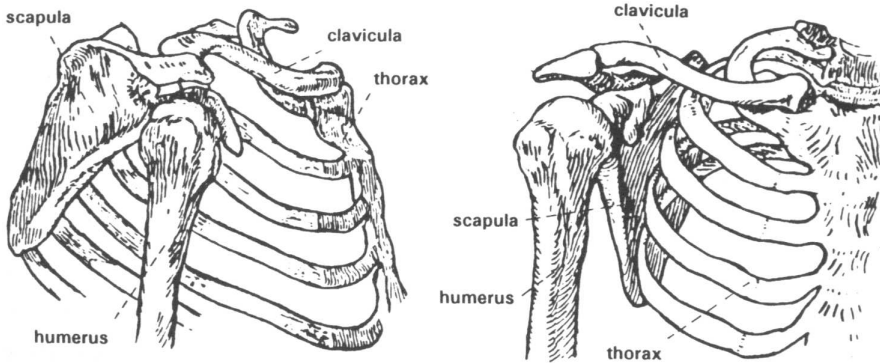
Joint replacement is indicated in cases where rheumatoid arthritis<sup>2</sup> or osteoarthritis<sup>3</sup>, the

---

<sup>1</sup>See the Reuma Stichting website at <http://www.reuma-stichting.nl/>.

<sup>2</sup>**Rheumatoid arthritis** is a severe inflammatory disease with chronic polyarthritis (inflammation in multiple joints) as its most characteristic clinical feature. This inflammation leads to joint pain, swelling and loss of mobility. Long-standing arthritis can lead to irreversible joint damage, joint space narrowing, erosions and deformities. Joint inflammation and joint destruction may result in serious disability.

<sup>3</sup>**Osteoarthritis**, also called osteoarthritis, is the most common form of arthritis. This disease is characterised by the degeneration of cartilage in the joints which is accompanied by the hypertrophy (overgrowth) of the underlying bone [38]. Osteoarthritis is associated with joint pain and stiffness, as well as loss of range of joint movement. This disease is one of the most frequent causes of physical disability among adults: 20% of all adults over the age of 55 suffer from osteoarthritis. 90% of adults over the age of 65 suffer from some form of osteoarthritis (source: <http://www.reuma-stichting.nl/>).



**Figure 3.1:** The skeletal structures of the shoulder. This figure is courtesy of the Delft Shoulder Group.

two most common forms of arthritis, has resulted in severe joint damage that in turn causes increased pain and reduced function. By 1994, 10000 shoulder replacements per year were being performed in the USA alone [99].

Hip and knee replacements are very successful procedures with regard to pain relief, post-operative joint functionality and durability. At ten years follow-up, the revision rate for cemented total hip prostheses is 7% and about 13% for uncemented total hip prostheses [48]. In contrast to this, shoulder replacement yields good results with regard to pain relief, but fares significantly worse with regard to post-operative joint functionality and especially replacement durability. Literature shows that after nine years, between 24% and 44% of shoulder glenoid components show radiological loosening [83], i.e. loosening that is visible on a radiograph.

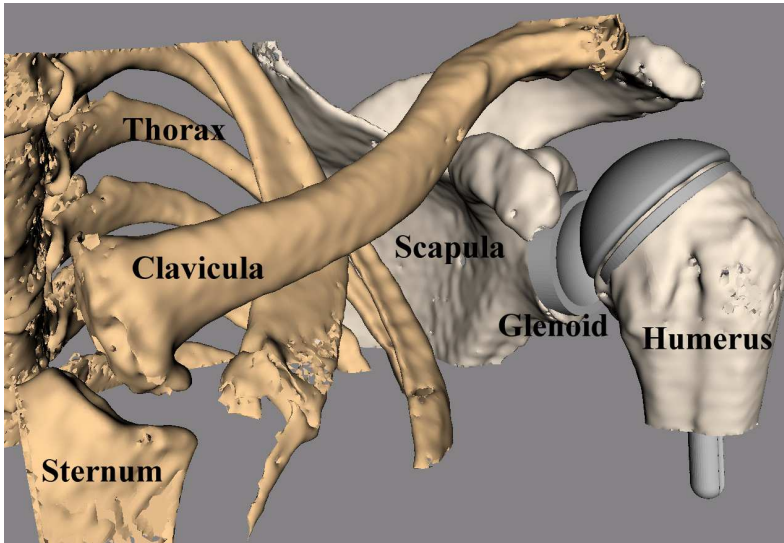
One of the reasons for this situation is the fact that the placement of shoulder prostheses is an extremely difficult procedure. The procedure is complicated by two factors:

- The shoulder joint is a far more complex mechanism than either the hip or the knee joints.
- A relatively small incision is made during the shoulder replacement operation. This small incision results in a very limited field of view that the surgeon has to contend with.

Figure 3.1 shows the skeletal structures in the human shoulder. The upper arm bone, or humerus, rotates against the glenoid, which is the very shallow cup-like part of the shoulder-blade, or scapula. The humerus is held in the shoulder joint by a collection of muscles and tendons called the rotator cuff. The scapula itself is non-rigidly attached to the thorax via the collar-bone, or clavicle, and is able to glide over the thorax.

This flexible construction yields an impressive range of motion for the shoulder joint, but as is almost always the case with increased complexity, is less robust than for instance the





**Figure 3.2:** CT-derived visualisation of the bony-structures in the human shoulder. Both the humeral head and glenoid prostheses have been virtually implanted, i.e. a total shoulder replacement.

hip joint. This complexity also makes shoulder replacement operations challenging tasks. Besides the fact that the precise functioning of this joint is not entirely known, the surgeon has to cope with extremely limited visibility during the replacement operation. These factors contribute to the lower long-term success-rate of shoulder replacement.

Shoulder replacement, or shoulder arthroplasty, entails that the glenoid and the humeral head are replaced with prostheses. Figure 3.2 shows a CT-derived visualisation of the skeletal structures in the human shoulder with implanted glenoid and humeral head prostheses.

Currently, the de facto standard for pre-operative planning for shoulder replacement is template-over-x-ray planning. This procedure entails that the orthopaedic surgeon manually overlays transparencies with various prosthesis sizes and types on an x-ray of the patient's shoulder. Consequently, the surgeon decides on a particular prosthesis type and size. This method of planning offers no patient-specific intra-operative guidance for prosthesis placement. Also keep in mind that this method by definition does not take into account the patient musculature.

More advanced systems for the pre-operative planning of hip and knee replacement exist, but, due to the complexity of shoulder replacements, no such tools exist for the shoulder.

The DIPEX (Development of Improved Prostheses for the upper EXtremities) project is a clinically-driven research effort by the Delft University of Technology, in cooperation with the Orthopaedics Department of the Leiden University Medical Centre, that attempts to improve the state of the art in shoulder replacement [88]. As an important facet of this effort, a novel approach to the surgical planning of shoulder replacement procedures is being

developed [89].

This approach is based on a flexible surgical planning software system and the concept of mechanical guidance devices, also called patient-specific templates. In short, the replacement operation is planned by making use of computer-based pre-operative planning software. The parameters of the virtually performed operation, such as the insertion position and angle of the glenoid component, are used to design a patient-specific mechanical guidance device that is used for intra-operative guidance.

The patient-specific mechanical guidance device is a jig, or template, that can be used by the surgeon during the operation to guide the use of another tool such as a drill or a pneumatic saw. In the case of a glenoid replacement, the template uniquely fits the patient's scapular glenoid and has a conduit for a drill, thus acting as a drill-guide. By making use of this template, the surgeon can ensure that the pre-operatively planned glenoid insertion position and orientation are realised. In this way, a cheap and robust coupling between the computer-based intra-operative planning and the operation itself is realised.

Experimental pre-operative planning functionality for glenoid replacement was implemented within the DeVIDE framework and consists of functionality to segment bony surfaces from CT-data of the patient, a CAD-based interface to manipulate 3D objects efficiently in a perspectively projected setting and functionality to generate a patient-specific glenoid template automatically. This section documents this application of the DeVIDE framework.

### 3.1.1 Segmentation

A pre-operative CT-scan of the affected shoulder is made. The resultant DICOM [56] data is loaded by DeVIDE and a segmentation of the bony structures is performed by thresholding and a 3D region growing starting from a set of user-selected seed points.

Since this method requires user assistance when the data of patients with rheumatoid arthritis (RA) or osteoarthritis (OA) is being processed and since most shoulder replacement patients have either RA or OA, we are also working on more advanced segmentation methods that can cope with the abnormalities of affected bony structures. See chapter 4 for more details on this.

After the segmentation, a surface model of the scapula is generated with the Marching Cubes algorithm [47]. Because this surface model is generated on the binary segmented data, it has the expected stair-step appearance. This is of course not the actual surface, but an artifact of the binary classification. In order to remedy this effect, we could have used a dilation of the binary segmentation as a mask of the original data and performed the surface extraction on the result. Instead we chose an elegant surface-smoothing approach that adapts analogue filtering techniques to polygonal meshes [79, 80]. This technique is often more robust when objects are close together in the image that is to be segmented.

The extracted surface can be visually checked for accuracy by interactively slicing through the unprocessed volume data, orthogonally or obliquely, and checking only the 2D contour intersection of the model and the slice. This is standard DeVIDE functionality and can be done for any number of simultaneous objects, slices and slice directions.

In our case, the accuracy of the scapular surface, especially on the glenoid itself, is of utmost importance. This is due to the fact that the patient-specific template is designed according to this surface and should fit the patient glenoid perfectly during the actual replacement procedure. In section 3.1.3 we discuss some of the complications of this fact.

### 3.1.2 Planning

Docking two 3D objects using only a perspective or an orthogonal projection on a conventional screen is by nature quite difficult. In our case, we need very precise control over the position and orientation of a prosthesis in order to implant it virtually into the surface model of the scapula.

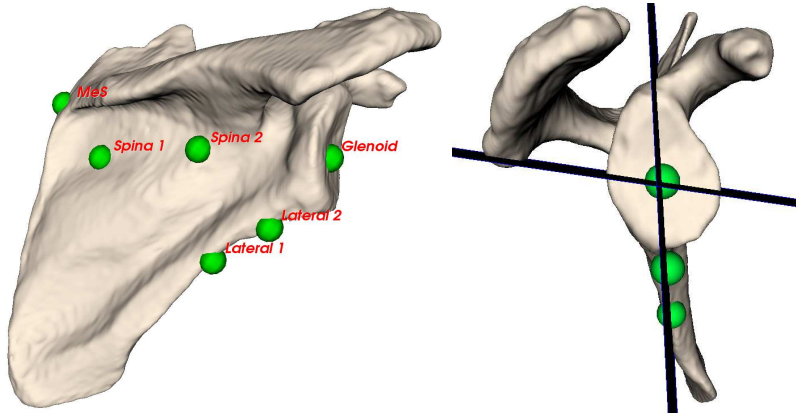
In order to solve this problem, we made use of some basic CAD interaction techniques combined with the practical techniques used by our orthopaedic colleagues during shoulder replacement surgery<sup>4</sup>. During a conventionally planned shoulder operation, the orthopaedic surgeon initially tries to align the prosthesis with the intersection between two imaginary planes. The first plane passes through the centre of the glenoid and the most lateral edge of the scapula. The second plane is almost orthogonal to the first, parallel to the spina and passes through the centre of the glenoid. Experience has shown that this will help to achieve a post-operative centre of gleno-humeral rotation that is close to, or coincides with, a healthy centre of rotation. This result plays a very important role in the success of the replacement [44].

DeVIDE integrates a range of geometrical constraint-based object manipulation possibilities based on points, axes and planes. With this functionality, the double plane insertion sketched above can easily be constructed and used as a first estimate for glenoid component placement. In order to do this, the surgeon will start by selecting three points: one in the centre of the glenoid and two on the lateral edge of the scapula. These points are shown on the left image in figure 3.3 and are marked with respectively “Glenoid”, “Lateral 1” and “Lateral 2”. The point selection logic makes it easy to select points on surfaces and will also keep these latched to the surface when they are moved. A scapula lateral edge slice can now be created by making use of these three points as a plane definition. The system will warn if the three selected points do not uniquely define a plane. See figures 3.3 and 3.4 for an example of such a plane intersecting the scapula. It can also happen that the plane defined by these three points does not have the desired orientation. In our experience, substituting one of the lateral edge points with the point where the spina meets the medial edge of the scapula, labeled “MeS” on the left image of figure 3.3, solves this problem.

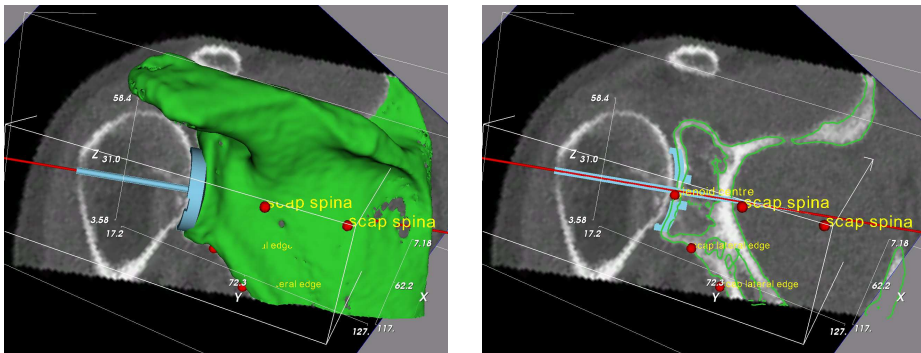
By repeating this process, but instead making use of the point on the centre of the glenoid and two points on the posterior side of the scapula approximately equidistant from the spina, a second plane can be defined. These two extra points are labeled “Spina 1” and “Spina 2” in the left image of figure 3.3. The intersection of these two planes constitute a very good first estimation of the glenoid insertion axis. This plane intersection is shown on the right of figure 3.3.

---

<sup>4</sup>Source: Prof. P.M. Rozing, head of the orthopaedics department at the Leiden University Medical Hospital, The Netherlands.



**Figure 3.3:** On the left, a scapula is shown with the points that are required to place the two anatomical planes. On the right, the same scapula is shown with the resultant anatomical planes intersecting to form an insertion axis for the glenoid prosthesis. See colour figure C.1.



**Figure 3.4:** Part of the glenoid component planning functionality. On the left the complete models are shown and on the right only their contours. The contour view enables the user to judge whether the prosthesis has a good fit and is reminiscent of the conventional template-on-x-ray planning. The slice is anatomically oriented but can be moved to check all parts of the glenoid component. See colour figure C.2.

The constraint system can now automatically align the prosthesis axis with the intersection of the two planes and optionally constrain all prosthesis motion to the plane intersection. The object with the horizontal pin on the left of figure 3.4 represents the glenoid prosthesis. Logically the constraint system can also work with a single plane: in this case the prosthesis can be aligned with the plane, i.e. be embedded in it, and optionally all prosthesis motion can be constrained to the plane. This is the approach that has been chosen in figure 3.4.

It is important to note that the constraint system works for arbitrary sets of points, lines and planes. The operating surgeon does not have to make use of the two-plane approach, but is free to choose any other constraint-based method. A system and interface is provided for defining new local coordinate systems by making use of existing geometry. These local coordinate systems can subsequently be used to define new geometry that can be used in conjunction with the local coordinate systems to manipulate and constrain 3D objects in a perspective or an orthogonal projection setting. This functionality has proven to be very useful.

During the final prosthesis manipulation, objects can be hidden and object contouring activated, so that the object intersection curves with the anatomical planes can be seen overlaid on the CT-data slices. This is shown on the right of figure 3.4. In this mode, the volume slice can still be moved to and fro. This enables the surgeon to judge the fit of the prosthesis with regard to the bone volume that is visible in the CT-scan. This view is more familiar to the surgeons, as it is strongly reminiscent of the template-over-x-ray planning method.

Once the surgeon is happy with the virtual placement of the glenoid component, a patient-specific template can be automatically generated by the planning software. In our case, this is called the glenoid drill guide.

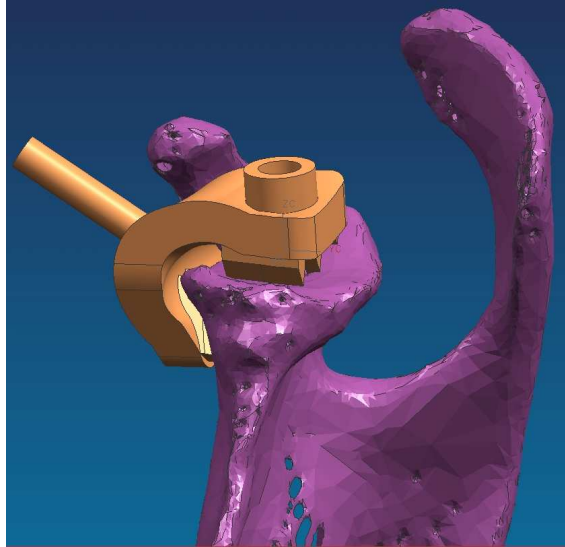
### 3.1.3 The Glenoid Drill Guide

The glenoid drill guide is a crucial link between the pre-operative planning phase and what happens in the operating room. During pre-operative planning, the size and type of prosthesis can be determined and this knowledge can easily be applied during the actual replacement. Glenoid component position and orientation, however, require some extra ingenuity.

In order to realise this planning in the operating room in a robust and efficient fashion, colleagues are working on applying principles for the design of pedicle screw drill guides [24] to the design of a patient-specific glenoid drill guide. Figure 3.5 shows the latest design, at the time of this writing, of such a glenoid drill guide.

Initial validation experiments have been performed on cadaver scapulae to test the design and functioning of the drill-guide [89]. The results were not entirely positive, as the cartilage covering the glenoid has a significant impact on the orientation of the resultant implant. Recall that the drill guide is designed based on the bony surface of the glenoid and not the cartilage covering it.

In spite of these results, we believe that the design has great potential. Currently, we are looking at ways of improving the performance of the drill guide. At least two kinds of solutions are conceivable:

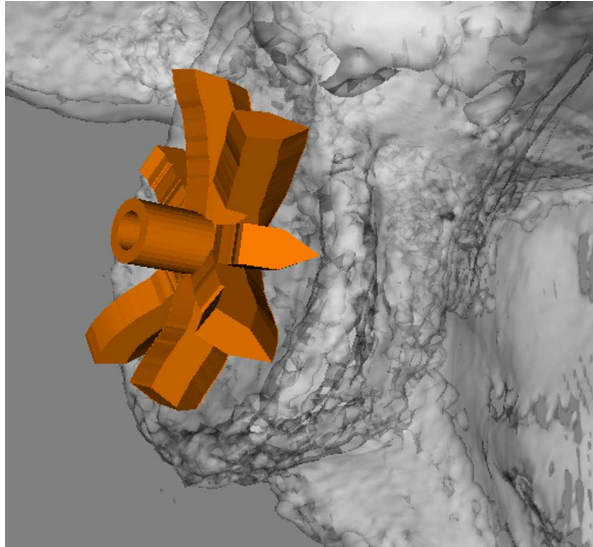


**Figure 3.5:** The latest design of the glenoid drill guide visualised on a surface model of a scapula. Illustration courtesy of Liesbet Goossens and Bart de Schouwer, Katholieke Universiteit Leuven, Belgium.

- The segmentation algorithms have to be adapted so that they are able to segment cartilage as well as bony structures. This poses two new problems, in that the segmentation of affected cartilage is even more challenging than the segmentation of affected bone and it is impossible at this moment to determine from a CT-scan which cartilage will be solid enough to support the glenoid drill guide and which will not. In chapter 4 we present an approach that solves the problem of segmenting affected bone, but the cartilage segmentation problem is still open. The use of MRI data could play an important role in solving this problem.
- The drill guide design has to be adapted to compensate for the presence of cartilage on the glenoid. Once again, it's currently impossible to predict the actual quality and stability of the affected cartilage.

While the design is being refined, functionality has been added to DeVIDE for the automatic design of a glenoid mould based on the performed planning. The end result of the glenoid component placement planning is an insertion position and an insertion orientation. Based on these parameters, as well as the patient-specific scapular mesh, a mould is procedurally constructed by a DeVIDE module. For example, in figure 3.6, an earlier mould design is being automatically performed according to a planning and the patient-specific surface. The procedure for this design was defined as follows:

1. Create a cylinder with an inner diameter of 3mm, an outer diameter of 5mm and a



**Figure 3.6:** An earlier mould design that is in the process of being constructed by DeVIDE according to a set procedure, the patient-specific scapular surface and the final pre-operative planning parameters.

length of 10mm.

2. Align this cylinder with the prosthesis insertion axis and place it a fixed distance above the glenoid surface.
3. Construct three planes at  $60^\circ$  angles to each other that intersect at the insertion axis. Determine the intersections of these three planes with the outside glenoid surface of the scapula.
4. Extrude a triangle and an adjoining rectangle along the plane-scapula intersections, starting from the exterior of the cylinder and moving radially outwards to just before the edge of the glenoid. Determine when the glenoid edge is reached by monitoring the tangent of the plane-scapula intersection line. These extrusions will form radial knife-edges that will allow the mould to be accurately placed on the glenoid.
5. Construct a thin platform to further strengthen the radial knife edges.

For any new guidance device design, the implementation process can be automated and made patient-specific if it can be described procedurally as in the example above.

### 3.1.4 Future work

We plan to continue extending the pre-operative planning functionality in DeVIDE by extending it with bio-mechanical models that will assist the surgeon during the planning process. As soon as a suitable mechanical guidance device design is found, we will refine the automatic patient-specific design functionality. Also, we will extend the constraint-based object manipulation with more types of constraints and interaction possibilities that will make it more accessible to clinical users. Section 8.2 has more details on our planned future work.

## 3.2 Visualisation of chorionic villi and their vasculature

The chorion is the outermost membranous sac that encloses the embryo in the higher vertebrates. The chorionic villi are thin, finger-like structures that protrude from the chorion and extend through the uterine lining into the maternal blood supply. These chorionic villi form the placenta. Each villus contains a number of blood vessels that carry oxygen and nutrients back to the embryo.

During the development of the chorionic villous vasculature, primitive lumen-less cords mature into the villous blood vessels. In addition, during the maturation, the vessels move to the periphery of the villus that contains them, a process called margination. In humans, this development takes place during the first twelve weeks of pregnancy. It is important that this villous vascular development progresses normally, as deficient vascular development could possibly play a role in the etiology of miscarriages and intra-uterine growth retardation [46].

The goal of this study was to create a 3D reconstruction of histological sections made of a sampling of human chorionic tissue and to use 3D visualisation techniques to find instances of cords during maturation, i.e. the process of changing from a lumen-less cord into a blood vessel, and margination, i.e. moving to the periphery of a villus, in the reconstructed data. DeVIDE was used for the reconstruction and 3D visualisation.

### 3.2.1 Method

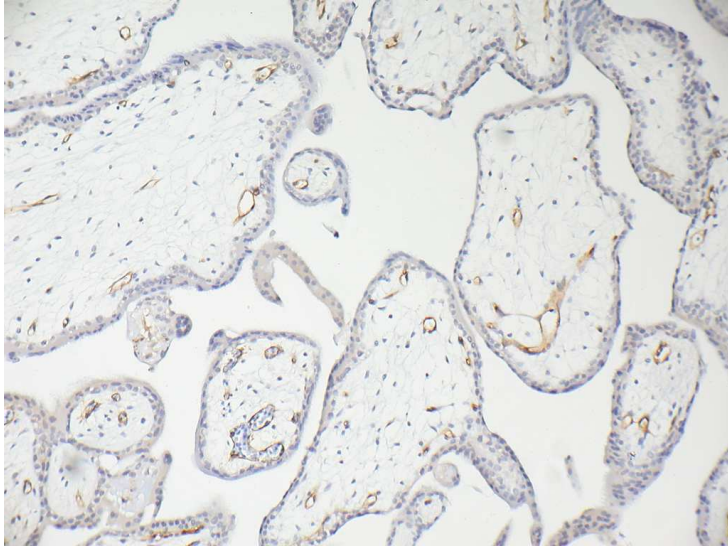
In a previous study [95], a small amount of chorionic tissue from chorionic villous sampling (CVS) was fixed in 2% buffered formalin and subsequently embedded in paraffin. After this,  $4\mu m$  serial sections were made and subsequently stained<sup>5</sup>. These microscopic sections were digitised. Figure 3.7 shows an example of such a digitised serial section.

For our study, three of the villi and the structures that they contain were manually delineated on all sections. Based on these delineations, we filled the structures with different grey levels, in order of containment with the most contained structures first. This filling ensured that the similarity metric we used for the subsequent registration process would yield usable results. Figure 3.8 shows the previous example after the delineation and filling operations. The outermost layer of the villus is called the syncytiotrophoblast, the layer just interior to that the cytotrophoblast. The bright structures in the interior are cords and vessels.

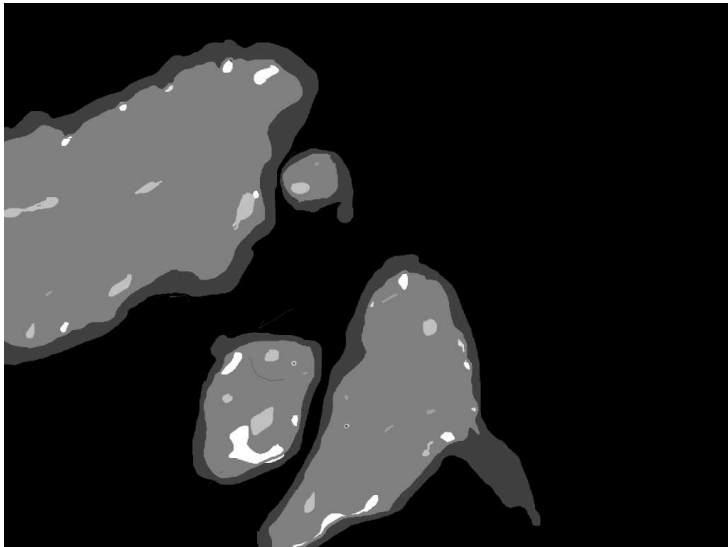
---

<sup>5</sup>Sections were stained with haematoxylin-eosin, CD31 immunohistochemistry and periodic acid-Schiff.





**Figure 3.7:** A digitised serial section of the chorionic tissue. The large structure covering almost half of the left image border is a villus. The circular structures in its interior are vessels.



**Figure 3.8:** The section shown in figure 3.7 after manual delineation and filling with grey levels.

The next step was to make use of registration algorithms to transform each filled slice until it was registered as well as possible on the previous slice, according to a selected similarity metric. In our case, we assumed a rigid transformation, i.e. only translation and rotation, and made use of the sum of the squared pixel intensity differences as the similarity metric. A gradient descent optimisation algorithm was used to search through the transform's parameter space based on the similarity metric. If we had not filled the data as explained above, the similarity function would show a few isolated spikes, instead of piecewise continuous behaviour, and the optimisation process would probably not have found a suitable transformation.

A number of DeVIDE modules, mostly making use of ITK [32] calls, were used in this process. Figure 3.9 shows the DeVIDE network that was used to perform the registration. All images were imported by the *imageStackRDR* module and processed by the *register2D* module. *register2D* allows the user to move through the stack of images at will and perform automatic image-to-image registration. In the case of a non-affine transformation, one would have to traverse the volume strictly in one direction. Registrations can also be fine-tuned manually, by modifying the automatically calculated rotation and translation. The image pair that is being worked on is visualised by alpha-blending the two images. In this way, real-time feedback is given on the quality of the registration. In addition, at each automatic registration step, the value of the similarity metric for the relevant image pair is also shown. This module combines ITK and VTK functionality.

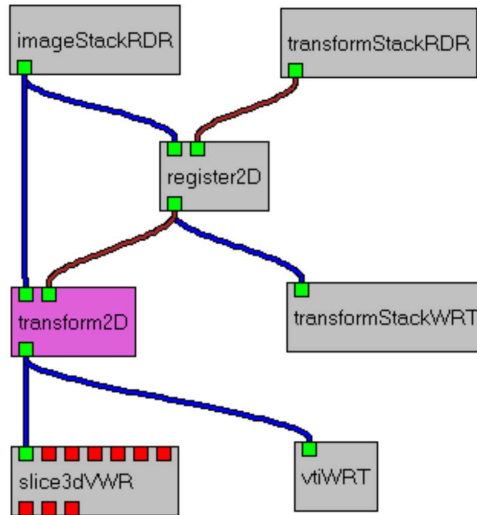
Figure 3.10 shows the interface of the *register2d* module. On the left, an unregistered image pair is visualised. Note that the registration parameters, rotation and the two translations, are all 0. By clicking on the “Register” button, the registration algorithm will attempt to find an optimal transformation. The automatic registration can be run as many times as is necessary. The user can fine-tune the current transformation at any point. On the right, the image pair has been successfully registered.

After all image pairs have been successfully registered, all transformations can be written to permanent storage with the *transformStackWRT* for later use. More importantly, the resultant transformations can be applied to the input images to reconstruct the complete 3-D data volume. This operation is performed by the *transform2D* module shown in figure 3.9. This module also combines VTK and ITK functionality. The resultant 3-D dataset can be processed and visualised with conventional 3-D techniques.

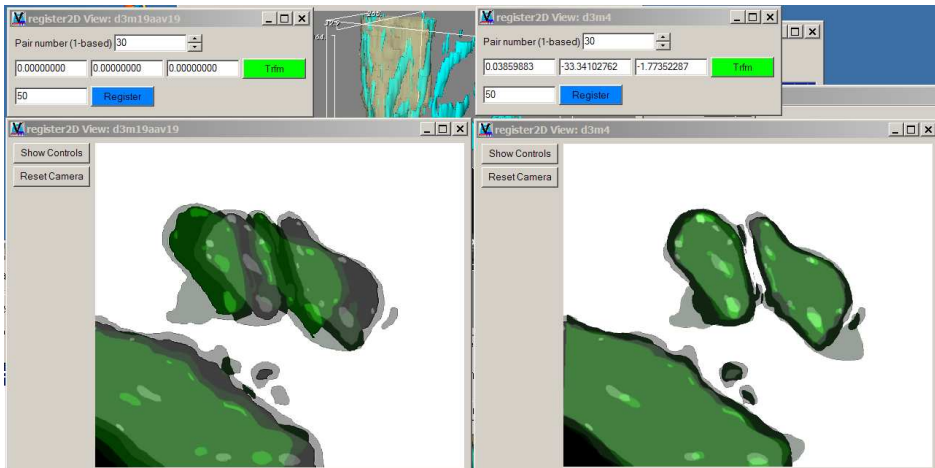
### 3.2.2 Results

The CVS dataset was registered and a volume was reconstructed. We chose to visualise the villi with transparent surfaces (for context) and the cords and vessels with opaque surfaces. Allowing the user to slice through the reconstructed volume at the same time aids understanding of the data. Figure 3.11 shows an example visualisation where a single villus and two of its vessels have been selected from the dataset by making use of 3-D region growing.

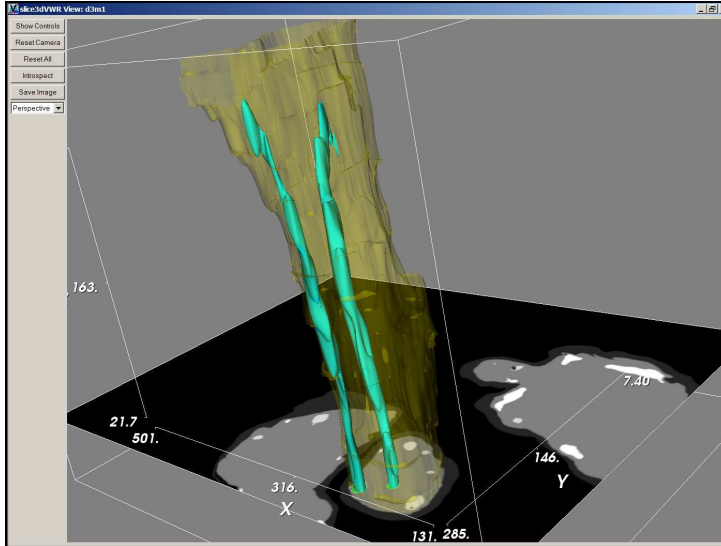
We were unable to find any instances of margination or of vessel maturation, even after a painstaking manual examination of all data. However, our clinical colleagues did find the reconstruction and subsequent visualisation illustrative. In that sense, the visualisation did



**Figure 3.9:** The DeVIDE modules and network that were used to perform the registration. *imageStackRDR* and *transformStackRDR* read collections of input images and image pair transforms respectively. *register2D* encapsulates the registration user interface. *transform2D* applies all derived transformations to all input images to generate a reconstructed 3D volume. *transformStackWRT* writes all derived transforms to disc. *vtiWRT* saves the resultant 3D volume to disc. *slice3dVWR* functions as the 3D visualisation user interface.



**Figure 3.10:** The user-interface of the *register2d* DeVIDE module. On the left an unregistered image pair is shown and on the right the same pair is shown after registration. See colour figure C.3.



**Figure 3.11:** An example of a visualisation of the reconstructed slices. A single villus and two of its vessels are visualised.

contribute towards insight in the data, although it was not successful in showing any instances of maturation or margination.

During this experiment, it once again became clear how important the acquisition process is for the success of the subsequent processing and visualisation. Making these  $4\mu\text{m}$  sections from the CVS specimens is extremely difficult, resulting in digitised images that often complicate the registration and reconstruction process. In addition, the manual delineation and registration processes were decoupled, i.e. there was no feedback from the registration process to the delineation process. We have come to the conclusion that coupling these processes, for instance in a unified user interface, would greatly improve the final results.

### 3.3 Pelvic floor displacement from MRI

DeVIDE was used in two studies on the female pelvic floor [34, 33] to assist in the visualisation and quantification of MRI, or Magnetic Resonance Imaging, data.

In the first study, the relation between displacement of the pelvic floor and the intra-abdominal pressure, or IAP, and pelvic muscle activation was examined in 10 healthy subjects and 10 patients that have undergone primary genital prolapse. Genital prolapse is a condition where the pelvic organs that are normally supported by the pelvic floor muscles, protrude outside the pelvic floor. MRI scans were made of all subjects during three conditions: maximal IAP, maximal contraction of the pelvic floor muscles and rest.



**Figure 3.12:** An example coronal slice from one of the MRI datasets. The arrows indicate the pelvic floor muscles. The surface shown in 3.14 describes the inner surface of these muscles in 3D.

The second study examined the loading effect of the weight of the internal organs on the pelvic floor in the erect and supine positions in 12 subjects. MRI scans were made of all subjects, also during the three conditions mentioned above, in both the erect and the supine positions. The loading effect was quantified as the displacement of the pelvic floor.

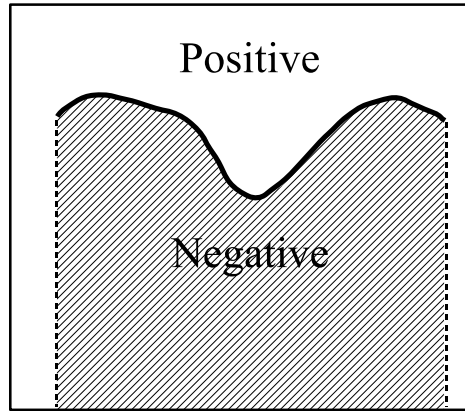
### 3.3.1 Surface derivation

In both cases, it was necessary to measure the displacement of the pelvic floor based on MRI data. Because MRI data is notoriously difficult to segment automatically and in order to simplify the quantification of the displacement, it was decided to derive a surface describing the inner border of the pelvic floor muscles.

Figure 3.12 shows a single coronal slice from one of the MRI datasets. The arrows indicate the pelvic floor muscles in the slice. On each coronal slice of each dataset, as the inner border of the pelvic floor was manually delineated.

The delineations were subsequently loaded into DeVIDE for further processing. The next step was to use the pelvic floor lines to derive a surface. The resolution in the coronal direction was between 5mm and 7mm, so special care was required during the interpolation. In addition, deriving a smooth, open and bounded surface from a volume is not straightforward. Iso-surface extraction techniques generate surfaces that are by definition closed, unless truncated by the boundaries of the dataset.

A DeVIDE module was created to perform all the necessary processing. We implemented

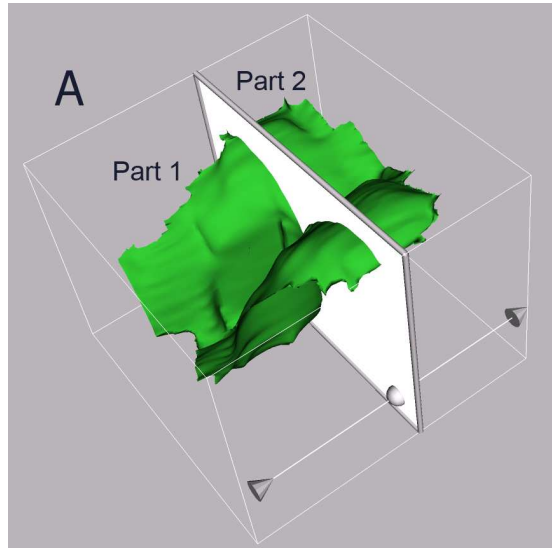


**Figure 3.13:** The signed distance field generated from the delineated curve. Above and to the sides the distance values are positive (i.e. the non-hatched areas)

the following steps:

1. Encode the manually delineated contours as 0-valued points on each slice.
2. Calculate the unsigned distance field with the lines as the origin.
3. In each slice, change the sign of the distance values *under* the delineated curve to negative, thus creating a signed distance field. We do this by performing a raster scan through the image and counting the number of times we intersect the contour. *Under* is of course entirely relative to an arbitrarily chosen origin, as long as this is consistent for all images in a dataset. Figure 3.13 shows an example of the signed distance field.
4. Extract an iso-surface with iso-value 0 from the complete 3-dimensional signed distance field. The extracted surface will have extraneous geometry throughout the volume wherever positive and negative values meet and there is no curve to define the boundary. For example, in figure 3.13 this will happen at the dashed lines. This geometry has to be clipped.
5. Create a new distance field by changing the sign of the regions to the left and to the right of the hashed block in figure 3.13. This new distance field can be seen as an implicit function and used to remove only the extraneous geometry.

The end result of this process is a smooth, open and bounded surface. An example of a surface that has been derived in this way is shown in figure 3.14. The use of a distance field results in a relatively smooth surface, in spite of the low resolution. Also, standard interpolation techniques can be used on the distance field volume before surface extraction.



**Figure 3.14:** Example surface describing inner border of pelvic floor. Image courtesy of Stepan Janda and created with DeVIDE.

### 3.3.2 Measuring the displacement

For the second study, MRI images were acquired of the subjects in both supine and erect positions. Since the relative position of the pelvic compartment is different for these two positions, the data from the erect position was transformed to the local coordinate system of the supine position. DeVIDE offers functionality for selecting a number of landmarks in both datasets and then deriving a rigid transformation that best maps the selected landmarks in the source dataset onto the selected landmarks in the target dataset. At least four bony landmarks were chosen in all datasets in the second study in order to perform this transformation. After the transformation, derived surfaces for the different positions could be directly compared.

In both studies, the pelvic floor displacement was measured by deriving the mean symmetric Hausdorff distance [3] between the different pelvic floor surfaces derived for each patient. First the surfaces were divided into two anatomical parts. In figure 3.14, these are denoted by respectively *Part1* and *Part2*. All surfaces were then exported from DeVIDE in a format that was readable by the Mesh software [3]. Mesh was used to calculate the Hausdorff distances.

### 3.3.3 Results

The first study showed no difference in pelvic floor displacement between the healthy subject and prolapse patient groups. In the first study, a significant difference in displacement was

found between scans made in the supine position and scans made in the erect position. This is due to the fact that the pelvic floor muscles are loaded with the weight of the internal organs in the erect position. From this work, it was concluded that patients should ideally be scanned in the erect position, as scanning in the supine position does not yield correct information about the pelvic floor during normal loading. Detailed results of these studies can be found in [34] and [33].

### 3.4 Conclusions

In this chapter, we presented three studies where DeVIDE was applied as a visualisation and data processing tool in clinically-driven research efforts. In the first, functionality for the pre-operative planning of shoulder replacements was shown. This is ongoing work. See sections 3.1.4 and 8.2 for more details on our plans. In the second application, microscopic serial sections of chorionic villi were registered and reconstructed to result in a three-dimensional dataset that could be used for further visualisation. The third study employed DeVIDE in the quantification of pelvic floor deformation.

In the first two applications, DeVIDE functioned mostly in its role as platform for the facilitation of algorithm development and testing. In other words, it was used primarily by someone in the module developer role. For example, all modules shown in figure 3.9, except for *slice3dVWR* and *vtiWRT*, were constructed during the chorionic villi study. In the last application, DeVIDE was used by a biomechanical researcher, i.e. an end-user, and served as an effective vehicle for the deployment of implemented techniques for use by a researcher who was not specialised in computer science or visualisation.

The pre-operative planning functionality relies heavily on generic functionality in the *slice3dVWR* DeVIDE module, so estimating development time for the planning application is difficult. In both the other two cases, developing and fine-tuning the required modules took a few hours. This is partly due to the flexible underlying software libraries, but to a large extent due to the pervasive interaction capabilities of DeVIDE. Writing module code is an incremental process where immediate feedback can be had for every small change. This helps the module developer to stay on the right track and prevents deviations from the design goals of the module that is being developed. Algorithm parameters are refined in the same way. All of this leads to rapid algorithm implementation.



---

## Segmentation of skeletal structures in the shoulder

---

### 4.1 Introduction

Patient-specific object representations of the skeletal structures in the shoulder<sup>1</sup> are important for visualisation, surgical simulation, quantification and modelling. The object representation of a skeletal structure, such as for example the scapula, or shoulder blade, describes the shape and position of that structure. Depending on different applications, different kinds of object representations are possible. A simple but high-level object representation of the humerus, or upper-arm bone, would be for example the centre of the humeral head, and the length and orientation of the stem.

In our case, these patient-specific object representations have to be derived from computerised tomography, or CT, images of the patient's shoulders. The object representations are to be used for visualisation, surgical simulation, measurement and finite element modelling. With these applications in mind, the most useful representation is an accurate description of the shapes, or surfaces, of the skeletal structures making up the shoulder joint.

The process by which these kinds of object representations are derived from CT data is called segmentation. Segmentation specifically refers to the process of finding boundaries that divide raw data into meaningful objects.

We often have to perform segmentations on CT data of patients with bone- and cartilage-affecting diseases. This, together with the fact that the shoulder joint has a complex geometry, results in a challenging segmentation problem.

In this chapter, we document our work on a new approach for deriving high-quality sur-

---

<sup>1</sup>See figures 3.1 and 3.2 for more details on the skeletal anatomy of the shoulder.

face meshes from CT images of patient shoulders. We start by defining more exactly the desired segmentation and resultant surface descriptions. We discuss factors complicating the segmentation process. After detailing related work, we explain our approach for the segmentation of the skeletal structures of the shoulder from CT data. We then show results of our approach on ex-vivo and in-vivo patient data and finally present conclusions and future work.

## 4.2 Requirements and complications

In section 3.1 we showed how the CAD-like functionality of DeVIDE could be used to perform pre-operative planning for a glenoid replacement. Another important and related application is the evaluation of the fixation of different glenoid prosthesis designs with finite element models. Both of these applications require an accurate triangulated surface defining the outer boundaries of the relevant skeletal structures of the shoulder.

The main goal of the work documented in this chapter is the development of methods to segment and to derive patient-specific triangle meshes from CT images of the shoulder. These meshes should be accurate and topologically correct. By accurate we mean that the meshes should describe the true shapes of the skeletal structures that are being segmented. By topologically correct, we mean that the topology of the mesh should correlate with our assumptions about the structure in question. See section 4.6 for more details on the topological assumptions. The methods should also be applicable to CT data of shoulders of patients with rheumatoid arthritis or osteoarthritis, as shoulder replacement patients very often suffer from these diseases. Figure 3.1 shows the different skeletal structures of the shoulder. We are most interested in the scapula and the humerus. For each skeletal structure, a separate mesh should be generated.

In figure 4.1, a cadaver scapula is shown. From this example and the anatomical sketch in figure 3.1, it can be seen that the scapula is geometrically complex with many concavities. The geometric complexity complicates the use of many traditional segmentation techniques. Also, the scapula is often extremely thin, especially in the interior. This, together with the partial volume effect<sup>2</sup>, results in parts of the scapula almost disappearing from the CT data. Figure 4.2 shows an example of this phenomenon. In the top-right part of the image, the scapula is so thin that it's almost invisible.

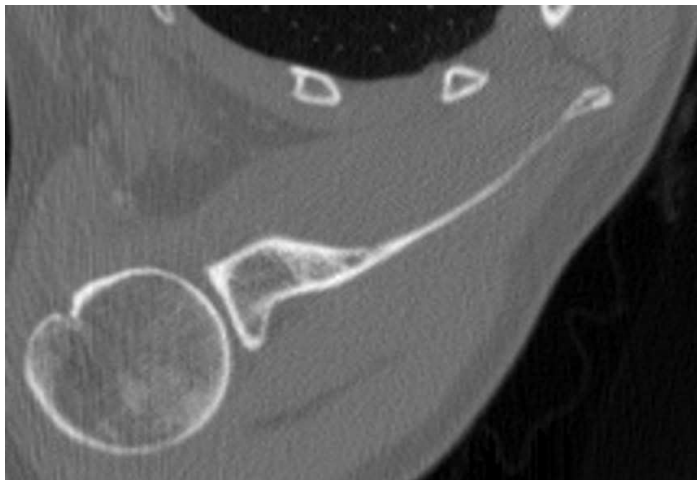
Joint space narrowing is often associated with rheumatoid arthritis. When the patient's shoulder is affected, the space between for instance the humeral head and the scapula could be extremely narrow. In many cases, CT images of such a shoulder will show a humerus and scapula that appear to be fused together. Figure 4.3 shows a single axial slice of such a dataset. Also note that the cortical bone of the humerus, i.e. the outer layer of dense bone material, is represented with a far lower than usual intensity. This is due to both decalcification of the bone and the partial volume effect.

---

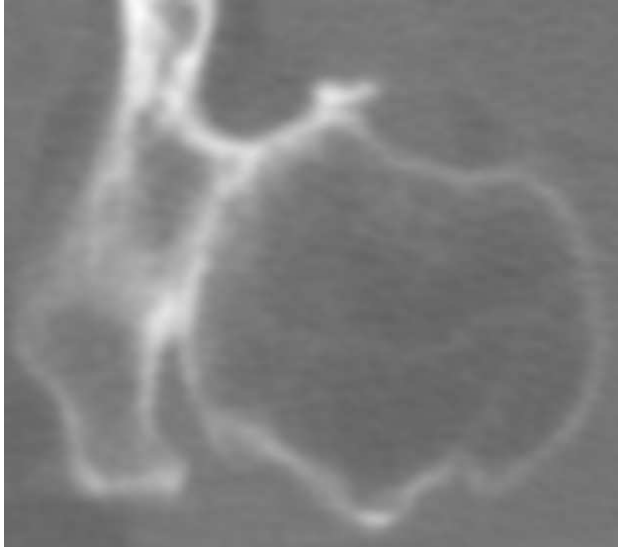
<sup>2</sup>The partial volume effect refers to the phenomenon where a voxel that actually contains two or more material types has an intensity value that is associated with a completely different material type. This happens when the resolution of the scanner is too low relative to the dimensions of the structures that are being imaged.



**Figure 4.1:** A cadaver scapula. In the centre, it is thin enough to be translucent. Together with the partial volume effect, this leads to extra complications during segmentation.



**Figure 4.2:** An axial slice from a shoulder CT dataset. In the top-right part of the image, the extreme thinness of the scapula and the partial volume effect results in voxels where the intensity is far lower than the expected intensity for bone. In extreme cases, this can result in parts of bony structures becoming almost invisible in CT images.



**Figure 4.3:** An axial slice from an arthritic shoulder CT dataset showing a humeral head and a scapula that appear fused together. This is due to the joint space being extremely narrow and the limited resolution of the CT scanner.

In short, the complex and non-convex geometry of the scapula, the thinness of the scapula, joint-space narrowing and bone decalcification, together with the limited imaging resolution and the consequent partial volume effects, complicate the successful segmentation of the skeletal structures in the shoulder. Our approach addresses these problems. It can also be applied to healthy shoulders where more straight-forward techniques are also not always successful.

### 4.3 Related Work

In our own research project a technique was developed to determine the radius and centre of the humeral head using a modified Hough transform [90]. This is an effective approach and might be combined with our current work, but for the purposes stated in the introduction we required complete object representations of the skeletal structures in the shoulder.

In [13] and [84] shoulders are segmented from T1-weighted MRI images of healthy shoulders. This work is based on MRI data, but is relevant because it specifically targets the shoulder. A separate 2D segmentation is performed on each 2D slice. This segmentation is based on Wiener filtering, Sobel edge detection and region growing. Morphological techniques are used to reconstruct a smooth 3D segmentation from the 2D segmentations. The results are good. However, this work focuses on healthy subjects and reports an average acromio-humeral distance of 6mm. From the included visualisations, the gleno-humeral distance is

also quite large. In addition, their work focuses on a relatively small region of interest around the gleno-humeral joint. Their technique does not address the problems of decreased joint space, scapular thinness and bone decalcification that we describe in section 4.2. It does however indicate the promise of MRI-based techniques for the segmentation of the shoulder skeleton.

A general approach for segmenting skeletal structures from CT data is presented in [35]. This approach is based on 3D region growing with locally adaptive thresholds following by a mixture of 3D and 2D morphological operations to close holes in the segmentation. The resulting segmentation is then smoothed by adjusting its containing iso-surface. The approach is applied on the hip, the knee and the skull. The authors state that a site-specific approach is required for separating different bony structures at joints *before* their techniques can be applied, and that this separation is site specific. In the case of the hip, they make use of a manually-placed sphere. In our opinion, the determination of this site-specific separation method is one of the most challenging tasks in the case of the shoulder.

In [102], hip joints are segmented by making use of histogram-based thresholding and an interesting hip-specific approach based on the convex hull of the initial thresholding. A number of landmarks are chosen relative to the convex hull. These points are used to construct an initial sphere around the femoral head. This sphere, more hip-specific parameters and the Hough transform are used to estimate an ellipsoid approximating the femoral head. This ellipsoid is used to derive an initial estimate of the joint space. This initial estimate is then refined on a Hessian-filtered version of the data. This is a promising technique, but is very much hip-specific. A similar approach could possibly be derived for the shoulder, but would require a significant amount of research.

There are a number of examples in literature on segmenting skeletal structures from CT data, but very few of these address the problem of separating neighbouring skeletal structures, such as those found in joints. If we further narrow the problem to that of separating neighbouring skeletal structures from pathological CT data, even fewer examples remain. When we add the extra specification that it should deal with shoulder joints, there's almost nothing to be found. The work documented in this chapter attempts to fill that niche.

## 4.4 Method Overview

Due to the great variation in patient shoulders and their CT images, finding a fixed technique that always results in a successful automatic segmentation is most unlikely. Instead, we present a flexible approach consisting of a small selection of algorithms that are suitable for segmenting shoulder data and work well in concert. Furthermore, most approaches will require a number of parameters to be experimented with until a satisfactory segmentation is derived. With both these aspects in mind, it is important to make use of a flexible platform or system that facilitates experimentation with the different permutations of our approach and its various parameters. DeVIDE, documented in chapter 2, is well suited to this kind of work.

Figure 4.4 shows our general approach to the segmentation of the skeletal structures of the shoulder. Ovals signify data and blocks signify operations. In broad terms, the scheme

is based on the use of a deformable surface that is initially larger than the structure that is to be segmented and is then deflated to fit precisely on the outside surface of that specific structure. For this to work, a suitable initial surface is required as well as image features with which the deformable surface can determine when it has reached an edge and can therefore stop deforming. In the diagram, the initial surface is indicated by the *Initial Level Set* oval and the stopping features are indicated by the *Edge Features* oval.

The following sections explain, step by step, the traversal from *CT images of shoulder* to *Final Surface*.

## 4.5 Deriving Structure Masks: Method A

A first, and crucial step, is to derive conservative structure masks for the structures that are to be segmented and in some cases also for structures that border on those that are to be segmented. We define a conservative structure mask as a list of *all* contiguous voxels that can be reasonably assumed to be contained within the boundaries of the structure that is being masked. Our use of *conservative* indicates that we prefer false negatives over false positives.

There are two methods of deriving these structure masks, denoted by the two grey blocks labeled *A* and *B* in figure 4.4.

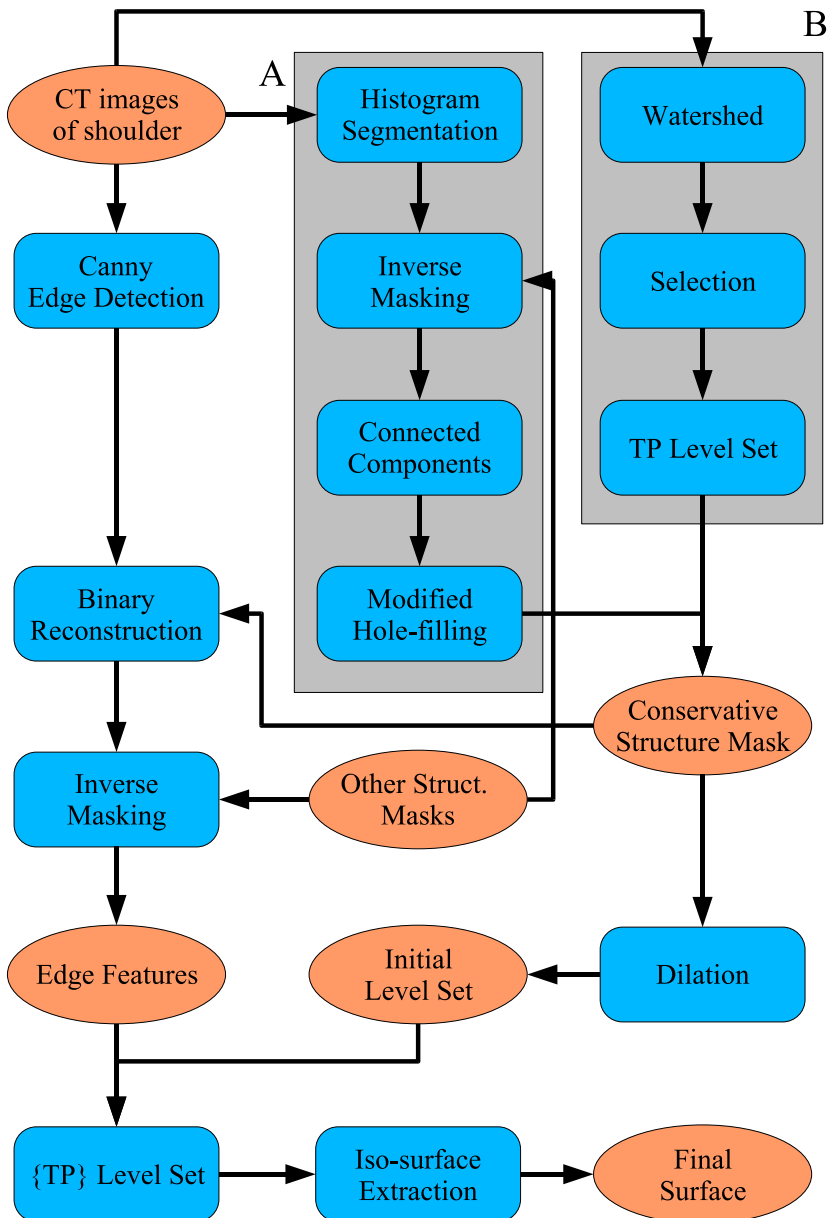
Method A is used in cases where the structure that is being segmented does not touch other foreground structures, or when structures *do* touch but structure masks are available for all neighbouring structures. In figure 4.2 there is a visible separation between the humerus and the scapula. In this case, method A should work without the availability of structure masks for neighbouring structures. Method B is used in cases where structures do touch, and is discussed in section 4.8. This discussion is deferred until after the rest of the complete segmentation approach has been explained. Generally, both methods will be used for any CT dataset. For example, method B can be used to generate a structure mask for a humerus that touches a scapula. Subsequently, that structure mask can be used in the inverse masking step of method A to generate a mask for the scapula.

Method A is based on histogram segmentation, connected components labelling and selection, inverse masking, and a modified hole-filling algorithm. These methods are explained in the following subsections.

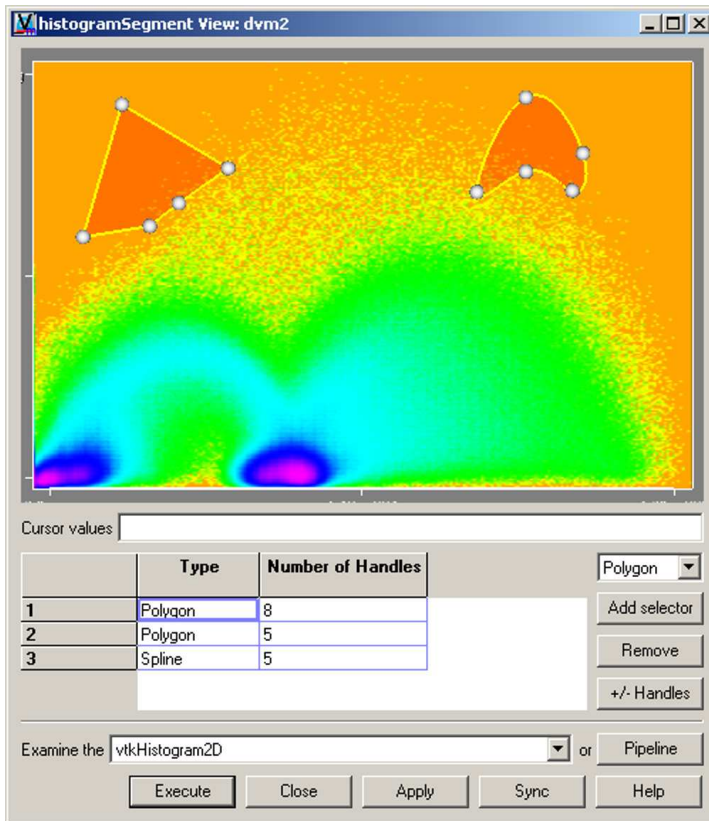
### 4.5.1 Histogram Segmentation

The histogram segmentation is similar to the method described in [31] and based on concepts presented in [37] and further applied to material classification in [74]. It makes use of a linked, or coupled, view that shows a 2-D histogram of image gradient magnitude over image intensity for the CT data that is being examined. By delineating regions on this histogram with closed polygons or splines, the user can select data points on the basis of their intensity and gradient magnitude. Figure 4.5 shows a screen-shot of this interface.

All corresponding points in the CT data are marked as *on*, whilst all other points are marked as *off*. The *on* points constitute an initial bone segmentation. This takes place inter-



**Figure 4.4:** A flow chart illustrating our approach to the segmentation of the skeletal structures of the shoulder. Ovals signify data and blocks signify operations. The A and B blocks show two distinct ways to generate the initial conservative bone mask. *TP* signifies *Topology Preserving*.



**Figure 4.5:** Screen-shot of the Histogram Segmentation DeVIDE module. The user is delineating two areas on the histogram with a polygon and a spline. See section 4.5.1 for an explanation of the arc-like shapes that are visible in the histogram. The colour map is logarithmic and has been optimised to accentuate the different arc-like shapes in order to facilitate the segmentation. See colour figure C.4.



actively: after each change that the user makes to the delineated regions on the histogram, a view of the CT images immediately shows the *on* voxels overlaid on grey-scale images of the raw data. In our case, the *on* voxels are bone voxels.

Usually, one of the first methods for segmenting bone voxels from CT data is by making use of intensity thresholding, i.e. all voxels above a certain intensity threshold are considered to be bone. On the gradient magnitude vs. intensity histogram, this could be done by selecting *all* voxels to the right of the threshold. Even making use of upper and lower thresholds on both the intensity and gradient magnitude scales comes down to an axis-aligned rectangle on the histogram. It is clear that being able to delineate an arbitrary number of arbitrarily-shaped regions on the histogram gives us far more control over the resulting segmentation.

In addition, the general appearance of this histogram is similar for most medical CT datasets. This can be explained by studying the edge model as presented in [37]. In short, an ideal transition between two material types should in reality be a discontinuous edge, going for example directly from air to tissue, or from tissue to bone. Due to the fact that the measuring process is band-limited, the resultant measured edge is actually this step edge smoothed by a Gaussian function approximating the frequency response of the measurement process. This model is illustrated in figure 4.6. The bottom-most plot shows both the measured edge and its normalised derivative with respect to position. The peak of the derivative is at the position of the true edge.

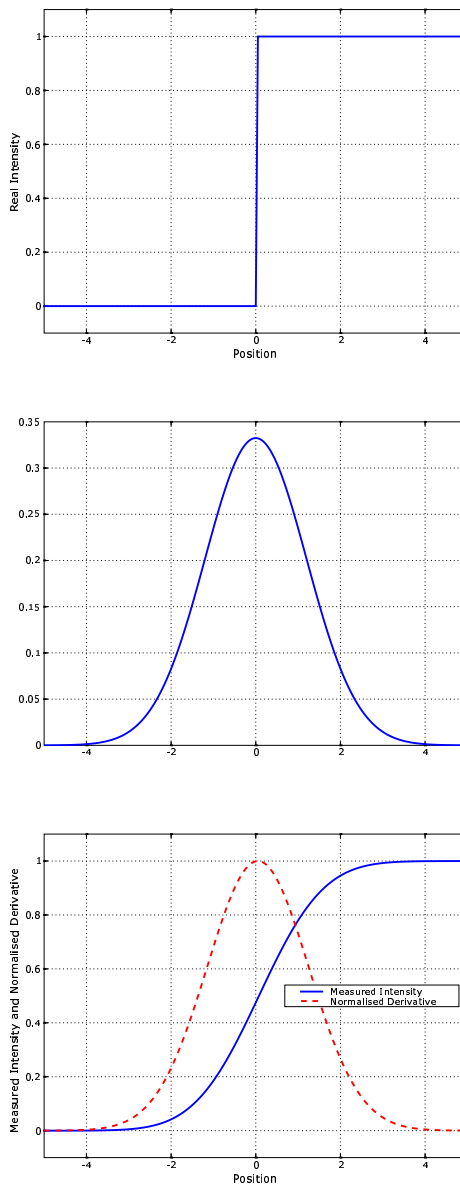
With this edge-model in mind, it is clear that the arc-like shapes visible in figure 4.5 should occur in all gradient magnitude over intensity histograms made from CT data of discrete objects. Each discrete blob on the  $x$ -axis represents a different material type. In this case, the blob on the far left represents air, the blob in the middle actually consists of two blobs that are very close to each other: the left sub-blob represents trabecular bone and the right sub-blob represents soft tissue. Trabecular bone, also known as spongy or cancellous bone, is a less dense type of bone found in the interior of skeletal structures.

Each arc represents a transition between two material types. There is a clearly discernible arc from air to the trabecular bone and soft tissue composite blob. There is unfortunately no clearly discernible blob representing the cortical bone, which refers to the hard protective bone that surrounds trabecular bone and also defines the outside surface of all skeletal structures. In addition, the arc originating from the soft tissue blob disperses quickly before disappearing. This is due to the fact that there are relatively few cortical bone voxels and that these voxels cover a relatively wide range of intensity and gradient magnitude values.

In spite of these factors, this tool makes it relatively easy to make an initial bone voxel segmentation. In general, a region is chosen so that the trabecular bone and soft tissue blob is excluded, but the arc originating from that blob is included, as well as all positions to the right. An added advantage is that the histogram segmentation aids in the understanding of the data that is being examined. Figure 4.7 shows an example histogram segmentation result.

## 4.5.2 Connected Components Labelling and Selection

After the initial bone voxel segmentation has been done, a connected components labelling algorithm is applied to the initial segmentation. This algorithm labels each distinct, contigu-



**Figure 4.6:** An ideal step edge, or material transition, the Gaussian representing the band-limited frequency response of the measurement process and the resultant measured edge as well as its normalised derivative with respect to position.



**Figure 4.7:** An example histogram segmentation result. This is the same slice as shown in figure 4.2.

ous object with a different value. If segmented skeletal structures do not touch, they will be assigned different labels. The user can then easily select one of the labeled objects. In figure 4.8, the scapula has been selected from the connected component labeling. This is the same slice as shown in figure 4.2.

If the connected component labeling does not assign different labels to structures that are close together, the histogram segmentation should be adjusted to be more conservative, i.e. more false negatives should be allowed. Often this adjustment will result in a successful separation. If not, inverse masking should be used as described in the next section, or one could choose to make use of method B for deriving structure masks as explained in section 4.8.

### 4.5.3 Inverse Masking

In cases where the connected component labelling is not able to separate structures that are close together, and adjustment of the histogram segmentation does not remedy the problem, inverse masking should be performed. Inverse masking refers to performing a logical AND between the result of the connected component labelling step and the complement, or inverse, of the mask (or masks) of the offending structures. This will effectively remove the offending neighbouring structures from the image. Masks that are used for this step can be generated by method B. This is explained in section 4.8.



**Figure 4.8:** The scapula has been selected from the connected component labeling of an initial bone segmentation shown in figure 4.7.

#### 4.5.4 Modified Hole-filling

The desired result of the structure mask sequence of algorithms is a conservative structure mask. We defined this as being a list of all contiguous voxels that can be reasonably assumed to be contained within the boundaries of the structure that is being segmented. This means that we would like to fill the holes that inevitably form during the initial histogram segmentation. These holes are also visible in figure 4.8.

In this section, we propose a modified hole-filling algorithm that performs well on our data. To explain this procedure, we need to set out some basic morphological principles. This exposition is based on material from [25].

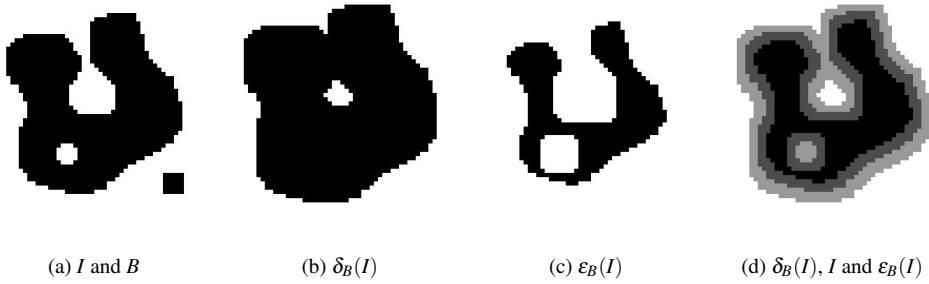
The morphological binary dilation  $\delta_B$  of a binary image  $I$  by a structuring element  $B$ , is defined as the union of the set of binary images that result when  $I$  is offset by all positions  $b$  in  $B$ . In other words:

$$\delta_B(I) = \bigcup_{b \in B} I + b$$

Geometrically put, the dilation of  $I$  with structuring element  $B$  is the set of points such that when the mirrored structuring element, i.e.  $\check{B} = \{-b | b \in B\}$ , is translated to those points, it still intersects  $I$ . Dilation expands objects at their boundaries.

The morphological binary erosion  $\varepsilon_B$  of a binary image  $I$  by a structuring element  $B$ , is defined as the intersection of the set of binary images that result when  $I$  is negatively offset by all positions  $b$  in  $B$ . In other words:

$$\varepsilon_B(I) = \bigcap_{b \in B} I - b$$



**Figure 4.9:** A simple example of binary dilation and erosion. (a) shows the original binary image  $I$  (black pixels denote points) with at its bottom right the  $5 \times 5$  rectangular structuring element  $B$ . (b) shows the dilation of  $I$  with  $B$  and (c) shows the erosion of  $I$  with  $B$ . In (d), all images are shown together to illustrate how dilation expands  $I$  at its boundaries and erosion eats  $I$  away at its boundaries.

Geometrically put, the erosion of  $I$  with structuring element  $B$  is the set of points such that when the structuring element  $B$  is translated to those points, it is still completely contained in  $I$ . Erosion eats objects away at these same boundaries.

Figure 4.9 illustrates dilation and erosion on a simple binary test object.

An opening  $\gamma_B$  with structuring element  $B$  is defined as an erosion followed by a dilation, or:

$$\gamma_B(I) = \delta_B \varepsilon_B(I)$$

This will remove small structures and opens up holes that are near object boundaries.

A closing  $\varphi_B$  with structuring element  $B$  is defined as a dilation followed by an erosion, or:

$$\varphi_B(I) = \varepsilon_B \delta_B(I)$$

This will remove small holes and merge small structures with larger structures if they are close together.

The conditional dilation of size 1 of binary image  $J$  by a structuring element  $B$  within binary image  $I$  is:

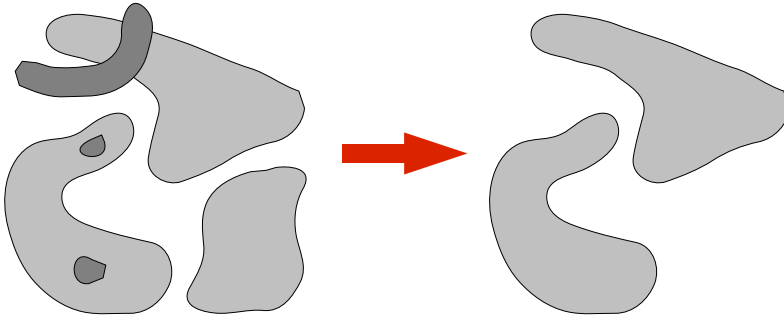
$$\delta_B^1(J|I) = \delta_B(J) \cap I$$

Image  $J$  is dilated, or expanded, with the structuring element  $B$  and the result is constrained to lie completely within  $I$ . The conditional dilation of size  $k$  is calculated by performing the above operation  $k$  times recursively, i.e.:

$$\delta_B^k(J|I) = \overbrace{\delta_B^1 \delta_B^1 \cdots \delta_B^1}^{\times k}(J|I)$$

Then the binary morphological reconstruction of mask image  $I$  from marker image  $J$ ,  $\rho_B(J|I)$ , is:

$$\rho_B(J|I) = \bigcup_{k \geq 1} \delta_B^k(J|I)$$



**Figure 4.10:** An example of morphological reconstruction of a mask image  $I$  (light grey) from marker image  $J$  (dark grey). Contiguous objects of the mask  $I$  that are marked by pixels of the marker  $J$  are extracted.

i.e. the union of the conditional dilations of *all* sizes  $k$ . More simply put, the dilation of marker image  $J$  is constrained to lie within  $I$ . The result of this is dilated and constrained again within  $I$ . This is continued until stability is reached, i.e. until there is no change in the output image after each iteration.

Thinking this through, it becomes clear that the binary morphological reconstruction can be used to select contiguous objects in image  $I$  that are marked by image  $J$ . Figure 4.10 shows an example of binary reconstruction.

Binary reconstruction is often used as part of a hole-filling, or close-hole, technique. A hole is defined as any contiguous background region in a binary image that is completely surrounded by foreground pixels. Hole-filling is performed by creating a marker image  $J_\beta$  where all boundary points (pixels in 2D, voxels in 3D) are set to *on*, or foreground, *except* where a foreground object intersects the boundary, and all interior points are set to *off*, or background. The complement of binary image  $I$ , or  $I^c$ , is calculated by inverting all points, i.e. *on* points are set to *off* and vice versa. The image  $I^c$  is then reconstructed from boundary marker image  $J_\beta$ . The result image will logically consist of all contiguous background regions that touch any of the borders. When it is inverted, the result is an image where all background regions that do not touch any of the borders, holes in other words, have been filled. More compactly, the hole-filling operator  $\psi_{\text{hf}}(I)$  is defined as:

$$\psi_{\text{hf}}(I) = [\rho_B(J_\beta | I^c)]^c$$

In order to remove the holes from the result image of the initial histogram segmentation and the connected component labeling, we combine a morphological closing with a special hole-filling operator where the 3D boundary marker image,  $J_\beta$ , is modified so that not all boundary points are *on*, as is usually the case for this operator. Our implementation allows the deactivation of one or more of the six boundary planes of the boundary marker volume. This is necessary in cases where cavities have been truncated by the CT imaging. For example, the humeral cavity often touches one of the boundary planes. In that case, that boundary



**Figure 4.11:** A slice of the conservative structure mask of the scapula, as generated by method A.

plane can be deactivated, and the hole-filling will successfully fill the humeral cavity as well, although it is not a hole according to the definition given above.

This hole-filling and the closing operations are actually interleaved: first the image is dilated, then the hole-filling operator is applied and finally the result is eroded. Denoting the adapted hole-filling operator with  $\psi_{\text{hf}2}$ , the modified hole-filling process is

$$\varepsilon_B(\psi_{\text{hf}2}(\delta_B((I))))$$

We have implemented the efficient grey-scale reconstruction algorithm from [93] for 3D. It reduces to binary reconstruction when applied to binary images, but is orders of magnitude faster than a straight-forward implementation. Applying the modified hole-filling, i.e. dilation, hole-filling with adapted border marker and finally erosion, to the image in figure 4.8 yields the result shown in figure 4.11. This result is a *Conservative Structure Mask* as indicated in figure 4.4. All cavities in the scapula have been filled.

## 4.6 Edge features and Initial Level Set

As described in section 4.2, we require accurate and topologically correct outside surfaces of the skeletal structures in the shoulder, and specifically of the humerus and the scapula. Additionally, the thinness of the scapula and the limited resolution of the CT data often manifests in virtual holes through the scapula. However, it is also possible that there are real holes through the interior of the scapula. Because virtual holes almost always occur and real holes are a relatively uncommon occurrence, we make the assumption that the outside

surface of the scapula has a topological genus of 0 for the majority of cases. For the other skeletal structures, this assumption also holds. Integrating this *a priori* information with our segmentation approach greatly increases its efficacy.

The *Initial Level Set* in figure 4.4 refers to an initial surface that will be deformed to result in the final accurate surface describing the exterior of the skeletal structure that is being segmented. Due to the assumption above, and the fact that we wish to derive an exterior surface, we utilise an initial genus 0 surface that is larger than the desired final surface. This initial surface is calculated by dilating the *Conservative Structure Mask*. The size of the structuring element is chosen empirically so that the initial surface is larger, i.e. contains, the final surface and so that it has genus 0 topology.

In this case, the edge features are voxels with an *on* value that are located on the edges of the structures that we are segmenting. The deformable surface will deflate until it fits and encloses all the edge features. They are determined as follows:

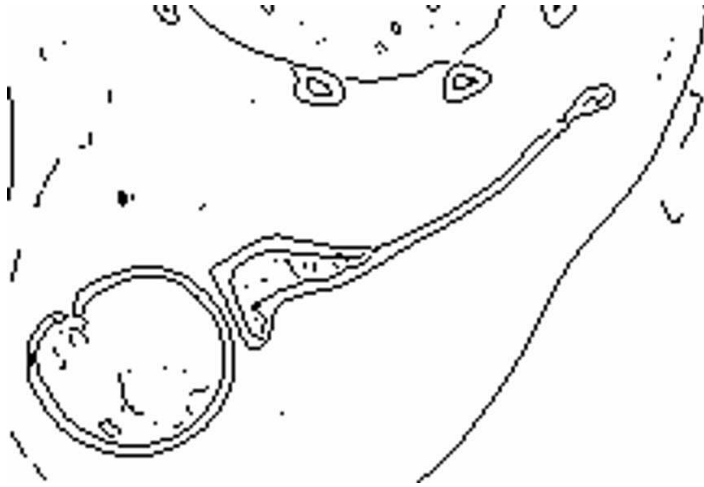
1. A 3D Canny edge detection [15] is applied to the input data. Figure 4.12 shows a slice of an example Canny edges volume.
2. The binary reconstruction  $\rho_B$  of the Canny edges from the *Conservative Structure Mask* is calculated. This eliminates all Canny edges that are not connected to edges that are marked by the structure mask. Refer to section 4.5.4 for details on binary reconstruction.
3. Optionally, an inverse masking step is performed. This step is only required when edge features of neighbouring structures that did not get removed in the reconstruction step affect the deformable surface. This happens when these offending edge features are contained within the initial surface, thus preventing the deflating surface from reaching its final goal, the scapula edge features. In these cases, the conservative structure mask of the offending neighbouring structure, optionally dilated, is inverted and used to mask the complete Canny edge volume. This effectively removes all edge features of the offending neighbouring structure.

Figure 4.13 shows a slice of the example scapula edge features. The optional inverse masking step was performed with a dilated humerus structure mask.

## 4.7 Level Set Segmentation and Topology Preservation

With the edge features and initial level set, we can continue to the final stage of the segmentation approach. As explained in section 4.4, this involves a deformable surface that is initially larger than the structure that is to be segmented, and then deflates until it fits precisely around that structure. This surface deforms over time, so there is clearly a notion of speed as well.





**Figure 4.12:** A slice of an example Canny edges volume.



**Figure 4.13:** A slice of the scapula edge features. This is after binary reconstruction of the Canny edges volume from the conservative structure mask of the scapula and inverse masking with the structure mask of the humerus. The isolated structure at the upper left is also part of the scapula.

### 4.7.1 Curve Evolution by Level Sets

Let  $C(p,t) = \{x_0(p,t), x_1(p,t), \dots, x_{N-1}(p,t)\}$  be the family of N-dimensional closed contours, parametrised by  $p$ , that are the result of moving the initial contour  $C(p,0)$  along its normal vector field over time  $t$  with speed  $F(C(p,t))$ . Then the evolution of this contour can be described with:

$$\frac{\partial C(p,t)}{\partial t} = F(C(p,t))\vec{n}(C(p,t))$$

One way of implementing this evolving contour is to make use of a discretised parameterisation. The initial contour is sampled at a number of points and the separate points are moved during each discrete time step according to a discretised version of the evolution equation above. At any point the original contour can be reconstructed as line segments in 2D and triangles in 3D.

This approach has a number of inherent problems, however. In the case of an expanding contour, new points have to be inserted as the resolution of the existing points will be too low to sample the contour accurately. In the case of a shrinking contour, points have to be removed. It could also happen that the contour, due to its natural evolution, self intersects. This situation has to be explicitly prevented. Furthermore, the splitting and merging of contours is a very complex implementation problem.

A far more elegant approach, the level set technique, was introduced in [58]. The contour  $C(p,t)$  is *implicitly* represented as the zero level set, or iso-contour, of an embedding scalar function  $\phi(x,t)$ . In other words, the contour is represented at time  $t$  by  $C(t) = \{x|\phi(x,t) = 0\}$ . The embedding function is called the level set function. It is convenient to select a signed distance function as the level set function.

Instead of directly evolving the contour, its level set function is evolved. Given the initial contour  $\phi(x,t=0)$ , the level set function is evolved as follows:

$$\frac{\partial \phi(x,t)}{\partial t} = -F(x,t)|\nabla \phi(x,t)| \quad (4.1)$$

where  $F(x,t)$  is a scalar speed function,  $\nabla$  is the gradient operator and  $|\nabla \phi(x,t)|$  is the magnitude of the gradient of the level set function [75].

The level set technique solves all problems inherent to the discretised parametric contour evolution approach. Topological changes, i.e. splitting and merging contours, are handled naturally. Self intersections are automatically avoided. All difficulties related to working with discrete points are not relevant. In addition, the level set technique can be used for contour evolution up to an arbitrary number of dimensions.

### 4.7.2 Geometric Deformable Models

Deformable models are N-dimensional contours that evolve within N-dimensional images under the control of external, e.g. image-based, and internal, e.g. curvature minimisation, forces [36]. The idea is that this contour evolves until it accurately delineates an object by the end of its evolution.

These contours can be implemented as parametric deformable models or geometric deformable models. When a discretised parameterisation, e.g. points and lines, is used to model the evolving contour, it is a parametric deformable model. When a level set approach is taken, it is a geometric deformable model. Due to its many advantages, we prefer making use of the latter approach.

In this setting, a contour is evolved along its normal direction with a speed function  $F(x, t)$  consisting of three terms: an inflationary or deflationary speed  $F_{\text{prop}}$  aligned with its normal direction, a curvature-dependent speed  $F_{\text{curv}}$  and a passive advection term  $\vec{F}_{\text{adv}}$  determined by an underlying velocity field. Equation 4.1, the level set propagation, can then be rewritten as:

$$\frac{\partial \phi(x, t)}{\partial t} = -F_{\text{prop}}(x, t)|\nabla \phi(x, t)| - F_{\text{curv}}(x, t)|\nabla \phi(x, t)| - \vec{F}_{\text{adv}}(x, t) \cdot \nabla \phi(x, t) \quad (4.2)$$

When attempting to segment structures from medical images, we need to pick a suitable initial level set, representing the initial contour. This was one of the end results of the work described in section 4.6.

We also need an advection field  $\vec{F}_{\text{adv}}(x, t)$  that will pull the contour in the direction of the edges of the object that we want to segment. The edge features, described in section 4.6, can be used to create this advection field. More complex algorithms can be applied [100], but because our initial level set and edge features have been carefully chosen, we make use of the gradient of the edge features. Close to the edges, the gradient vectors will point towards the edges.

During the final segmentation step, our initial contour is guaranteed to be outside of the object that is being segmented, so the propagation force is chosen to point inwards, but with a magnitude taken from the inverse of the edge features. In other words, the propagation force will deflate the contour until it reaches the object edges. When any part of the contour reaches an edge, the deflationary force will be set to zero for that part.

The curvature-dependent term is chosen to be the negative local curvature, also gated with the magnitude of the inverse of the edge features. In this way, the local curvature of the contour is minimised.

### 4.7.3 Topology Preservation

As explained in section 4.7.1, one of the advantages of the level set method for contour propagation is that topological changes are naturally handled. However, in section 4.6 we stated and motivated our assumption that the outside surfaces all skeletal structures in the shoulder that we were segmenting possessed genus 0 topology. In order to make use of the genus 0 assumption, we have adapted the level set implementation in ITK [32] to be topology preserving according to the method presented in [28].

This modification is based on the theory of digital topology, and specifically on simple points and their classification by topological numbers [5].

A simple point is a point that, when removed from an object, or added to an object, does not change its topology. By calculating two topological numbers of a point, one for the foreground and one for the background, it can be determined whether that point is a simple point

or not. The topological numbers are based on the number of  $n$ -connected components within a geodesic neighbourhood of the point, where  $n$  must be chosen differently for the foreground and the background and must form a topologically complementary pair, e.g.  $\{26, 6\}$ .

We chose to use 6-connectivity for the foreground and 26-connectivity for the background. This means that we had to calculate, for each point  $x$  that we tested, two topological numbers within the  $3 \times 3$  neighbourhood of  $x$ :

$$T_6(x, X) = \#C_6(N_6^2(x, X))$$

and

$$T_{26}(x, X') = \#C_{26}(N_{26}^1(x, X'))$$

$N_6^2(x, X)$  is the second order 6-connected geodesic neighbourhood of the point  $x$  with respect to foreground  $X$ . What this means, is that we record the *on* voxels that are 6-connected to  $x$ . Because it is the second order neighbourhood, we also record the voxels that are 6-connected to this first set of voxels.  $x$  itself is not recorded. All voxels that we have now recorded form the neighbourhood  $N_6^2(x, X)$ . The topological number  $T_6$  is the number of 6-connected components, or  $\#C_6$ , in this neighbourhood.

$T_{26}$  was calculated for the background  $X'$ . In this case, the neighbourhood is first order and consists of the 26-connected neighbours of  $x$  that are *off*, i.e. background.  $T_{26}$  is the number of 26-connected components in this neighbourhood.

If  $T_6(x, X) = T_{26}(x, X') = 1$ , i.e. both topological numbers are equal to 1, then  $x$  is a simple point.

The geodesic neighbourhoods involved are all within the  $3 \times 3$  neighbourhood of the point. This, added to the fact that counting the connected components is done with a fast algorithm such as that documented in [7], results in an efficient implementation.

At every timestep of the contour propagation, the values at the grid points of the embedding level set  $\phi(x, t)$  are updated according to equation 4.2. When a value changes sign, this means that that point is either being added to or being removed from the object that is delineated by the deformable model. If that point is a simple point, the topology of the object does not change, and the sign change can be allowed. If the point is not a simple point however, the sign change is prevented. By doing this, the topology of the evolving contour is preserved. In our case, starting with a genus 0 contour means that the final contour will also be genus 0.

Applying the topology preserving level set propagation described above along with the initial level sets and edge features yields a final level set. The final object-delineating surface can be derived from this level set by making use of an iso-surface extraction method such as the Marching Cubes algorithm [47].

## 4.8 Deriving Structure Masks: Method B

In cases where method A does not succeed in generating the required conservative structure mask because touching structures can not be separated, method B is utilised. Generally, method B is used on the humerus and the clavícula. The resulting masks can then also be

used for the inverse masking step in method A in order to generate a structure mask for the scapula. This method consists of a watershed segmentation, followed by selection and a level set-based deformable model refinement stage.

### 4.8.1 The Watershed Segmentation Algorithm

The watershed algorithm [94] is a well-known segmentation technique from the field of mathematical morphology. This technique segments a function  $f(x)$ , for example a 2-D image, into catchment basins.  $f(x)$  is seen as a digital elevation map, i.e. a mapping from position  $x$  to elevation  $f(x)$ . A catchment basin is defined as all points from whence the steepest path of descent ends in the same local minimum of the function  $f(x)$ .

This technique is a popular algorithm for object segmentation. For example, it can be applied to the gradient magnitude of an input image. In the gradient magnitude image, objects with homogeneous intensity in the input images will be visible as areas of low gradient magnitude surrounded, at the object edges, with a high gradient magnitude border. This can be seen as a large catchment basin.

In its native form, however, the watershed will greatly over-segment such an image, as the large catchment basin that is the object actually consists of many smaller catchment basins, each with its own local minimum. There are two popular remedies for this problem. The gradient image can be modified so that all local minima within the large object catchment basin are removed [93]. Alternatively, after the watershed has been applied, catchment basins can be merged according to the height of the ridges that separate them. The smaller constituent basins are separated by lower ridges. We make use of the latter approach, in which case a merge level must be chosen by the user. All ridges lower than this level are erased by merging the relevant catchment basins.

We apply the watershed to the unprocessed input images. The skeletal structures that we are segmenting consist of an outside shell of cortical bone and an interior of trabecular bone. As the density of the cortical bone is much higher than that of the trabecular bone, this has a much higher intensity in the CT images. This forms a catchment basin and can be segmented by the watershed and a merging step. This is in general *not* successful in the case of the scapula, due to its irregular geometry. However, the watershed is able to separate bones that touch and, due to low resolution, appear to be fused in the CT data.

### 4.8.2 Selection

The end result of the watershed segmentation and merging step is an image where all catchment basins have a unique label. The basin, or basins, making up the structure that is being segmented have to be chosen by the user. The result is an initial segmentation of that structure.

### 4.8.3 Refinement by level set-based deformable model

In section 4.7, we explained how the initial level set could be deflated to fit the outside surface of the skeletal structure. The watershed initial segmentation is, for the most part, inside the outermost edge of the cortical bone. In this case we make use of exactly the same deformable model formulation and parameters, but the propagation speed term will be inflationary. Because the Canny edge detection generates edges for both the inner and outer edges of the cortical bone, we do have to make sure that the initial segmentation is at least outside the inner edge by making use of a dilation with a small structuring element.

The end result of the deformable model segmentation is a signed distance field embedding a surface that accurately describes the outside edge of the skeletal structure. By thresholding this distance field, a structure mask can be constructed. An added advantage is that by changing the threshold, the size of the structure mask can be very precisely controlled. This is necessary in cases where the mask is to be used in an inverse masking step (see section 4.5.3) and fine control is required over how much of the image data is to be masked out.

## 4.9 Results

Figure 4.14 shows the scapular surface as generated by the complete segmentation approach on the data shown in figure 4.2. The initial surface that was used in the deformable model segmentation is shown as the transparent outer surface. In this case, method A for generating the conservative structure mask was sufficient in all cases. The data is of high quality and the joint space is great enough in all cases.

The data shown in figure 4.3 constitutes a far more difficult problem for our segmentation. In that figure, the humerus and the scapula even appear to be fused together. Method B was used to generate structure masks for the humerus and also for the part of the clavicle neighbouring on the scapula. These two masks were then used as inverse masks in the inverse masking stage of Method A in order to segment the scapula successfully. Figure 4.15 shows the scapular and humeral surfaces derived from this data.

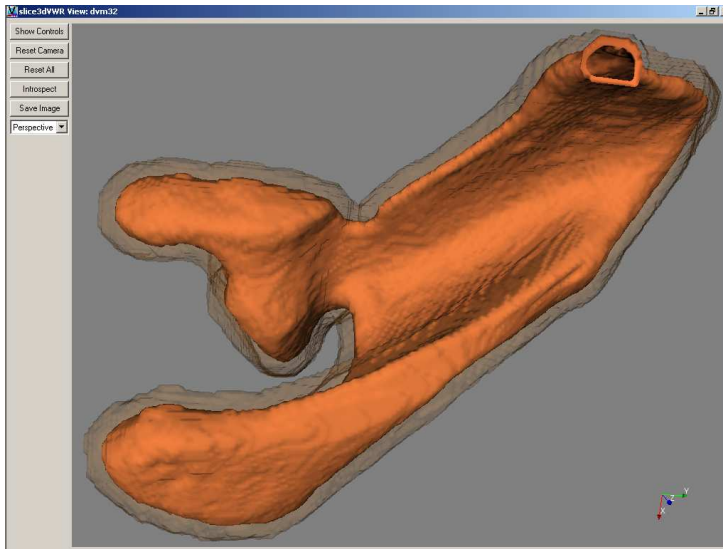
In order to validate the results of our approach, we compared its results on an experimental CT dataset of an *ex vivo* scapula to a manual expert segmentation.

The manual segmentation was performed by drawing a contour describing the outside scapular surface on each axial slice of the  $512 \times 512 \times 420$  dataset, sampled at a resolution of  $0.333 \times 0.333 \times 0.5\text{mm}$ , and then linking these contours together with triangles.<sup>3</sup> The mesh was specifically created for the finite element modeling of the scapula. It consists of 6900 triangles, is closed and it is of genus 0 topology, as expected.

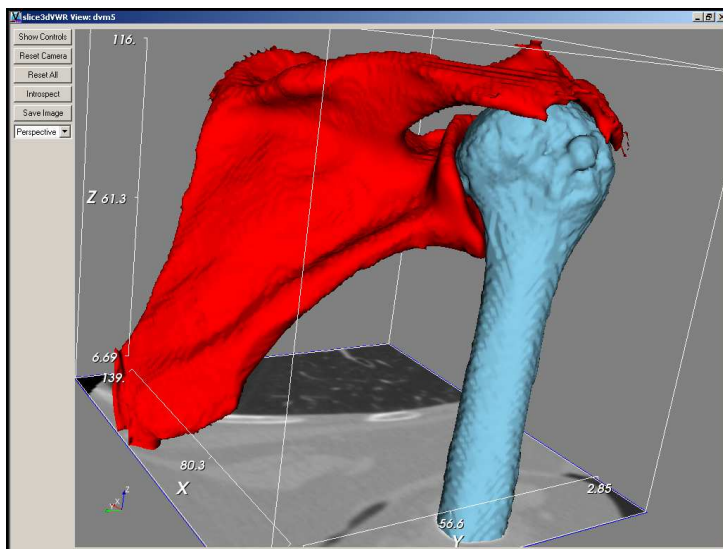
We used our method to segment the scapular mesh from the same data. Our resultant mesh consists of 729212 triangles, is closed and is also of genus 0 topology. The large number of triangles is the usual result of applying the Marching Cubes algorithm to a volume of this resolution to extract a surface of this complexity. We did not apply any decimation techniques to the mesh.

---

<sup>3</sup>Gerard Poort, a collaborator on the DIPEX project, performed the manual segmentation.



**Figure 4.14:** Segmented outer surface of the scapula from the data shown in figure 4.2. The transparent outer surface is the initial contour and the inner surface is the final contour delineating the scapula. The hole that is visible is due to the truncation of the CT dataset.



**Figure 4.15:** Segmented outer surfaces of the scapula and humerus from the data shown in figure 4.3.

	Absolute (mm)	% of bounding box diagonal
Maximum	3.7 mm	1.6 %
Mean	0.4 mm	0.2 %
RMS	0.5 mm	0.2 %

**Table 4.1:** Symmetric Hausdorff distance between the manually and automatically segmented meshes.

We compared the meshes using the Mesh software [3]. Table 4.1 shows the maximum, mean and root-mean-squared symmetric Hausdorff distance. The symmetric distance is the maximum of the distances from the first mesh to the second mesh and from the second mesh to the first. In the first column, the absolute distance in millimetres is shown and in the second column, the distance as a percentage of the largest diagonal of the bounding box of the scapula is shown. The RMS error is in the order of a single voxel. We find this result quite acceptable for a manual vs. automatic reconstruction.

Most of the outliers, and specifically the parts of the mesh that cause the maximum Hausdorff distance, are caused by cartilage being misclassified as cortical bone. This is especially a problem with cadaveric material, as the cartilage seems to harden over time and thus appears in the CT data with greater intensity.

Working with this large dataset proved difficult even on a machine with 1 Gbyte of physical memory and ample virtual memory. This was not due to a lack of total virtual memory, but due to the constraints of the 32 bit addressing system. Great care had to be taken to optimise memory efficiency so that our algorithm could run to completion. In the light of this, it is interesting to note that down-sampling the data to  $256 \times 256 \times 210$  resulted in a symmetric RMS Hausdorff distance of 0.7 mm, which is still a pleasing result.

## 4.10 Conclusions and Future Work

In this chapter we presented an approach for the segmentation of skeletal structures from CT images of the shoulder. More specifically, the approach focuses in deriving accurate and topologically correct meshes that describe the outside surfaces of the humerus and the scapula. Our technique is also applicable to shoulder data where joint space narrowing has occurred or where the bone density has been affected. Both of these phenomena are associated with rheumatoid arthritis and osteoarthritis.

The approach is based on deriving structure masks of the various skeletal structures, calculating edge features, using these masks to eliminate parts of the data and the edge features during the further stages of the segmentation, and finally refining a rough segmentation with a geometric deformable model.

The masks are a very important aspect of this technique. In some cases, they are used to exclude all other structures and in other cases, they are used to mask out the structure that they represent. There are two semi-automatic methods for deriving these masks. The method is chosen depending on the nature of the data and on the specific structure that is



being segmented. Obviously, masks for removing problematic features can also be manually constructed and applied.

We have also shown that the approach can be used on data where the skeletal structures have been severely affected with regards to bone density and joint space narrowing.

We attempted to validate the accuracy of our approach by comparing its results with a manually created surface mesh of a cadaveric scapula. The RMS symmetric Hausdorff distance was 0.5mm, which is a good result for this type of experiment. One problem that manifested, was that our technique has difficulty distinguishing cartilage from cortical bone, especially in the case where the cartilage has hardened to some extent. This caused the greatest differences in the mesh comparison.

Another very important conclusion of the work documented in this chapter can best be put in the words of Dr Michael Vannier<sup>4</sup>: *Good imaging beats good image processing*. In other words, optimising the acquisition process is much more effective than improving the utilised image processing techniques. Also in this regard, we plan to experiment with using a soft spacer and an arm restraint on patients during scanning. If the spacer is placed between the arm and the torso, as close as possible to the shoulder, and the arm restraint is used to keep the elbow close to the torso, the leverage on the upper arm might slightly increase the width of the gleno-humeral space. Such a slight increase would greatly facilitate the segmentation approach described in this chapter.

The development of this segmentation approach has also highlighted the usefulness of an experimental platform such as DeVIDE, presented in chapter 2. Being able to test new ideas, fine-tune parameters and examine results in a flexible way greatly facilitates the development process.

Now that a viable segmentation technique exists, the process should be streamlined. At the moment, there are a number of parameters that need to be adjusted for every new dataset, a time-consuming process. In addition, the interface should be simplified and the protocol formalised in order to make it possible for clinicians to perform the segmentation independently of a visualisation or image processing expert.

The techniques described in this chapter form an important component of an evolving system for pre-operative planning and intra-operative guidance for shoulder replacement. As we apply the system in more studies and eventually on patients, the segmentation technique will be refined and validated on more datasets.

Finally, we wish to study the possibilities of shoulder skeleton segmentation from high-resolution MRI data where the acquisition parameters have been optimised with this in mind.

---

<sup>4</sup>Dr Vannier is head of the radiology department of the University of Iowa and member of the board of directors of Vital Images. The quote is from his keynote at SPIE Medical Imaging 2004.



---

## Real-time visual feedback for Transfer Function Specification in Direct Volume Rendering

---

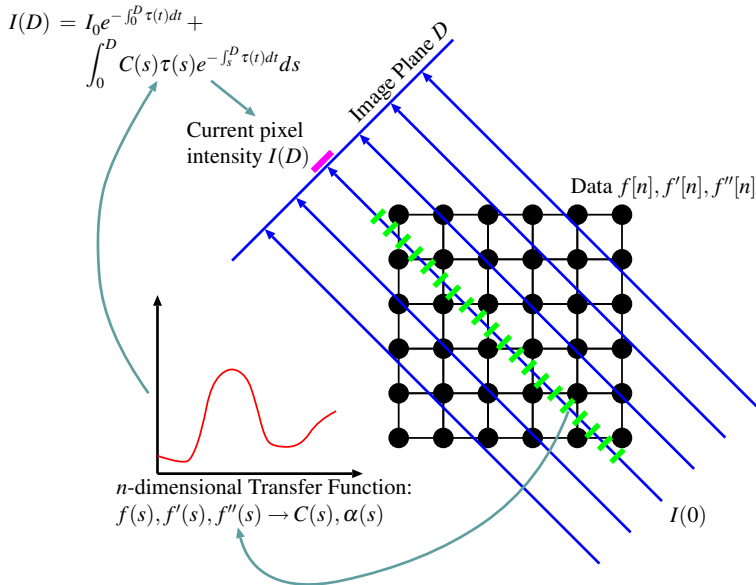
This chapter, based on [10] and [9], presents two related techniques for supplying meaningful visual feedback during direct volume rendering transfer function specification.

Both of the techniques generate voxel-registered previews of the expected volume rendering. Because of the registration, this preview can easily be alpha-blended with a grey-scale image of the data that is being volume rendered. In this way, the user gets real-time visual feedback on her transfer function specification with regards to *both* the expected composited optical properties *and* the “fidelity” (how closely the rendering matches the original data) of the resulting rendering. This facilitates and speeds up the user-driven transfer function specification process.

The first technique makes use of a simple voxel lookup and low opacity compensation scheme to generate the sliced-based previews. The second technique uses meta-data calculated during a pre-processing step to generate interactively an approximate volume rendering that is voxel-registered with a single user-selected slice.

### 5.1 Introduction

Direct Volume Rendering [45, 53], or DVR, is an important and useful technique for visualising structures in volumetric data. However, the difficulty in specifying the DVR transfer function [64] can be prohibitive to the deployment of this technique in practical visualisation applications. The transfer function is a critical component of the volume rendering process that specifies the relation between scalar data (e.g. computerised tomography Hounsfield



**Figure 5.1:** Illustration of 2D raycasting showing the role of the transfer function.

units), as well as derivative values (e.g. the gradient volume of an MRI dataset), and optical characteristics (e.g. colour and opacity).

Figure 5.1 illustrates raycasting with a 2D volume that shows the role of the transfer function. Raycasting is an example of image-order direct volume rendering, but the role of the transfer function is the same in other direct volume rendering techniques. For each pixel on the image plane, a ray is traversed through the volume. The volume is sampled at regular intervals along this ray. At each point, the volume scalar value is interpolated. This interpolated scalar value has no inherent optical properties, such as colour and opacity, so these have to be looked up by making use of the transfer function. For each of these interpolated scalar values, the transfer function assigns colour and opacity. These can then be shaded by using for example the Phong shading model. All looked up points along the ray are substituted in the volume rendering integral shown at the upper left of the illustration. This results in an accumulated colour for the pixel corresponding to the ray that has been traversed.

In general, this transfer function, often consisting of a number of separate 1D functions transforming volume scalar and derivative values to colour components and opacity, has to be specified by the user. In other words, the user has to decide on which colours and translucency to assign to each different voxel scalar value, or material type, in a data volume. This is a challenging task.

Our work attempts to give the user full control over the transfer function specification and to supply real-time visual feedback in such a fashion that the system's reliance on the user's visualisation expertise is minimised and the leverage of the user's application-specific (e.g. clinical) knowledge is maximised. The visual feedback is supplied either with a simple per-voxel lookup and opacity compensation technique or with a special volume rendering preview technique.

The preview technique has been designed to give a mathematically defensible indication of the composited optical properties in the resultant volume rendering. Both techniques yield explicitly data-registered feedback, meaning that the fidelity of the resultant rendering (and the transfer function) can be checked. On a higher level, the one-to-one correspondence between data and volume rendered structures is accentuated.

In section 5.2 we discuss existing methods for finding transfer functions. Sections 5.3 and 5.4 document respectively our simple data-registered feedback scheme and the more complex and accurate previewing scheme. In section 5.5 we show some examples that demonstrate the effectivity of these two methods. We summarise our work and point out possible avenues for future research in section 5.6.

## 5.2 Related Work

In [64], the statement is made that finding good transfer functions is one of the top ten problems in volume visualisation. Consequently, much effort has recently been spent on improving this situation. Existing schemes range from fully manual to fully automatic techniques for finding transfer functions.

Probably the oldest method of finding a transfer function is by trial and error. This usually involves manipulating a transfer function whilst periodically checking the resulting volume rendering. If special volume rendering hardware that can do this at interactive rates is not available, this can be a very laborious and time-consuming process.

Another very interesting technique that is based on the design galleries paradigm [49] generates many volume renderings simultaneously, each representing a different configuration of the transfer function. The user selects the renderings that satisfy her requirements and thus implicitly optimises the transfer function. The considerable challenges here are to generate automatically the different transfer functions that are going to generate a wide spread of dissimilar output renderings [29] and to present these renderings to the user in an effective way.

As potentially hundreds of different renderings have to be made, this technique does rely on fast rendering hardware being available to reach its full potential as an interactive method.

Also take into account that many of the optimised software volume rendering techniques such as shear-warp factorisation [42] require pre-calculated transfer function lookups, which makes them less applicable to this problem where the transfer functions are being continuously modified.

The work of König and Gröller combines elements of the design galleries and trial-and-error techniques with the required use of real-time raycasting hardware [40]. They present

transfer function specification as a simplified three step process: First the user indicates scalar ranges of interest, then she assigns colours to these ranges and finally she assigns opacities. Numerous feedback renderings are performed during this process in order to guide the user's choices. This technique simplifies the specification to quite an extent.

Bajaj's contour spectrum [17] consists of metrics that are computed over a scalar field. This spectrum, presented as a user interface component, can be used to assist the user in finding a suitable transfer function. It offers an alternative and condensed way of examining the volume of data that reveals global characteristics which can be very helpful in creating a transfer function.

The semi-automatic method of Kindlmann [37] is a highly-regarded technique for generating transfer functions from volumetric data. This method makes the reasonable assumption that the features of interest in the data are the boundary regions between areas of relatively homogeneous material.

Bajaj's and Kindlmann's methods are designed for creating transfer functions with minimal or no user-interaction. According to our references, they do not address the problem of providing meaningful feedback during manual transfer function specification or fine-tuning. Because our method focuses on providing meaningful and fast feedback, it does not replace these schemes, but could be used very profitably in augmenting them.

The design galleries approach, trial-and-error scheme and their derivatives can be considered as being focused on providing feed-back in a user-driven transfer function specification process and this is where our scheme excels.

Even if interactive volume rendering facilities are available that can cope with continuous changes in the transfer function, it is often difficult to identify the fuzzy structures in a feedback volume rendering that can result from a particular transfer function and accurately relate them to the structures in the data that are being rendered. This makes it very difficult to quantify the impact that a small modification to the transfer function has.

It is desirable to have some kind of directed search process where the user is gradually but purposefully moving towards an optimal transfer function. The difficulty in quantifying the impact of a transfer function change, especially with regards to structures visible in the unprocessed data, hinders this process and necessitates a high level of visualisation experience from the user. Our method yields explicit and real-time feedback on the relationship between the structures in the data that are being rendered and the structures that can be expected in the resultant volume rendering.

### 5.3 Simple data-registered feedback

In this section, the first and more simple feedback technique is explained.

A common way of examining 3D data-sets, especially in medical applications, is to view the grey-scale representation of a voxel-thick slice of the data. The position and orientation of the slice can be changed interactively so that the user has the impression of "moving" through the data. Clinical users are cognitively well-adapted to recognise structures in data-sets that are represented in this fashion.

In addition to allowing the user to examine the data in this fashion, she is also granted access to the transfer function and is able to manipulate it directly by adjusting five piecewise linear functions representing the hue, saturation, value, scalar opacity and gradient opacity transfer function components. Our method is not constrained to this particular type of transfer function interaction. Far simpler interaction schemes can also be employed.

Throughout this interaction, all voxels in the current slice are transformed with the current transfer function and a new slice is created by applying straight-forward mappings on the transformed voxels. Henceforth, this new slice will also be referred to as the overlay slice.  $R_t, G_t, B_t$  and  $A_t$  refer to the red, green, blue and alpha characteristics that have been looked up for a given voxel with the current transfer function. The mapped optical characteristics in the overlay slice,  $M = \langle R_m, G_m, B_m, A_m \rangle$ , are a function of the transformed voxel optical properties, i.e.  $M = f(\langle R_t, G_t, B_t, A_t \rangle)$ .

The new slice is super-imposed on a grey-scale representation of the current slice by alpha-blending. If the grey-scale representation is  $X = \langle R_g, G_g, B_g, A_g \rangle$ , the result of the alpha-blending is, as per usual:

$$\langle (1 - A_m)R_g + A_mR_m, (1 - A_m)G_g + A_mG_m, (1 - A_m)B_g + A_mB_m, (1 - A_m)A_g + A_m \rangle$$

Instead of seeing a grey-scale slice of the data volume, the user sees a coloured slice. This coloured slice shows an estimation of how the materials present in the slice will appear in a volume rendering made with the current transfer function. The coloured slice can be moved through the dataset as per usual. Colours are updated in real-time as the user manipulates the transfer function, giving real-time feedback on the suitability of the changes.

Several different mappings can be used, each accentuating some aspect of the transfer function in the feedback process:

- With  $M = \langle A_t, 0, 0, A_t \rangle$ , only the opacity is visualised, but both as red intensity and as blending opacity. At high opacities  $A_t$ , the red overlay is brighter and also obscures most of the grey-scale slice view with which it is blended.
- Another way of visualising only the opacity is by making use of  $M = \langle 0, 1.0, 0, A_t \rangle$ . The intensity of the green overlay is kept at a maximum, but the opacity is visualised by the amount of the grey-scale slice that is visible through the new super-imposed slice. This mapping yields better results than the previous one at low opacities due to the fact that the green intensity is not affected by opacity.
- Both the colour and opacity are visualised with  $M = \langle R_t * A_t, G_t * A_t, B_t * A_t, A_t \rangle$ . The colour is scaled with opacity, so that the opacity is fed back as both blending transparency and colour intensity. Remember that the colour of a voxel is scaled with the opacity during raycasting. However, because our scheme does not accumulate optical properties as is the case in volume rendering, this results in low opacities decreasing and potentially neutralising colour intensities.

- With

$$M = \langle R_t * A_t, G_t * A_t, B_t * A_t, 1 - e^{-\frac{A_t}{c}} \rangle \quad (5.1)$$

we try to emulate what happens with low opacities in a raycasting. Recall that the absorption term in the direct volume rendering integral is

$$I(D) = I_0 e^{-\int_0^D \alpha(t) dt}$$

where  $I(D)$  is the light that reaches the eye (via the light-attenuating volume),  $I_0$  is light that is present at the opposite side of the volume, and  $\alpha(t)$  is the instantaneous opacity [51, 41]. Thus, during raycasting, low opacities accumulate very rapidly whereas opacities that are closer to unity accumulate more slowly. The last component in equation 5.1 emulates this behaviour. The time-constant  $\tau$  determines how rapidly the component increases with increasing  $A_t$ .<sup>1</sup> Figure 5.2 shows the behaviour of this compensation for  $\tau = 0.25$ .

- The mapping  $M = \langle 0, 1.0, 0, 1 - e^{-\frac{A_t}{\tau}} \rangle$ , combines the maximum intensity monochromatic overlay slice with the low opacity compensation. This is a very good way of working with low opacities, as they are compensated for *and* they are not allowed to affect colour intensity.
- By ignoring the scaling effect of opacity on the colour intensity and by compensating for low opacities,  $M = \langle R_t, G_t, B_t, 1 - e^{-\frac{A_t}{\tau}} \rangle$  qualitatively yields the best all-round results and most closely approximates the resultant volume rendering. This mapping will be referred to as the “unscaled colour and compensated opacity mapping”.

Once super-imposed, the new slice clearly shows the expected visibility and optical properties of structures in the final volume rendering. Because the preview is voxel-registered with the data, the one-to-one correspondence between the grey-scale representation of the raw data and volume rendered structures is clearly visualised and the fidelity of the transfer function can be checked.

As the user makes small changes in the transfer function, the effects of these small changes are immediately reflected in the preview. In effect, a “soft” segmentation is performed on the volume data with the specified transfer function in real-time. When the super-imposed segmentation agrees with the user’s comprehension of the structures in the data, the transfer function has been implicitly optimised and the resulting volume rendering will also reflect the user’s expectations.

## 5.4 Data-registered predictive volume rendering

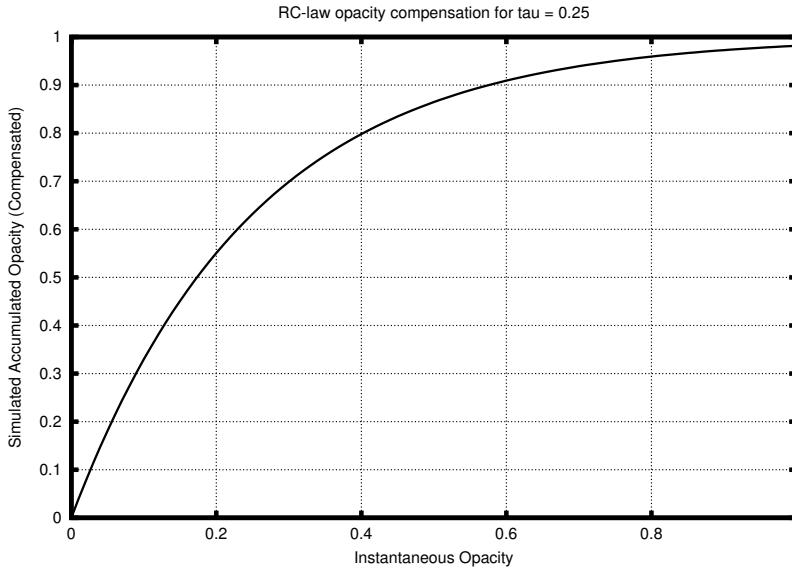
In this section, a more involved technique is described whereby one can predict the results of the volume rendering on a per slice basis.

The central idea of this technique is to predict the optical result of casting a ray through a voxel and parallel to the view direction, through *all* the material represented by that particular

---

<sup>1</sup> $\tau$  in this context, the parameter-less time-constant, should not be confused with the extinction coefficient  $\tau(s)$ .





**Figure 5.2:** Behaviour of  $f(A_t) = 1 - e^{-\frac{A_t}{\tau}}$  for  $\tau = 0.25$ .  $A_t$  is the instantaneous opacity,  $f(A_t)$  is a compensated opacity that can be used in sliced-based previews.  $\tau$  in this context is a parameter-less time-constant.

voxel. The term “material” is defined as the set of optical properties assigned to a given scalar value by the DVR transfer function.

The amount of a specific material intersected by a ray can be estimated for each voxel position by making use of fast run-time processing of pre-calculated per-ray scalar frequency distributions (i.e. histograms of scalar values). The estimated amount, measured as a number of voxels, can be used in a simplified form of the volume-rendering integral to predict the optical outcome.

This prediction is done for all voxels in a particular slice of the raw data and all predicted results are then alpha-blended with that slice.

What the user sees is an estimate of what the volume rendering would look like with her currently configured transfer functions, super-imposed on a grey-scale slice image of the raw data. When the user makes any changes to the transfer functions or when she “moves” (by changing the slice position) through the data, the visual update can be done in real-time, as relatively little processing is required.

This method is based on the assumption that the data consists of a finite number of contiguous volumes of optically homogeneous material, as is often the case with medical datasets. The closer the truth is to this assumption, the more accurate the preview is.

In the following sub-sections, we will first derive the simplified form of the volume-rendering integral, then explain the method we use to predict the expected amount of homogeneous material in the path of a ray and finally show how this has been implemented.

### 5.4.1 Mathematical Preliminaries

We start with the well-known absorption and emission model volume-rendering integral [51, 41]. The light intensity at the eye (position  $D$ ) is found by integrating from  $s = 0$  at the opposite side of the volume that is being rendered to  $s = D$  at the eye:

$$I(D) = I_0 e^{-\int_0^D \tau(t) dt} + \int_0^D C(s) \tau(s) e^{-\int_s^D \tau(t) dt} ds \quad (5.2)$$

where  $I_0$  is the light intensity at the opposite side of the volume,  $\tau(x)$  is the extinction coefficient (“material density”) at position  $x$  and  $C(x)$  is the emitted colour at position  $x$ .

This relation models absorption (first term) and emission (second term). In the absorption term, already present light is attenuated by the density of the volume material obscuring it from the eye. In the emission term, light reflected and/or emitted by the material at any point is weighted by the material’s density at that point and then attenuated by the density of all material between that point and the eye.

The emitted and/or reflected light  $C(s)$  is simulated with the Phong shading model [65]:

$$\begin{aligned} I_\lambda(s) = & I_{a\lambda} k_a O_{d\lambda}(s) + \\ & I_{p\lambda} [k_d O_{d\lambda}(s) (\mathbf{N}(\mathbf{s}) \cdot \mathbf{L}) + \\ & k_s (\mathbf{N}(\mathbf{s}) \cdot \mathbf{H})^n] \end{aligned} \quad (5.3)$$

The  $\lambda$  subscript denotes wavelength dependence so that we are not limited to a specific colour model. For example, in the case of RGB-raycasting, we will calculate respectively  $I_R(s)$ ,  $I_G(s)$  and  $I_B(s)$  and these would then represent  $C(s)$ .

The symbols in equation 5.3 are as follows:

$I_{a\lambda}$	ambient light intensity
$k_a$	ambient reflection coefficient
$O_{d\lambda}(s)$	object diffuse colour at $s$
$I_{p\lambda}$	point light source intensity
$k_d$	diffuse reflection coefficient
$\mathbf{N}(\mathbf{s})$	surface normal at $s$
$\mathbf{L}$	direction of light source
$k_s$	specular reflection coefficient
$\mathbf{H}$	vector halfway between view and light source vectors
$n$	specular reflection exponent

We make use of this model in our direct volume rendering preview to determine  $C(s)$  (as it is in the raycasting implementation), but we refer the reader to the textbooks for a detailed discussion of Phong shading.

It is desirable to be able to evaluate the volume rendering integral numerically for practical application. Discretising equation 5.2 with distance step  $\Delta s$  yields:

$$I(D) = I_0 \prod_{i=1}^{D/\Delta s} e^{-\tau(s_i)\Delta s} + \sum_{i=1}^{D/\Delta s} C(s_i) \tau(s_i) \Delta s \prod_{j=i+1}^{D/\Delta s} e^{-\tau(s_j)\Delta s}$$

Without loss of generality, we can substitute  $\tau(s_i)\Delta s$  with the opacity of the  $i$ th segment along the ray, i.e.  $\alpha(s_i)$  and reverse the order of summation and multiplication (i.e.  $s = 0$  at the eye and  $s = D$  at the opposite side of the volume):

$$I(0) = I_D \prod_{i=0}^{D/\Delta s-1} e^{-\alpha(s_i)} + \sum_{i=0}^{D/\Delta s-1} C(s_i)\alpha(s_i) \prod_{j=0}^{i-1} e^{-\alpha(s_j)} \quad (5.4)$$

Usually a two-term Taylor series approximation of the exponential terms,  $e^{-\alpha(s_j)} \approx 1 - \alpha(s_j)$ , is employed to yield a computationally efficient form that can be incrementally calculated (an exponential term can be updated with a single multiplication at each discretisation step) as a ray is being traversed. However, we will deviate from the usual derivation and continue with the more exact discrete form in equation 5.4.

If a ray were to pass through a known number  $N$  of voxels such that the voxels have constant object diffuse colour  $O_{d\lambda k}$  and they have constant opacity  $\alpha_k$ , i.e. the  $N$  voxels are optically identical, and we assume that

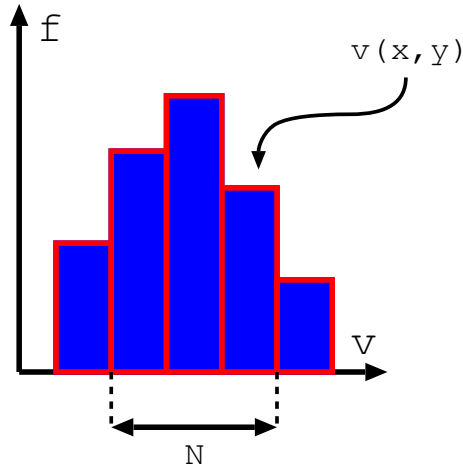
1. the ray sampling distance  $\Delta s$  is of the same dimension as a voxel, i.e.  $D/\Delta s = N$  and
2. surface normals are constant for the  $N$  voxels

then equation 5.3 would yield constant  $C_k$  and we could simplify equation 5.4 to yield a closed-form solution:

$$\begin{aligned} I(0) &= I_D \prod_{i=0}^{N-1} e^{-\alpha_k} + \sum_{i=0}^{N-1} C_k \alpha_k \prod_{j=0}^{i-1} e^{-\alpha_k} \\ &= I_D e^{\sum_{i=0}^{N-1} -\alpha_k} + C_k \alpha_k \sum_{i=0}^{N-1} e^{\sum_{j=0}^{i-1} -\alpha_k} \\ &= I_D e^{-N\alpha_k} + C_k \alpha_k \sum_{i=0}^{N-1} e^{-i\alpha_k} \\ &= I_D e^{-N\alpha_k} + C_k \alpha_k \frac{1 - e^{-N\alpha_k}}{1 - e^{-\alpha_k}} \end{aligned} \quad (5.5)$$

## 5.4.2 Algorithm

An orthogonal previewing direction is selected by the user. For each voxel position in the plane normal to the viewing direction, a frequency distribution of voxel scalar values is calculated for all voxels intersected by a ray through that voxel position and parallel to the viewing direction. In our case, the frequency distribution is a listing of the frequency of occurrence of voxel scalar values as a function of ranges (classes) of scalar values. The number of constant-size ranges (classes) is chosen so that the average number of voxels per class remains small (e.g. by choosing the number of classes to be a fifth of the number of voxels in



**Figure 5.3:** An illustration of the frequency distribution for a single  $(x, y)$  position. The distribution bin containing voxel scalar value  $v(x, y)$  is found with a binary search. Subsequently, bins to the left and right of the found bin are merged according to the similarity of the optical properties they represent. After merging, the number of estimated voxels  $N$  can be calculated by adding up the number of voxels in all merged bins.

the orthogonal previewing direction, we can expect an *average* of five voxels per class) and the classes are maintained in a sorted order determined by the scalar values at their limits. A gradient volume is also calculated. All of this is done once per previewing session.

Whenever a new super-imposable preview slice has to be generated (i.e. when the user modifies a transfer function or selects another slice position), the following procedure, illustrated in figure 5.3, is executed: For each voxel in the current view-direction-orthogonal slice, its scalar value is transformed with the current transfer functions. The frequency distribution corresponding to its position is selected and the class to which this voxel belongs is found with a binary search.

Recall that these classes have been chosen to contain on average a small number of voxels. In addition, these are classes of *scalar* voxel values, not optical properties. However, by merging a group of neighbouring classes if the optical properties that they represent are equal (or very similar) we can obtain a much better estimate of the number of voxels in the path of the ray that have this set of optical properties. The merging is performed as follows:

The class that we have just found is merged with its neighbouring class to form a larger class *if* the value corresponding to the farthest edge of the neighbouring class, after having been transformed by the current transfer functions, is optically equal (or very similar) to the current voxel's transformation. Four-dimensional Euclidean distance in the colour-opacity space (in our case hue, saturation, value and scalar opacity) is used as a metric for optical similarity. This merging is continued until a neighbouring class's edge is optically distinct from the current voxel's transformation. Merging is performed in both directions.

After merging, we have an estimate of how many adjacent (spatially as well as in a scalar

sense) voxels, with optical characteristics equal (or very similar) to our selected voxel, we can expect to be intersected by a cast ray that passes through the current voxel position and is parallel to the viewing direction. We will use this estimate as  $N$  in equation 5.5 to calculate the shaded light intensity that would result from a ray cast through just this optical material. As explained in section 5.4.1, we use the normal of the selected voxel to compute the shading for the ray that passes through it.

After having done this for each voxel in the current slice, the resulting image can be superimposed on a slice of a grey-scale image of the original data by using  $1 - e^{-N\alpha_k}$  as the opacity (representing absorption) and the second term of equation 5.5 as the colour (representing emission and reflection).

To summarise: Whenever the slice is changed or the transfer function is modified, each voxel in the current slice is transformed, bin merging is performed on the frequency distribution corresponding to its position and the predicted optical characteristics for that position are calculated. It is obvious that the processing overhead is highly dependent on the amount of merging that has to be done as the rest of the computations can be performed in logarithmic time.

Storage requirements are modest when the algorithm has access to the data-structures required by the final raycasting itself, for example the gradient volume. In our implementation, the previewing requires, per previewing direction, a fifth of the data-set size for the frequency distributions. This requirement is dependent on the number of frequency distribution classes that an implementation deploys.

### 5.4.3 Implementation

The DSCAS1 platform<sup>2</sup> [8] was used to create test implementations of both the described methods as it offers the necessary data structures and communication and rendering functionality.

By using the GUI, the user can set up objects for the data itself, the transfer function, the slice-viewer and the DVR previewer. After having connected these objects the necessary pre-processing will be done and the user can start manipulating the transfer function by interacting with hue, saturation, value, scalar opacity and gradient opacity functions by making use the UI-element shown in figure 5.4. UI-elements for adjusting all shading parameters are also supplied.

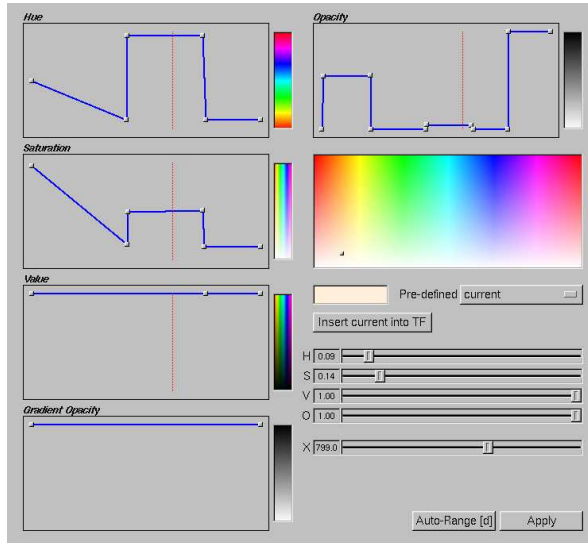
Throughout this process, the UI remains completely interactive and the DVR preview is updated in real-time. Once a satisfactory preview has been created the data can be volume rendered with the resultant transfer function.

## 5.5 Results

We have applied the feedback schemes to make transfer functions for volume rendering a clinical CT-dataset of a shoulder as well as the tooth and sheep datasets used in the Transfer

---

<sup>2</sup>DSCAS1 was an ancestor of the DeVIDE platform described in chapter 2.



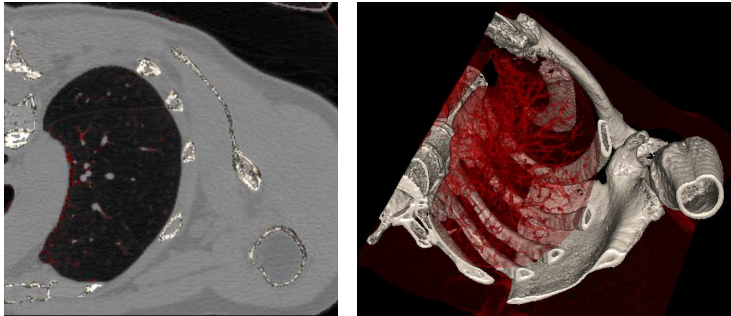
**Figure 5.4:** Our implementation of a DVR transfer function editing widget. The user can add and manipulate an arbitrary number of points in the piecewise continuous lines that represent the hue, saturation, value, scalar opacity and gradient opacity transfer functions. Our method does of course extend to other more complex transfer function representations, e.g. Bézier curves.

#### Function Bake-Off [64].

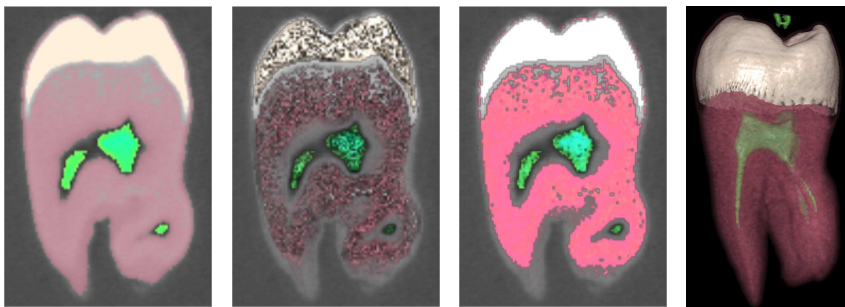
To appreciate the efficacy of the feedback methods one actually needs to interact with it or view a video recording which displays the interactivity of the preview. However, in the following figures one can clearly see the similarity between the preview and the raycasting resulting from the same transfer function as well as the subjective quality of the resultant raycasting. Keep in mind that the preview is updated in real-time as the user interacts with the transfer function and moves through the data. It can be viewed overlaid on a grey-scale image of the original data as well as by itself. Due to the flexible nature of the implementation, an arbitrary number of preview slice positions can be viewed from all orthogonal view directions simultaneously, overlaid and non-overlaid.

Figure 5.5 shows a preview and volume rendering of clinical CT-data of a shoulder. Figure 5.6 shows (from left to right) the simple feedback method, a preview with shading, a preview without shading (i.e. equation 5.3 with  $I_{a\lambda} = 1$ ,  $k_a = 1$  and  $k_d = k_s = 0$ ) and a volume rendering of industrial CT-data of a tooth. Figure 5.7 shows (from left to right) the simple feedback method, a preview with shading, a preview without shading and a volume rendering of MRI-data of a sheep heart.

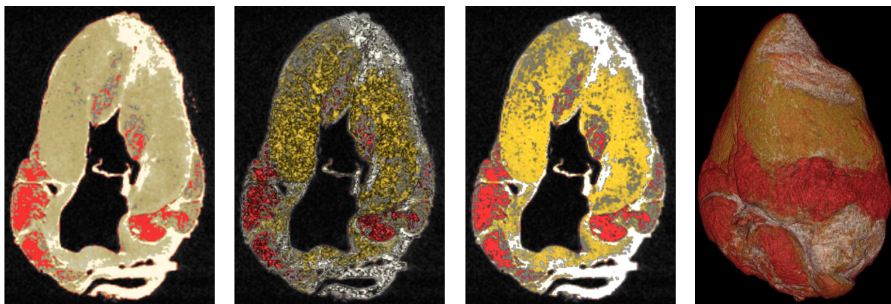
The data can be investigated with a point probe to get some idea of the different scalar values involved. By clicking on any point in any slice the scalar voxel value at that point is returned. In all cases, setting up an initial transfer function and then converging on a “good” transfer function takes less than ten minutes of session time of an experienced operator.



**Figure 5.5:** Preview and direct volume rendering of shoulder CT-data showing bony and bronchial structures. See colour figure C.5.



**Figure 5.6:** Feedback with simple method, previews with and without shading and direct volume rendering of industrial CT-data of tooth. See colour figure C.6.



**Figure 5.7:** Feedback with simple method, previews with and without shading and direct volume rendering of MRI-data of sheep heart. See colour figure C.7.

## 5.6 Conclusions

We have presented two methods that provide real-time visual feedback during DVR transfer function specification. The feedback is explicitly data-registered and is usually shown superimposed on the corresponding slice of a grey-scale image of the “raw” data. It changes in real-time as the user manipulates the DVR transfer function. The user is also allowed to continue investigating the results by continually changing the current slice position, i.e. “moving” through the data.

The first method is based on a straight-forward mapping of the transformed voxels in the current slice and the use of a RC-law function to model the accumulation of low opacities. The second method is based on a simplified form of the volume rendering integral with the optional use of Phong shading.

These schemes have some advantages which differentiate them from the other feedback-based methods:

- The preview is sliced-based and data-registered with the data that is being rendered, so it can be overlaid on a grey-scale image of the data and the user can easily relate structures in the volume rendering to structures in the “raw” data. Because of this, the fidelity of the resulting volume rendering can easily be checked and the (clinical) user’s experience in working with slice-based medical data is leveraged.
- The preview is very fast so the effect of small changes are instantly fed-back to the user. This aids tremendously in effectively converging on a good transfer function.
- The method is easy to implement.
- The method does not require special hardware.

The methods that we have developed give a very good data-registered preview of a potential volume rendering and is invaluable in the transfer function specification process.

We have also done some informal experimentation with students in a visualisation course: as part of a direct volume rendering exercise, a similar slice-based previewing technique is explained and the students are instructed to make use of the technique in order to find suitable transfer functions. In the two years that we have been doing this, the feedback from the students has been unanimously positive. In spite of the slow software raycasting implementation used, students find that the process of finding a suitable transfer function is greatly facilitated.

In spite of the fact that the second, more complex, method is more mathematically defensible, it turns out that the first straight-forward method, with the RC-law low opacity accumulation compensation, yields satisfactory results in most practical cases.

There is definitely room for improvement in the way in which the user edits a transfer function (and this is so in most current volume rendering applications). An interesting avenue of future research would be the investigation of different feedback-based interaction techniques.



## **Acknowledgements**

This research is part of the DIPEX (Development of Improved endo-Prostheses for the upper EXtremities) program of the Delft Interfaculty Research Center on Medical Engineering (DIOC-9).

The clinical CT-images of the shoulder were provided by R.E. van Gelder of the Academic Medical Center of Amsterdam.



---

## ShellSplatting: Interactive Rendering of Anisotropic Volumes

---

This chapter is a slightly modified version of our ShellSplatting article [11]. Figure 6.1 was added to illustrate the shell rendering data structures.

### **Abstract**

This work presents an extension of shell rendering that is more flexible and yields higher quality volume renderings. Shell rendering consists of efficient data-structures and methods to manipulate and render structures with non-precise boundaries in volume data. We have updated these algorithms by creating an implementation that makes effective use of ubiquitously available commercial graphics hardware. More significantly, we have extended the algorithm to make use of elliptical Gaussian splats instead of straight-forward voxel projection. This dramatically increases the quality of the renderings, especially with anisotropically sampled volumes. The use of the graphics hardware alleviates the performance penalty of using splats.

### **6.1 Introduction**

Volume visualisation can be performed in three ways: rendering two-dimensional slices of the volume, rendering surfaces that have been extracted from the volume and direct volume rendering [14].

An extracted surface is usually an isosurface and is approximated as a triangulated mesh for accelerated rendering with modern polygon graphics hardware. This method assumes that

extractable isosurfaces are present in the data and that these isosurfaces correctly model the structures in the volume [53].

Direct volume rendering [45, 21] (DVR) allows us to visualise structures in the data without having to make decisions about the precise location of object boundaries by extracting polygonal isosurfaces. Instead, we can define multi-dimensional transfer functions that assign optical properties to each differential volume element and directly visualise structures on the grounds of this transformation.

Shell rendering [85, 86], which can be seen as a hybrid of surface and direct volume rendering, was proposed as a fast volume visualisation method in the early nineties. It had significant advantages at that time: it was an extremely fast software algorithm that also required far less memory than competing volume visualisation algorithms. As recently as 2000, software shell rendering was still found to be faster than hardware assisted isosurface rendering [26]. Originally it supported only parallel projection but work has been done to extend it to perspective projection [16]. In this chapter, we will consider only parallel projection.

However, this object-order method makes use of simple voxel projection to add a voxel's energy to the rendered image. In addition, texture-mapping, shading and compositing hardware has become common-place. These two facts have led us to extend shell rendering by translating it to a hardware-accelerated setting and to allow voxels to contribute energy to the rendered image via a splat-like elliptical Gaussian. This makes possible interactive high-quality renderings of anisotropically sampled volumes.

In this chapter we present this extension that we have dubbed ShellSplatting. The algorithm generates higher-quality renderings than shell-rendering and does this more rapidly than standard splatting. It retains all advantages of the shell rendering data-structures. We also present a straight-forward way to calculate the splat projections that accommodates anisotropically sampled volumes at no extra speed or memory cost.

Section 6.2 supplies information about volume visualisation, focusing on splatting and shell rendering and the development of this work. In section 6.3 our algorithm is documented. Section 6.4 contains visual results as well as comparative timings. In section 6.5 we detail our conclusions and mention possible avenues for future work.

## 6.2 Related Work

Much work has been done to improve the quality and the speed of splatting. Two early papers focusing on hardware-assisted implementations of splatting are by Laur and Hanrahan [43] and Crawfis and Max [19]. The former represents each splat as a collection of polygons whilst the latter makes use of a single texture-mapped polygon, thus utilising available graphics hardware for both modulation and compositing.

A more recent contribution is that of Ren *et al* [69], who extend EWA (Elliptical Weighted Average) Surface Splatting [103] to an object-space formulation, facilitating a hardware-assisted implementation of this high-quality surface point rendering method. There are obviously also similarities between our method for finding an object-space pre-integrated Gaussian reconstruction function and the logic employed by EWA surface and volume splat-

ting [104] and the original EWA filtering [30].

An advantage of our work over more recent hardware-accelerated splatting methods that make use of vertex and pixel shaders as well as other hardware-assisted direct volume rendering schemes [70, 39] that utilise features such as register combiners and dependent textures, is the fact that ShellSplatting works on any device with two-dimensional texture-mapping and compositing facilities. This makes the algorithm practically hardware-independent whilst still enabling it to profit from advances in newer generations of graphics accelerators.

The idea of combining splatting with a more efficient data-structure for storing selected voxels and traversing them more rapidly is also not new. Yagel *et al* present a *Splat Renderer* which relies on the concept of a *fuzzy voxel set* that consists of all voxels in the dataset with opacities above a certain threshold and stores these voxels in a compact data-structure [101]. Mueller *et al* accelerate volume traversal for their splatting advances by making use of list-based data-structures and binary searches to find voxels within certain iso-ranges rapidly [55]. The QSplat point rendering system makes use of a hierarchy of bounding spheres for rapid volume traversal and adaptive level-of-detail rendering [71]. Crawfis employs a list of coordinates sorted by corresponding scalar value so that voxels with a given scalar value can be rapidly retrieved and splatted [20]. Because all voxels are splatted with the same colour, splatting order is not important.

The work by Orchard and Möller [57] is probably the closest to ShellSplatting. They devise a 3D adjacency data structure that stores only voxels with opacity above a certain threshold and allows rapid back-to-front traversal. In this chapter they mention the possible improvement of eliminating voxels that are surrounded by opaque material.

Our algorithm, due to its use of the shell-rendering data-structures, not only eliminates voxels with opacity beneath a certain threshold, but also all voxels that are surrounded by non-transparent material. In addition, it automatically discards voxels at render time which are occluded from the observer by non-transparent material.

A more subtle but important difference is that ShellSplatting functions anywhere on the spectrum between a voxel-based surface rendering algorithm and a complete volume rendering algorithm. Its position on this spectrum is controlled by two algorithm parameters that will be explained in section 6.2.2.

In the following subsections we explain splatting and shell-rendering in order to facilitate understanding of the algorithm description in section 6.3.

### 6.2.1 Splatting

Splatting is an object-order (i.e. forward projection) direct volume rendering method that treats an N-dimensional sampled volume as a grid of overlapping N-dimensional volume reconstruction function kernels, often Gaussians, weighted with voxel densities. These weighted kernels are projected onto the image plane to form “splats” that are composited with affected pixels [97]. In this way, splatting approaches the problems of volume reconstruction and rendering as a single task.

In original splatting [97], the volume is traversed from front to back or from back to front. Centred at each voxel position a reconstruction kernel is integrated along the view axis

to form a pre-integrated two-dimensional kernel footprint. The kernel footprint is used to modulate the looked-up and shaded voxel optical characteristics (colour and opacity) of that voxel and projected onto the image plane where it is composited with the affected pixels.

Different compositing rules are used for front-to-back and back-to-front traversals: respectively Porter and Duff's [66] **under** and **over** operators. In the latter case, the splat kernel projection is composited with all pixels at positions  $p$  of the image buffer  $I$  that are affected by the kernel's projection as follows:

$$I_\lambda(p)I_\alpha(p) = S_\lambda(p)S_\alpha(p) + I_\lambda(p)I_\alpha(p)(1 - S_\alpha(p)) \quad (6.1)$$

Subscript  $\lambda$  represents the colour band, e.g. red, green or blue.  $I_\lambda$  and  $I_\alpha$  are the colour and opacity currently in the image buffer.  $S_\alpha(p)$  and  $S_\lambda(p)$  represent the modulated and shaded opacity and colour of the kernel projection at pixel position  $p$ . Note that, although this appears similar to alpha-blending, it is quite different due to the opacity pre-multiplication. There seems to be some confusion in literature concerning this compositing rule. The opacity  $I_\alpha$  in the image buffer is updated as follows:

$$I_\alpha(p) = S_\alpha(p) + I_\alpha(p)(1 - S_\alpha(p)) \quad (6.2)$$

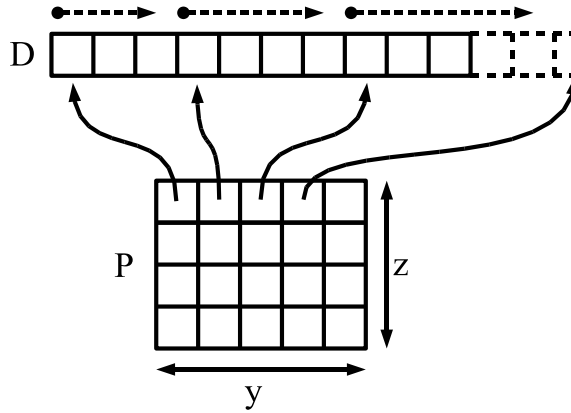
This compositing of splats on the image buffer approximates the colour and opacity integration of the direct volume rendering integral.

The use of pre-integrated reconstruction kernels causes inaccuracies in the composition as each kernel is independently integrated and not in a piecewise fashion along with other kernels in the path of a view ray. This can result in colour-bleeding of obscured objects in the image. Westover proposed first performing compositing of piece-wise kernel integrations into volume axis-aligned sheet-buffers [96] and then onto the image buffer to alleviate this affect. However, the axis-aligned sheet-buffering resulted in sudden image brightness changes, also known as "popping", during rotation. Mueller and Crawfis introduced image-aligned sheet-buffers to eliminate this problem [54].

## 6.2.2 Shell Rendering

Shell rendering considers volumes as fuzzily connected sets of voxels and associated values. These sets are known as shells and the member voxels as shell voxels. In short, all voxels with opacities higher than a configurable lower opacity threshold  $\Omega_L$  and with at least one neighbour with opacity lower than the configurable higher opacity threshold  $\Omega_H$  are part of the shell.  $\Omega_L$  is the minimum opacity that will contribute to the volume rendering. A voxel with opacity  $\Omega_H$  or higher occludes any voxels directly behind it, which is why we exclude all voxels that are surrounded by these high opacity voxels. A compact binary neighbourhood code that indicates *which* of a voxel's neighbourhood voxels have  $\Omega_H$  or higher opacities is also stored for each shell voxel.

If  $\Omega_L = \Omega_H$ , the shell is a one voxel thick object boundary that is a digital approximation of the isosurface with iso-value  $v = \Omega_L$ . By increasing the difference between the two thresholds, we increase the number of voxels in the shell, until, in the limiting case, the set of shell



**Figure 6.1:** Illustration of the  $P$  and  $D$  data structures.  $P$  contains, for every  $(y,z)$  tuple, an index and a run-length into  $D$ .  $D$  contains at each position the information to render a single shell voxel. Voxels that do not contribute to the rendering are not stored at all.

voxels is equal to the set of all voxels in the volume that is being analysed. In this way, shell rendering can be seen as a hybrid volume visualisation method that can be utilised anywhere on the spectrum between surface rendering and direct volume rendering.

Shell voxels are stored in two data-structures,  $P$  and  $D$ , illustrated in figure 6.1.  $D$  is a list of all shell voxels and associated attributes in a volume row-by-row and slice-by-slice order. For each row in the volume,  $P$  contains an index into  $D$  of the first shell voxel in that row as well as the number of voxels in that row.  $P$  itself is indexed with a slice and row tuple. This enables a very efficient way of storing and accessing the volume, but more importantly, the voxels can be very rapidly traversed in a strictly front-to-back or back-to-front order from any view direction in an orthogonal projection setting. The shells have to be re-extracted for any change in opacity transfer function or the opacity thresholds.

When rendering, we can determine the order of indexing by using the view octant: Shell rendering supports front-to-back as well as back-to-front volume traversal when projecting voxels. When projecting voxels in the back-to-front setting, the projection is composited with all pixel positions  $p$  of the image buffer  $I$  that are affected by the voxel's projection as follows:

$$I_{\lambda}(p) = S_{\lambda}(v)S_{\alpha}(v) + (1 - S_{\alpha}(v))I_{\lambda}(p)$$

where  $v$  is the voxel that is being projected,  $S_{\alpha}(v)$  is its opacity,  $S_{\lambda}(v)$  represents its colour in the  $\lambda$  band as determined by the transfer function and optional shading. Unlike equation 6.1, this is identical to standard alpha-blending.

The neighbourhood code mentioned above is used during rendering to determine if a voxel is occluded from the current viewpoint by the neighbouring voxels in which case such a voxel is skipped. Due to the binary packing of the neighbourhood code, this checking can be done in constant time.

Shell rendering also supports the 3D editing of structures, the computation of the volume of structures surrounded by shells and the measurement (linear and curvilinear) of rendered surfaces. These facilities make this technique very attractive especially for clinical use.

## 6.3 The ShellSplatting Algorithm

The algorithm consists of a pre-processing step and a rendering step. During pre-processing, which happens once per volume or transfer function change, the shell structure is extracted from the volume or from an existing shell structure. This shell structure is exactly the same as for traditional shell rendering.

Instead of performing only voxel projection during rendering however, we render each shell voxel by projecting and compositing a pre-integrated Gaussian splat modulated by the shaded optical properties of that voxel. This is done by placing a rectangular polygon, coloured with the looked-up voxel colour and texture-mapped with the pre-integrated reconstruction function, at each and every voxel position. The texture is uploaded to the rendering pipeline once and can be re-used as many times as necessary.

In the following two subsections, the calculation of the rectangular polygon and the iteration and rendering of voxels with textured polygons will be explained in more detail.

### 6.3.1 Calculation of splat polygon

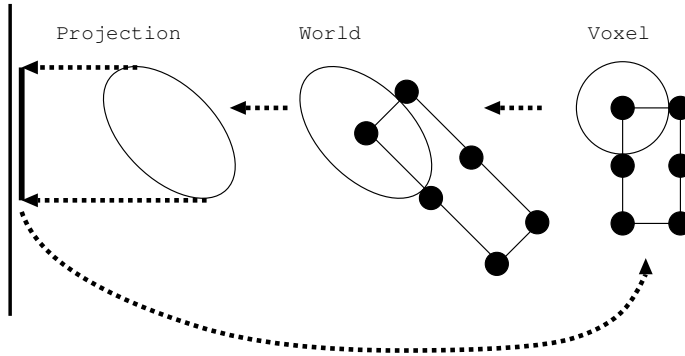
The polygons have to be perpendicular to the view axis since the reconstruction function has been pre-integrated along the view direction. In addition, they should be sized so that the mapped texture is scaled to the correct splat dimensions. Remember that for anisotropically sampled volumes, the splats potentially differ in size and shape for each different view direction.

To visualise this, imagine a three-dimensional ellipsoid bounding the reconstruction kernel at a voxel. If we were to project this ellipsoid onto the projection plane and then “flatten” it, i.e. calculate its orthogonally projected outline (an ellipse) on the projection plane, the projected outline would also bound the pre-integrated and projected splat. A rectangle with principal axes identical to those of the projected ellipse, transformed back to the drawing space, is used as the splat polygon.

Figure 6.2 illustrates a two-dimensional version of this procedure. In the figure, however, we also show the transformation from voxel space to world space. This extra transformation is performed so that rendering can be done in voxel space, where reconstruction functions can be spherically symmetric, even if the volume has been anisotropically sampled. Alternatively stated, the anisotropic volume is warped to be isotropic. The voxel-to-model, model-to-world, world-to-view and projection matrices are concatenated in order to form a single transformation matrix  $\mathbf{M}$  with which we can move between the projection and voxel spaces.

In order to perform the steps outlined above, we require some math. A quadric surface, of





**Figure 6.2:** Illustration of the calculation of the reconstruction function bounding function in voxel space, transformation to world space and projection space and the subsequent “flattening” and transformation back to voxel space.

which an ellipsoid is an example, is any surface described by the following implicit equation:

$$\begin{aligned} f(x,y,z) &= ax^2 + by^2 + cz^2 + 2dxy + 2eyz + \\ &\quad 2fzx + 2gx + 2hy + 2jz + k \\ &= 0 \end{aligned}$$

This can also be represented in its matrix form as:

$$\mathbf{P}^T \mathbf{Q} \mathbf{P} = 0$$

where  $\mathbf{Q} = \begin{bmatrix} a & d & f & g \\ d & b & e & h \\ f & e & c & j \\ g & h & j & k \end{bmatrix}$  and  $\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$

Such a surface can be transformed with a 4x4 homogeneous transformation matrix  $\mathbf{M}$  as follows:

$$\mathbf{Q}' = (\mathbf{M}^{-1})^T \mathbf{Q} \mathbf{M}^{-1} \quad (6.3)$$

A reconstruction kernel bounding sphere in quadric form  $\mathbf{Q}$  is constructed in voxel space. Remember that this is identical to constructing a potentially non-spherical bounding *ellipsoid* in world space. In this way anisotropically sampled volumes are elegantly accommodated.

This sphere is transformed to projection space by making use of equation 6.3. The two-dimensional image of a three-dimensional quadric of the form

$$\mathbf{Q}' = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix}$$

as seen from a normalised projective camera is a conic  $\mathbf{C}$  described by  $\mathbf{C} = c\mathbf{A} - \mathbf{b}\mathbf{b}^T$  [76, 77]. In projection space,  $\mathbf{C}$  represents the two-dimensional projection of  $\mathbf{Q}$  on the projection plane.

An eigendecomposition  $\mathbf{C}\mathbf{X} = \mathbf{X}\lambda$  can be written as

$$\mathbf{C} = (\mathbf{X}^{-1})^T \lambda \mathbf{X}^{-1}$$

which is identical to equation 6.3. The diagonal matrix  $\lambda$  is a representation of the conic  $\mathbf{C}$  in the subspace spanned by the first two eigenvectors (transformation matrix) in

$$\mathbf{X} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix}$$

where  $R$  and  $t$  represent the rotation and translation sub-matrices respectively. The conic's principal axes are collinear with these first two eigenvectors.

In other words, we have the orientation and length of the projected ellipse's principal axes which correspond to the principal axes of a reconstruction function bounding sphere that has been projected from voxel space onto the projection plane. Finally, these axes are transformed back into voxel space with  $\mathbf{M}^{-1}$  and used to construct the rectangles onto which the pre-integrated reconstruction function will be texture-mapped.

### 6.3.2 Back-to-front shell voxel traversal

The rendering pipeline is configured so that all geometry can be specified in voxel space. A back-to-front traversal of the shell voxels is initiated. We have chosen back-to-front traversal, as this method of composition (also known as the painter's algorithm) is easily accelerated with common graphics hardware.

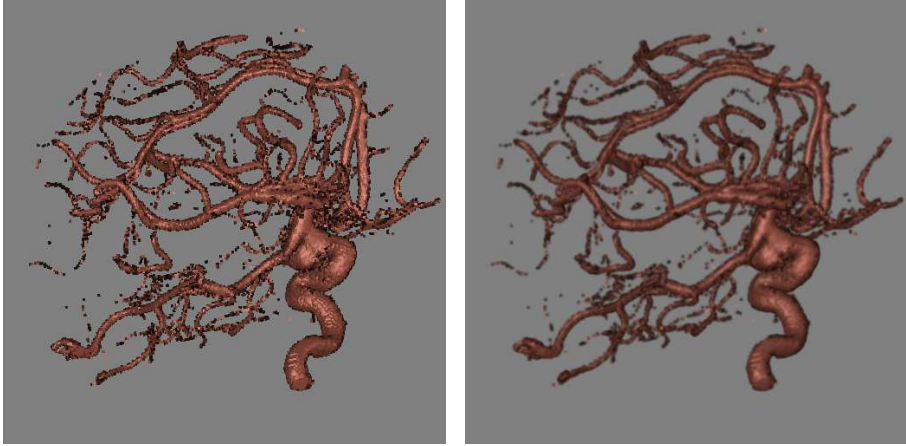
For each voxel, the corresponding optical characteristics and opacity are uploaded to the pipeline. The rectangle created in section 6.3.1 is translated so that its centre is at the voxel position and the geometry is then uploaded and associated with the pre-integrated kernel texture for texture-mapping.

The hardware is configured to shade each voxel once and then modulate both the resultant colour and opacity with the pre-integrated kernel texture. Texture-mapped polygons of successive voxels are composited on the image buffer according to equations 6.1 and 6.2. This kind of shading, modulation and blending are straight-forward and standard operations in currently available commodity graphics hardware.

After having iterated through all shell voxels, a complete image frame has been built up on the image plane with very little use of the computer's general purpose processor.

## 6.4 Results

The ShellSplatter was implemented as two VTK [73] objects. All rendering is performed via OpenGL. Parameters such as the bounding volume for the Gaussian reconstruction kernel (i.e. at which radius it's truncated) and the Gaussian standard deviation can be interactively modified during rendering.



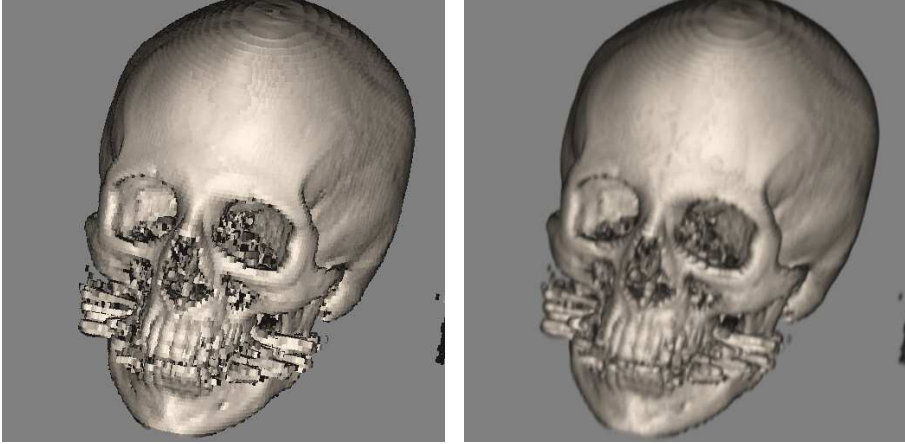
**Figure 6.3:** ShellSplat rendering of rotational b-plane x-ray scan of the arteries of the right half of a human head, showing an aneurism. On the left is the fast rendering and on the right is the high quality version. Data from volvis.org courtesy of Philips Research, Hamburg, Germany. See colour figure C.8.

Additionally, our implementation offers two major render modes: fast and high quality. In the fast mode, rectangular polygons are rendered for all voxels, but they are not texture-mapped with splats. This is equivalent to a fast hardware-assisted form of standard shell rendering. In high-quality mode, all polygons are texture-mapped. The fast mode is very usable for rapid investigation of the volume and using this in an automatic level-of-detail rendering system would be straight-forward.

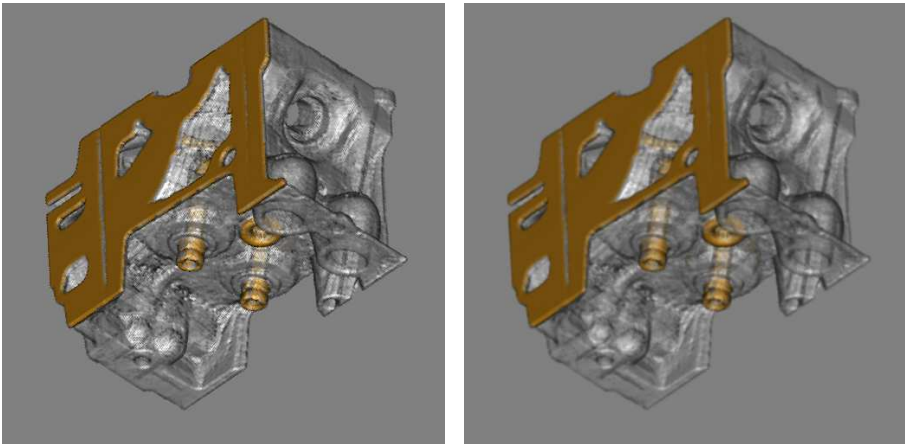
Figures 6.3, 6.4 and 6.5 show examples of ShellSplatter renderings in both fast and high-quality mode. The high-quality renderings were made with a Gaussian splat with radius 2 voxels and standard deviation  $\sigma = 0.7$ . The fast renderings were made with rectangular polygons of 1.6 voxels across.

Table 6.1 shows rendering speed in frames per second for three example data sets. These tests were done on a Linux machine with an AMD Athlon 1.2GHz and a first generation NVidia GeForce3. The image size was  $512 \times 512$ . The speed difference between the fast and high quality modes is much more pronounced on lower end graphics hardware.

In the table the number of shell voxels per dataset has been specified. This already represents a tremendous saving over splatting the whole volume. Also take into account that, during rendering, not all voxels are rendered due to the octant-dependent occlusion checking. In the case of the engine block for instance, an average of 140000 out of 230000 shell voxels (out of an original total of 8388608 voxels) are actually sent to the rendering pipeline. Pre-processing (i.e. extraction of the shell data-structures) takes approximately 10 seconds for a complete  $256^3$  volume.



**Figure 6.4:** ShellSplat rendering of the Stanford CTHed data set. The fast rendering is on the left and the high quality is on the right. Note that this data set is anisotropically sampled. See colour figure C.9.



**Figure 6.5:** ShellSplat rendering of the well-known engine block data set. The grey material has been made transparent. Data set supplied by volvis.org, originally made by General Electric. Fast rendering on left, high quality on the right. See colour figure C.10.

	Frame rate (frames per second)		
Rendering Mode	Aneurism	CT Head	Engine Block
	$256^3$	$256^2 \times 99$	$256^2 \times 128$
	0.2% shell 35000 voxels	2.5% shell 160000 voxels	2.7% shell 230000 voxels
Fast	44	12	8
High Quality	41	6	8

**Table 6.1:** ShellSplatter rendering frame rates for three example data sets. Each data set’s resolution is shown along with the percentage and number of voxels that are actually stored in the shell rendering data structures.

## 6.5 Conclusions and Future Work

In this chapter we have presented a volume rendering algorithm called ShellSplating. Combining the data-structures of shell rendering with the rendering techniques of splatting, this method offers interactive and high-quality volume rendering. The data-structures enable rapid back-to-front or front-to-back volume traversal that efficiently ignores voxels that are transparent or occluded, whilst the splatting enables higher-quality volume renderings than with traditional shell rendering. In addition, the shell rendering data structures support 3D volume editing and measuring of volumes and surfaces.

We have also demonstrated a straight-forward method of creating a suitable splat polygon for the hardware-assisted rendering of anisotropically sampled volumes.

An important point with regards to hardware acceleration is the fact that our algorithm requires only basic functionality from the graphics accelerator. This makes it useful on more generic devices whilst still enabling it to profit from newer generations of hardware.

ShellSplating is obviously more efficient with datasets consisting of many “hard” surfaces, i.e. the volume contains large contiguous volumes of voxels with opacity greater than  $\Omega_h$ . If this is not the case, the algorithm gradually becomes a hardware-accelerated traditional splatting method. In other words, ShellSplating can be very simply configured to function anywhere on the continuous spectrum between voxel-based iso-surface rendering and direct volume rendering.

Due to the way in which the splat quads are blended, combining ShellSplatted volume renderings with opaque polygonal geometry in a single rendering is trivial. All these factors make this method very suitable for virtual orthopaedic surgery where bony structures are sculpted by polygonal representations of surgical tools, which is one of our planned applications.

We would like to extend the ShellSplatter to support perspective rendering. In order to do this, the shell rendering octant-dependent back-to-front ordering has to be modified and the splat creation and rendering has to be adapted. Excellent work has been done on the former problem [78, 16] and these resources will be utilised. Preliminary experiments on adapting the ShellSplatter are promising.

## **Acknowledgements**

This research is part of the DIPEX (Development of Improved endo-Prostheses for the upper EXtremities) program of the Delft Interfaculty Research Center on Medical Engineering (DIOC-9).

We would like to thank Jorik Blaas for valuable discussion and his expert advice on low-level graphics programming issues. We would also like to thank Roger Crawfis and Klaus Mueller for enlightening e-mail conversations on the topic of splatting.

---

## Improved Perspective Visibility Ordering for Object-Order Volume Rendering

---

This chapter is an extended version of [12].

### **Abstract**

Finding a correct *a priori* back-to-front (BTF) visibility ordering for the perspective projection of the voxels of a rectangular volume poses interesting problems. The BTF ordering presented by Frieder et al. [23] and the permuted BTF presented by Westover [96] are correct for parallel projection, but not for perspective projection [78]. Swan presented a constructive proof for the correctness of the perspective BTF (PBTF) ordering [78]. This was a significant improvement on the existing orderings. However, his proof assumes that voxel projections are not larger than a pixel, i.e. voxel projections do not overlap in screen space. Very often the voxel projections *do* overlap, e.g. with splatting algorithms. In these cases, the PBTF ordering results in highly visible and characteristic rendering artefacts.

In this chapter we analyse the PBTF and show why it yields these rendering artefacts. We then present an improved visibility ordering that remedies the artefacts. Our new ordering is as good as PBTF, but it is also valid for cases where voxel projections are larger than a single pixel i.e. when voxel projections overlap in screen space. We demonstrate why and how our ordering works at fundamental and implementation levels. A VTK-based implementation of hardware-accelerated splatting incorporating our new ordering is available.

## 7.1 Introduction

In volume visualisation, there are three main options: two-dimensional slices of the volume can be rendered, extracted surfaces can be rendered or direct volume rendering (DVR) can be utilised [14].

Direct volume rendering [45, 21] (DVR) allows the visualisation of structures in the data without having to make decisions about the precise location of object boundaries by extracting polygonal surfaces. Instead, we can define multi-dimensional transfer functions that assign optical properties to each differential volume element and directly visualise structures on the grounds of this transformation. Most DVR methods can be classified as image-order or object-order algorithms.

Image-order algorithms [45] entail that all pixel locations of the expected result image are traversed. At each pixel location, a view ray is cast through the volume. The volume is sampled at regular positions along this ray. At each position, the scalar volume value is interpolated, very commonly with a trilinear interpolator. These values are transformed by the accompanying transfer function to their corresponding optical characteristics. The order of this operation can be changed: the optical characteristics can be determined before the interpolation stage. All the transformed characteristics along a ray are composited in order to determine the optical characteristics of a single pixel.

In *traditional* object-order volume rendering, we traverse voxel locations instead. At each voxel location, we determine the full contribution of that voxel to the final image and composite it. An important advantage of object-order rendering is that we retrieve any voxel value only once, whereas with ray casting, it is most often the case that we sample a single voxel value multiple times during a single rendering. Ray casting does make early ray termination and frustum clipping optimisations possible. In the same vein, object-order rendering can make use of for instance run length encoding and empty space skipping. However, in order to ensure correct image composition during object-order rendering, we have to visit and project the voxels in a strictly front-to-back (FTB) or back-to-front (BTF) order.

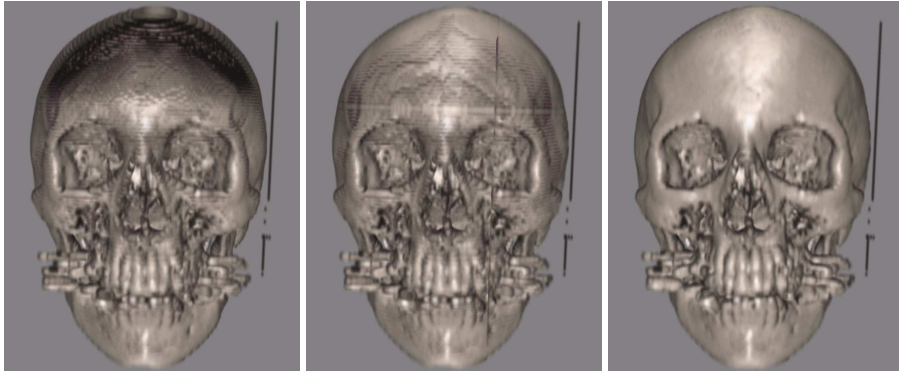
This constraint does lead to another very important advantage of object-order volume rendering, and that is the fact that the hybrid rendering of opaque surfaces and semi-opaque volumes is straight-forward. Voxels are treated as translucent geometry and rendered after the traditional opaque geometry in BTF order, resulting in seamless blending between polygonal and direct volume rendering.

Satisfying the ordering constraint means one of two things:

1. the voxels can be sorted according to their distance from the viewpoint before rendering;
2. the inherent regularity of the volume grid can be used to derive a valid *a priori* visibility order, so that the voxels are retrieved in a correct BTF or FTB ordering.

It is obvious that the latter method, if possible, has the potential to be more efficient, as no sorting operation is required for each rendered view.





**Figure 7.1:** A perspective splatting of the Stanford CT Head dataset from the same viewpoint with three different orderings: traditional BTF, PBTF and our IP-PBTF.

For parallel projection, this traversal exists and is easy to find. For perspective projection, the situation is not as positive. The conventional traversals for parallel projection are *incorrect* for perspective projection, as is demonstrated by the left-most image in Figure 7.1. Alternative solutions have been proposed, of which the perspective back-to-front ordering, or PBTF, is “most correct” [78]. PBTF has been proven to be correct for cases where each voxel contributes to at most a single pixel in the resultant image. However, this is most often not the case.

When each voxel affects more than a single pixel, the PBTF ordering introduces very visible rendering artefacts. The centre image in Figure 7.1 and the image in Figure 7.3 show examples of one such artefact. The “cross” artefact can easily be seen on the engine block. In the case of the skull, the cross centre is right above its left eye. The skull also shows shading and stair-stepping artefacts that are due to incorrect visibility ordering. The reasons for both of these artefacts are explained in section 7.4.

This is a significant problem, as there is no alternative ordering for perspective object-order volume rendering of discrete voxels.

The research question we attempt to answer in this chapter is: *How can we traverse a regularly spaced voxel grid during object-order perspective mode rendering, where voxel projections potentially overlap in screen space, so that a back-to-front ordering with regards to the view point can be derived, without explicitly sorting the voxels?*

This problem is analysed and a solution is proposed. We explain why the PBTF ordering is only partially correct. We then propose a new ordering, called IP-PBTF, or Interleaved and Permuted Perspective Back-to-Front ordering, that is based on PBTF and rectifies the demonstrated problems. We explain why it works at basic and implementation levels. We also present a method to implement the IP-PBTF ordering that is compatible with empty space skipping volume rendering implementations. The presented implementation is just as fast as the regular PBTF.

A VTK-based hardware-accelerated implementation of a splatting-variant incorporating the new ordering is available<sup>1</sup>.

The rest of this chapter is organised as follows: In Section 7.2 we discuss the existing orderings in more detail. We also discuss splatting as one of the better known examples of object-order volume rendering algorithms. The PBTF is explained in Section 7.3, where we also show when, why and how the PBTF fails.

Although these two sections are for a large part previous work and they constitute a significant part of this chapter, the aspects that are documented are very important in understanding the development of our solution. In addition, the PBTF volume partitioning is not widely known and it was deemed necessary to explain it as part of our work's context.

We present our new ordering in Section 7.4. In this section we also give implementation-oriented tips and we also show how to implement the ordering for empty space skipping implementations. Performance timings are given and briefly discussed in Section 7.5. In Section 7.6 we discuss our findings.

## 7.2 Previous Work

It has been shown that, for parallel projection, one can select a traversal direction for each of the axes of the rectilinear grid on which the volume was defined so that the retrieved list of voxels would be ordered strictly back-to-front with respect to the viewing plane [23].

For example, the two-dimensional volume shown in Figure 7.2 can be traversed in a strictly BTF order relative to view plane A by iterating  $y$  from 0 to  $y_1$  in the outer loop and  $x$  from 0 to  $x_1$  in the inner loop. For a BTF order relative to view plane B, the traversal direction of  $x$  would have to be reversed, i.e. it would iterate from  $x_1$  to 0.

The BTF order is still maintained if  $y$  and  $x$  change places, i.e.  $x$  is moved to the outer loop and  $y$  to the inner loop. As long as the correct traversal direction for each of the grid axes has been chosen, the permutation of the axes, i.e. determining how the axis indices are nested during grid traversal, does not affect the correctness of the BTF ordering for orthogonal projection.

This ordering is not correct for perspective projection however. An extreme example of this can be seen in the left-most image in Figure 7.1. Westover's ordering, henceforth called the WBTF as per Swan's exposition [78], added a permutation constraint [96]. Algorithm 1 shows the WBTF. The permutation constraint determines how the axes should be nested during the traversal.

The WBTF is correct for more cases than the conventional BTF during perspective projection, but it is relatively easy to construct simple examples where it yields incorrect orderings. Practically, it yields similar but less severe rendering artefacts than those of the BTF.

Swan unified and proved [78] the perspective-correct orderings previously published by Anderson for 2D [2] and Max for 3D [52]. This order will henceforth be called the PBTF. The PBTF was a significant improvement on the existing orderings and was proved to be

---

<sup>1</sup>As soon as this work is published, the implementation and its source will be released under the VTK license.

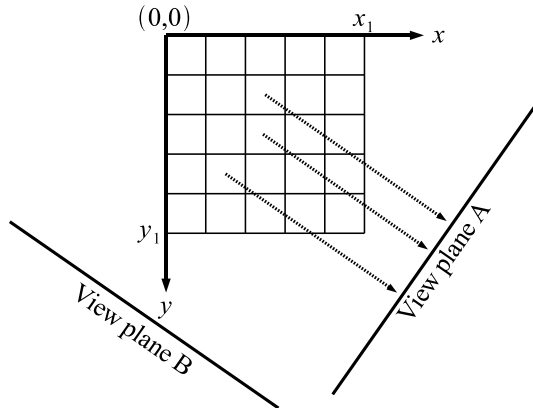


Figure 7.2: Diagram illustrating BTf and WBTF ordering for parallel projection.

---

**Algorithm 1** The WBTF ordering.

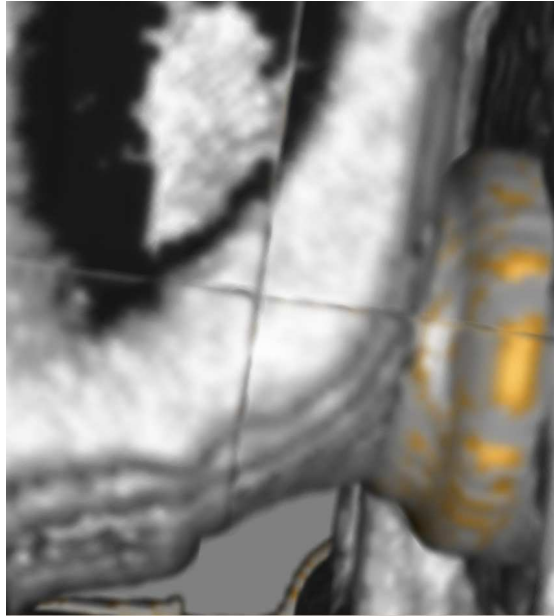
---

```

 $k \leftarrow$  axis most perpendicular to view plane
 $j \leftarrow$  axis next most perpendicular to view plane
 $i \leftarrow$  axis most parallel to view plane
choose  $k_{\min}, k_{\max}, j_{\min}, j_{\max}, i_{\min}, i_{\max}$  according to view plane
for  $k = k_{\min}$  to  $k_{\max}$  do
  for  $j = j_{\min}$  to  $j_{\max}$  do
    for  $i = i_{\min}$  to  $i_{\max}$  do
      visitVoxel( $i, j, k$ )
    end for
  end for
end for

```

---



**Figure 7.3:** Close-up of splatting of “engine block” dataset showing the cross artefact caused by PBTF ordering. The cross artefact occurs at the volume subdivision boundaries due to the fact that splats overlap in screen space.

correct for cases where voxel projections are no larger than a pixel. However, when this is not the case, the use of the PBTF ordering results in very noticeable rendering artefacts. The centre image in Figure 7.1 and the image in Figure 7.3 show the very typical PBTF “cross” artefact.

This is quite a serious problem, as PBTF is currently the only known correct ordering for perspective object-order volume rendering of discrete voxels. In section 7.3 we explain the PBTF ordering in more detail and also show why and how it breaks.

In [18], the problem was worked around by adapting the data structures so that the viewing angle was always large relative to the splat size. This minimised the visual artefacts. However, ensuring a large viewing angle in all applications is often not practical. This work also recognised the importance of the traversal axis permutation.

Splatting is a popular object-order (i.e. forward projection) volume rendering method that treats an N-dimensional sampled volume as a grid of overlapping N-dimensional volume reconstruction function kernels, weighted with voxel densities. These weighted kernels, often modelled as Gaussians, are projected onto the image plane to form “splats” that are composited with affected pixels [97]. In this way, splatting approaches the problems of volume reconstruction and rendering as a single task.

In original splatting, the volume is traversed from front to back or from back to front.

Centred at each voxel position a reconstruction kernel is integrated along the view axis to form a pre-integrated two-dimensional kernel footprint. The kernel footprint is used to modulate the looked-up and shaded voxel optical characteristics (colour and opacity) of that voxel and projected onto the image plane where it is composited with the affected pixels. During splatting, a single voxel typically affects more than one pixel, and thus the PBTF ordering is not correct for perspective projection in this case.

The use of pre-integrated reconstruction kernels causes inaccuracies in the composition as each kernel is independently integrated and not in a piecewise fashion along with other kernels in the path of a view ray. This can result in colour-bleeding of obscured objects in the image. Westover proposed first performing compositing of piece-wise kernel integrations into volume axis-aligned sheet-buffers and then onto the image buffer to alleviate this effect [96]. However, the axis-aligned sheet-buffering resulted in sudden changes in image brightness, also known as “popping”, during rotation. Mueller introduced image-aligned sheet-buffers to eliminate this problem [54, 55].

Image-aligned sheet-buffer splatting does not show any of the mentioned artefacts due to the fact that sections of voxel-centred reconstruction functions are accumulated into sheet buffers first. This technique yields the highest quality renderings. However, it is slower and more complex than traditional per-voxel splatting. Also very importantly, per-voxel splatting translates easily to simple and ubiquitous graphics hardware and allows easy hardware assisted blending with traditional opaque geometry. This is invaluable in for example surgical simulation.

In short, traditional per-voxel splatting fills an important niche in the direct volume rendering world. This accentuates the necessity of a correct perspective ordering for the object-order rendering of discrete voxels.

## 7.3 PBTF

In this section, we explain the PBTF ordering as proposed by Anderson for 2D [2], Max for 3D [52] and proven by Swan [78]. We also point out, for the first time, the straight-forward relationship between the Meshed Polyhedra Visibility Ordering (MPVO) algorithm [98] and the PBTF.

Let  $\mathbf{v} = (v_x, v_y, v_z)$  be the position of the camera or view point. Depending on the location of  $\mathbf{v}$  relative to the volume, the volume is partitioned into smaller sub-volumes. Very simply speaking, the volume is divided into sub-volumes by three dividing planes, each orthogonal to a different volume grid axis and passing through  $\mathbf{v}$ .

For example, if  $\mathbf{v}$  were completely *inside* the volume, the three dividing planes would divide the volume into 8 sub-volumes. This is called the “volume-on” case. If  $\mathbf{v}$  were to move in the  $x$  direction till it was *outside* the volume, the three dividing planes would divide the volume into 4 sub-volumes, as the plane orthogonal to the  $x$ -axis passing through  $\mathbf{v}$  would not intersect the volume any more. This is called the “face-on” case. If  $\mathbf{v}$  were now to move in the  $y$  direction till it was above the highest extent of the volume, the three dividing planes would divide the volume into 2 sub-volumes, as only the plane orthogonal to the  $z$ -axis would

still intersect the volume. This is called the “edge-on” case. Moving  $\mathbf{v}$  in the  $z$  direction till the  $z$ -orthogonal plane does not intersect the volume would not result in any partitioning. This is called the “corner-on” case. Algorithm 2 shows a pseudo-code description of the PBTF partitioning. For each iteration of  $k$ , a single axis is split.

---

**Algorithm 2** Determining volume partitioning for the PBTF.

---

```

Volume dimensions are  $(x_0, x_1, y_0, y_1, z_0, z_1)$ 
 $\mathbf{v} = (v_x, v_y, v_z)$  is perspective view point
for  $k = x, y, z$  do
  if  $v_k < k_0$  or  $v_k > k_1$  then  $\{v_k$  is “outside” $\}$ 
     $(k_{\min 1}, k_{\max 1}) \leftarrow (k_0, k_1)$ 
     $(k_{\min 2}, k_{\max 2}) \leftarrow (0, 0)$ 
  else  $\{v_k$  is “inside” $\}$ 
     $k_v \leftarrow v_k$  rounded to nearest voxel
     $(k_{\min 1}, k_{\max 1}) \leftarrow (k_0, k_v - 1)$ 
     $(k_{\min 2}, k_{\max 2}) \leftarrow (k_v, k_1)$ 
  end if
end for
output:  $(k_{\min 1}, k_{\max 1}), (k_{\min 2}, k_{\max 2})$  partitioning for each axis

```

---

After having determined the volume-partitioning, a traditional BTF traversal is executed for each sub-volume, i.e. for each partition an independent set of traversal directions is chosen according to the BTF. Swan’s test implementation does indeed include a WBTF per volume subdivision, but his proofs explicitly allow any permutation.

For the given view point, no voxel in any sub-volume can occlude any voxel in any other sub-volume. This is easy to see, as the planes dividing the volume into sub-volumes all pass through the view point. None of the view-rays radially emanating from the view point can ever intersect any of the dividing planes, i.e. no view-ray can pass through more than a single sub-volume. Hence, there can be no occlusion between sub-volumes. In other words, we can treat the partitioned sub-volumes as independent volumes and render each one separately with a traditional BTF traversal. It is important to note that each partitioned sub-volume constitutes a corner-on case.

It is interesting to note that the Meshed Polyhedra Visibility Ordering (MPVO) algorithm proposed by Williams [98] reduces to the PBTF ordering and volume subdivision for a rectangular volume. Swan mentions this work, but doesn’t show the relationship to the PBTF. The MPVO orders the polyhedra in convex meshes by constructing a directed adjacency graph for all polyhedra. A direction is assigned to each edge by constructing a plane through the face separating the two adjacent polyhedra. This plane divides the world into two half-spaces. The direction of the edge is always towards the half-space containing the view point. If a topological sort of this directed adjacency graph is performed, the resultant ordering is a valid BTF order.

The MPVO can be applied to regularly spaced voxel grids if each voxel is seen as a

cubic cell. If we were to construct, for each cubic cell, six dividing planes (according to the principle above), there is a maximum of three main plane orientations. Continuing this reasoning, it is easy to see that in the worst case scenario (view point within the volume), the volume will be divided into eight regions, where the voxel cuboids in each of the eight regions will have identical graph edge directions. A topological sorting of this graph will result in eight volume subdivisions, each with its own BTF ordering. One can continue this mental exercise for all possible cases (volume-on, face-on, edge-on and corner-on), and the MPVO will reduce to the PBTF ordering every time.

Note that the MPVO, just like the PBTF, only determines different sets of traversal directions for all sub-volumes, but places no constraint on the axis permutation of each sub-volume. Any of the three permutations (for any of the sub-volumes) constitutes a correct topological sort of the directed adjacency graph.

## 7.4 IP-PBTF

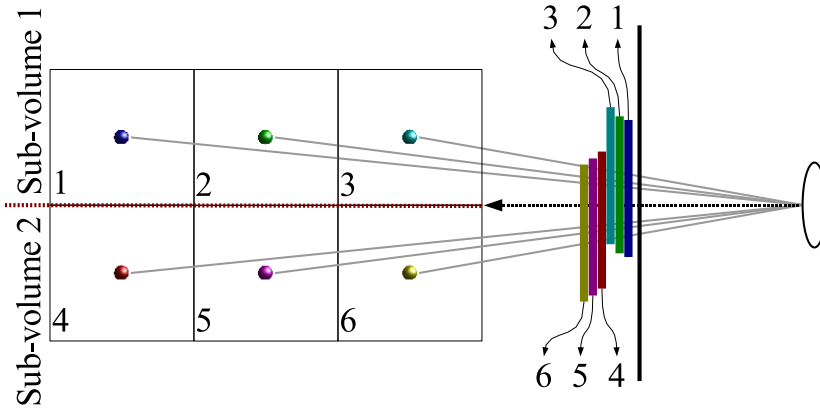
By showing a few simple examples, we demonstrate the two major reasons why PBTF generates incorrect orderings. Identifying the deficiencies leads directly to the remedies. Subsequently, we analyse the problem in more detail and show that our ordering is an improvement in all possible cases. We then present some implementation details.

### 7.4.1 Constructing the IP-PBTF

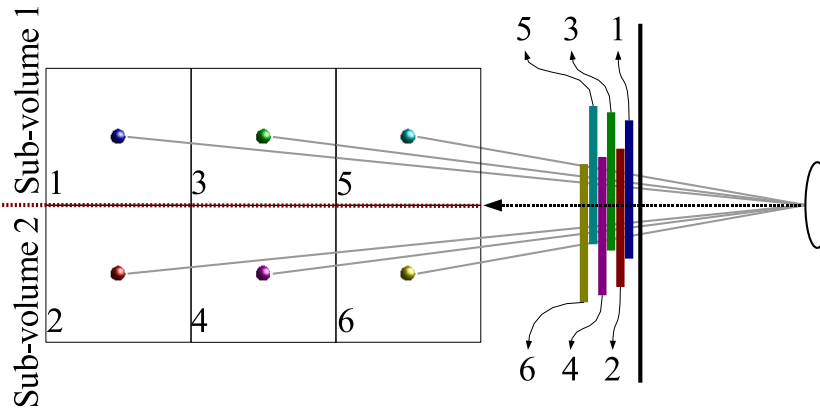
When voxel projections are no larger than a single pixel, there is generally no occlusion between sub-volumes. However, when voxel projections are larger than a single pixel, this assumption is incorrect, i.e. significant occlusion takes place between sub-volumes. This is the primary reason why the PBTF ordering is incorrect for the case where voxel projections are larger than a pixel. Figure 7.4 shows a very simple 2D example where the PBTF is not sufficient. Notice that, due to the larger-than-pixel voxel projections, voxel 2 is composited *before* voxel 4, and voxel 3 is composited *before* voxels 4 and 5. In a larger volume, this incorrect compositing will take place along any sub-volume division and cause the visible artefacts that we have shown. The cross artefact in Figure 7.3 is a specific example of this.

Studying the problem at this scale already presents the first part of our solution. By interleaving the voxels along the volume division, i.e. rendering voxels alternately from the top and bottom sub-volume, whilst still maintaining the intra-sub-volume BTF ordering, we would solve at least the problem shown in Figure 7.4. Figure 7.5 shows the results of this change.

In this case, the interleaving is quite simple, as the sub-volumes are equally large on the interleaved dimension. If this is not the case, the largest sub-volume has to be traversed first until the untraversed remainder is as small as the smaller sub-volume. At that point, the interleaving should begin. Also, sub-volumes are often partitioned differently on all three axes. It is not sufficient to interleave only along the sub-volume division: the complete sub-volumes have to be interleaved for as far as their extents allow.

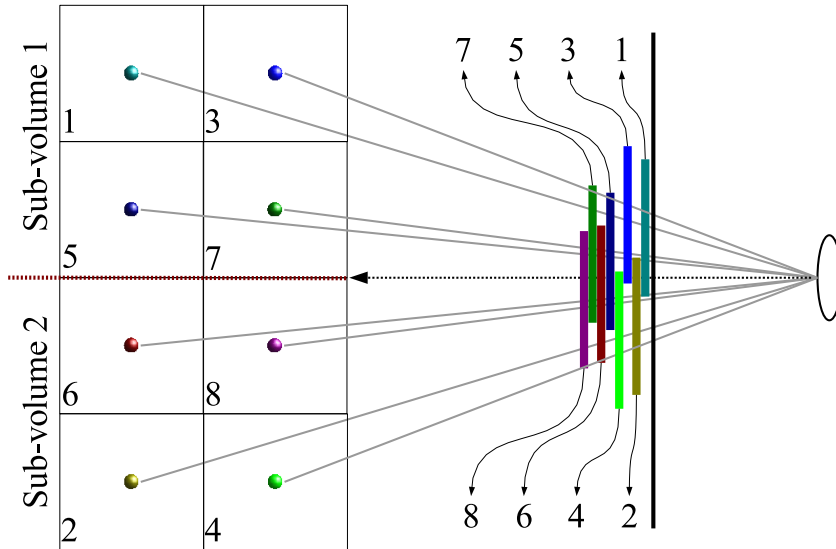


**Figure 7.4:** A simple 2D example showing how the PBTf visibility ordering is incorrect for cases where the voxel projection is larger than a single pixel. Each numbered block represents a voxel.



**Figure 7.5:** Interleaving the ordering of the voxels in the sub-volumes solves the problem shown in Figure 7.4.





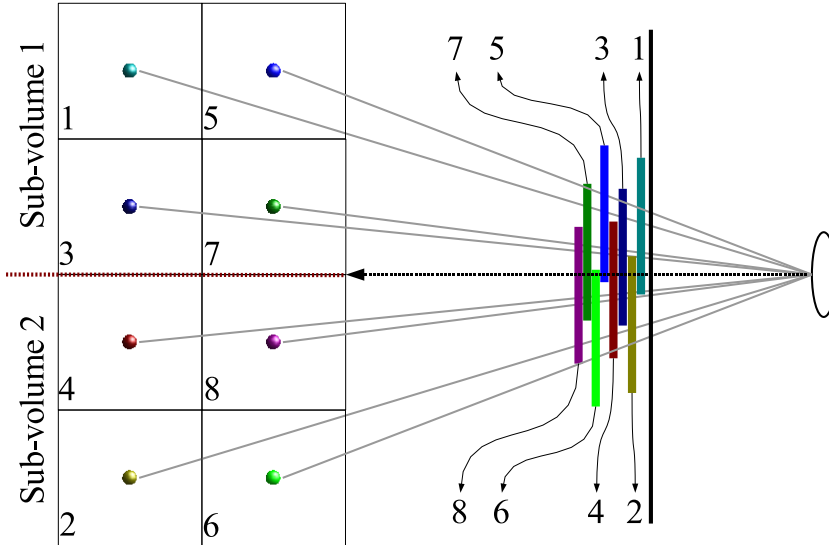
**Figure 7.6:** Sub-volume interleaving by itself is not sufficient to remedy the PBTF visibility problems. In this simple case, in spite of interleaving, voxels 3 and 4 have been incorrectly composited before voxels 5 and 6.

In this simple case, the interleaving seems to solve the visibility ordering. However, we can construct another simple case that shows how the interleaving by itself is not quite sufficient. In Figure 7.6 interleaving has been applied: the voxels are rendered alternately from sub-volumes 1 and 2. However, the slowest changing grid axis index belongs to the axis orthogonal to the sub-volume division. In this very simple case, voxels 3 and 4 are incorrectly rendered *before* voxels 5 and 6. Once again, in a more extensive volume, this will cause visible artefacts.

From this example, it's clear that one should carefully choose an applicable permutation: the slowest changing axis index should never be that belonging to the axis orthogonal to any of the sub-volume division planes. The result of applying this rule to the simple example in Figure 7.6 is shown in Figure 7.7. Note that all voxels are now projected in the correct ordering. We re-iterate the fact that Swan's proof allows any axis permutation for the PBTF sub-volumes, although his specific implementation did include a WBTF ordering for each sub-volume.

## 7.4.2 Analysis

Our new ordering for perspective object-order volume rendering is called the Interleaved and Permuted Perspective Back-to-Front ordering, or IP-PBTF. We have demonstrated with a few simple examples why the PBTF is not correct for cases where each voxel affects more than one pixel and how the IP-PBTF would be correct in these cases as well.



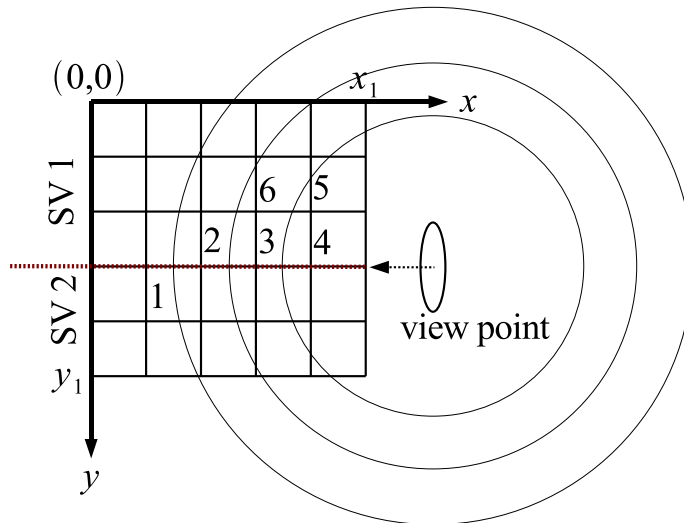
**Figure 7.7:** Interleaving between sub-volumes *and* selecting a suitable axis permutation remedies the PBTf problems in this example.

There are two reasons why the PBTf is incorrect for these cases:

1. There is a definite interaction between sub-volumes.
2. A non-permuted BTF (i.e. traditional) ordering is often incorrect even for a simple corner-on case, which is what every sub-volume reduces to in the PBTf ordering.

Studying Figure 7.8, which is valid for every possible 2D perspective projection configuration and can be constructed for the 3D case, it becomes obvious why the PBTf breaks. From this figure, it is clear that rendering for example voxels 2, 3 and 4 before 1 would be incorrect, as they are inside the outermost circle and 1 is outside, indicating that 1 is further away from the viewpoint and should be rendered first. This is exactly what the PBTf ordering would do, as it would completely render sub-volume 1 (indicated by SV 1) before sub-volume 2 (indicated by SV 2). Interleaving the two sub-volumes would order these correctly.

Interestingly, this figure also indicates the necessity for a *PBTf-based* ordering, i.e. one where the volume is partitioned into view point related sub-volumes. If we were to attempt a WBTF ordering for the whole volume,  $x$  would be chosen as the slowest changing axis, i.e. the outer-loop, and  $y$  as the fastest changing, i.e. the inner-loop. However, the direction of  $y$  would be either  $0 \rightarrow y_1$  or  $y_1 \rightarrow 0$ , neither of which can potentially yield a purely BTF order, as can be seen from the outermost circle: as we move outwards from the volume division line, the distance increases! This clearly indicates that the volume should be split and that



**Figure 7.8:** Circles (spheres in 3D) of equidistance from the view point. It is clear that the  $y$  should be partitioned and that each partition should be traversed in the opposite direction in order to maintain a BTF ordering. This illustration also helps to show why interleaving and permutation are required. See the text for more detail. “SV 1” and “SV 2” refer to “Sub-volume 1” and “Sub-volume 2” respectively.

the sub-volumes should be traversed in opposite directions of  $y$ , *towards* the sub-division line (plane in 3D).

The figure also shows why the traversal axes of each sub-volume should be permuted so that the slowest changing axis is not orthogonal to the volume sub-division line (plane in 3D). If we were to traverse sub-volume 1 with the  $y$ -axis as the slowest changing, we would render for example voxel 5 *before* voxel 3 and all voxels to its left, voxel 6 *before* voxel 2 and all voxels to its left, and so forth. This is of course not a correct ordering.

The permutation constraint can not be satisfied for the volume-on case, i.e. the case where the view-point is located *inside* the volume that is being rendered. In the volume-on case, there are three volume sub-division planes and 8 sub-volumes. There are 3 volume axes, i.e. one orthogonal to each sub-division plane, which means that it’s impossible to select a slowest traversal axis that is not orthogonal to a division plane. In this case, a good practical solution is to select the axis which is most perpendicular to the view plane as the slowest changing axis.

This also brings us to a limitation of almost all current object-order direct volume rendering methods: Distance from the view point increases radially outwards from the view point, i.e. the distance field is spherical, but grid-traversal orderings are constrained to the orthogonal volume grid. Thus, there are still bound to be cases when voxel pairs are incorrectly ordered, especially with very large voxel projections. In other words, if there were no constraints on processing speed, one would employ extremely thin and spherical sheet buffers

with the view point as centre. If there were no constraints on processing speed, but one had to render a voxel at a time, the distance measure around the view point could be sampled at the voxel centres and the voxels rendered in order of decreasing distance value.

That being said, we believe that the IP-PBTF is currently the best grid-constrained ordering for the object-order perspective direct volume rendering of discrete voxels. It is generically applicable to all cases where a volume on a rectilinear grid is volume rendered with discrete voxel projections overlapping in screen space and where the cost of explicit sorting is not affordable.

### 7.4.3 Implementing an interleaved split-dimension traversal

The IP-PBTF ordering is conceptually very simple. However, creating an efficient implementation is rather complex. Sub-volumes often have differing sizes on all dimensions as each dimension is independently split. The interleaving is not trivial, as the split most often yields two parts of differing size, which means that the interleaving only takes place part of the time. In addition, each sub-volume has its own set of traversal directions.

The crux of a good implementation is an efficient split-dimension traversal, i.e. code to handle the interleaving of the various axes. In order to facilitate implementation, we show an efficient way of setting up and executing a single split-dimension. Algorithm 3 shows how to set up a single split-dimension. This has to be done for all dimensions that have been partitioned by any of the dividing planes. Fortunately, this part happens only once per frame rendered. Algorithm 4 shows how to iterate through such a split-dimension during the actual voxel projection phase.

An axis is partitioned into two sections by the split-point  $k_v$ . The largest section is determined and its endpoints are stored in  $b_0$  and  $b_1$ .  $b_i$  determines the direction with which we traverse the largest section. We also determine a threshold,  $b_t$ . When the larger section index,  $b$ , reaches this threshold, the interleaving with the smaller dimension starts.  $s$  is used as an index for traversing the smaller section of the axis. Its direction is always opposite to that of the  $b$  index.

If the currently traversed interleaved axis has nested axes, *nestedLoop()* is called twice. Each of the *nestedLoop()* calls represents another potentially split dimension, but could also represent a straight-forward non-interleaved axis traversal, depending on the relevant PBTF partitioning case. If the axis that is being traversed is already at the most nested level, *visitVoxel()* is called, meaning that that particular voxel is rendered. For three nested split-dimensions for example, this means that *visitVoxel()* will be called a maximum of 8 times during each iteration of the outer loop.

### 7.4.4 Efficient interleaving with space skipping

If the volume rendering algorithm allows random access to the voxels without a performance penalty, the interleaved split-dimension traversal described above is sufficient and should not affect performance significantly.

---

**Algorithm 3** Setting up a split-dimension. See section 7.4.3 for details.

---

```

split-dimension is  $k_0 \leq k \leq k_1$ 
split-point is  $k_v$ 
if  $k_v - k_0 > k_1 - k_v$  then
   $b_0 = k_0$ 
   $b_1 = k_v + 1$ 
   $b_i = 1$ 
   $b_r = (k_v - k_0) - (k_1 - k_v)$ 
   $s_0 = k_1$ 
else
   $b_0 = k_1$ 
   $b_1 = k_v$ 
   $b_i = -1$ 
   $b_r = (k_1 - k_v) - (k_v - k_0)$ 
   $s_0 = k_0$ 
end if

```

---



---

**Algorithm 4** Iterating through a single split-dimension. See section 7.4.3 for details.

---

```

 $s = s_0$ 
if  $s = b_r$  then
  interleaved = true
else
  interleaved = false
end if
if atMostNestedLevel then
  action() = visitVoxel()
else
  action() = nestedLoop()
end if
for  $b = b_0; b \neq b_1; b = b + b_i$  do
  action( $b$ )
  if interleaved then
    action( $s$ )
     $s = s - b_i$ 
  else
    if  $b + b_i = b_r$  then
      interleaved = true
    end if
  end if
end for

```

---

However, many voxel-based volume rendering algorithms make use of some form of empty space skipping. In general, only voxels that will actually contribute to the rendering are compactly stored and other voxels are completely ignored. Because the significant voxels account for a very small percentage of the complete volume and no time is wasted even examining non-significant voxels, these schemes result in significantly faster rendering.

The interleaved split-dimension scheme detailed above performs explicit interleaving, i.e. we have counter variables iterating through all dimensions to ensure that the interleaving is correctly done. With many empty space skipping implementations, such an explicit interleaved traversal would partly negate the advantages of the space skipping, as non-significant voxel positions would still be traversed by counter variables in order to ensure a correct interleaving. Even if a particular voxel in one sub-volume is non-significant, the voxels at matching positions in other sub-volumes have to be rendered in an interleaved fashion.

For cases such as these, an implicit interleaving traversal is more appropriate. We have dubbed it “implicit” as no counter or traversal variable is involved. In this section we present a simple technique for implicit interleaving and, as we show in section 7.5, the implicit interleaving allows us to perform empty space skipping whilst correctly interleaving, without a measurable speed difference compared to the regular PBTF.

For any given axis permutation, we make use of space skipping only for the inner loop, i.e. the fastest changing dimension. Consequently, we only make use of implicit interleaving for the fastest changing dimension. For the outer loops, i.e. the slower changing dimensions, we make use of the standard explicit interleaving scheme described in section 7.4.3.

Algorithm 5 shows the implicit interleaving technique over one complete inner loop. This is performed for each combination of the two explicitly interleaved or non-interleaved outer loop counter variables.

At the start of the inner loop, the relevant space skipping voxel run for each partitioned sub-volume is determined and a pointer for each run is set to the start or the end of that run depending on the PBTF-required traversal direction. For each of these pointers, the distance to the partitioning plane orthogonal to the space skipping dimension is calculated by a simple subtraction.

The greatest distance is then determined. All current voxel run pointers that represent a voxel with distance equal to this greatest distance are rendered and then incremented or decremented depending on the various required traversal directions. These directions are always towards the partitioning plane, so the distances and the maximum distance are by definition decreasing. A new distance is calculated for each incremented or decremented voxel. If a pointer is incremented or decremented and the voxel that it represents subsequently crosses the partitioning plane, that pointer is deactivated. After rendering, incrementing and calculating new distances for the relevant voxels, we start again by determining the new maximum distance. The inner loop is terminated when all voxel pointers have been deactivated.

In contrast to explicit interleaving, this scheme never accesses or iterates over non-significant voxel positions.

Figure 7.9 shows an example of a single complete inner loop of implicit interleaving. Only a single  $z$ -slice is shown that has been partitioned into four sub-volumes in this case,

---

**Algorithm 5** The complete implicit interleaving inner loop.

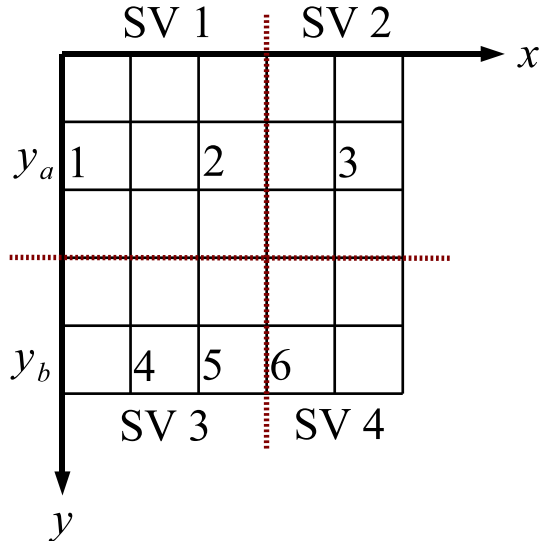
---

```

 $N \leftarrow$  number of sub-volumes (maximum 8)
for  $i = 0$  to  $N - 1$  do
   $p_i \leftarrow$  initial voxel pointer for sub-volume  $i$ 
  if IsValid( $p_i$ ) then
     $d_i \leftarrow$  distance of voxel  $p_i$  from sub-division
     $\Delta p_i \leftarrow$  increment and direction for sub-volume  $i$ 
  else
    Deactivate( $p_i$ )
  end if
end for
while IsActive( $p_0$ ) or IsActive( $p_1$ ) or ... or IsActive( $p_{N-1}$ ) do
   $d_{\max} = \max(\{d_0, d_1, \dots, d_{N-1}\})$  {only for active  $p_i$ 's}
  for  $i = 0$  to  $N - 1$  do
    if IsActive( $p_i$ ) and  $d_i = d_{\max}$  then
      renderVoxel( $p_i$ )
       $p_i = p_i + \Delta p_i$ 
       $d_i \leftarrow$  distance of voxel  $p_i$  from sub-division
      if  $d_i < 0$  then { $p_i$  has skipped across the sub-division}
        Deactivate( $p_i$ )
      end if
    end if
  end for
end while

```

---



**Figure 7.9:** Illustration of an example single inner loop of implicit interleaving. “SV” refers to sub-volume. In lines  $y_a$  and  $y_b$ , only the numbered voxels are stored in the space skipping data structures. The non-numbered voxels are completely discarded during a pre-processing step.

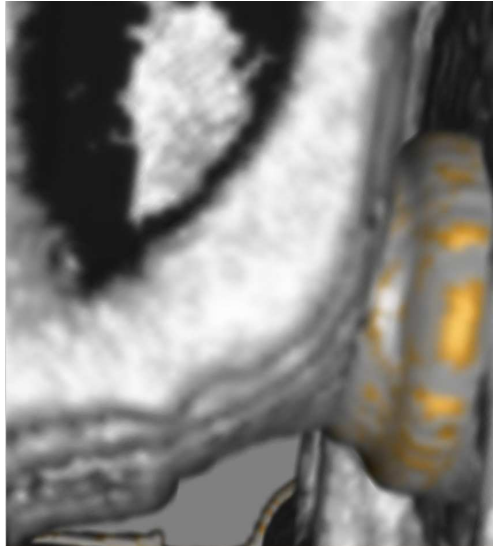
i.e. a “face-on” case according to section 7.3. The view axis is along the  $z$ -axis and into the page. The current  $y$ -coordinate is  $y_a$  in sub-volumes 1 and 2 and  $y_b$  in sub-volumes 3 and 4.  $y$  is being explicitly interleaved, but  $x$  is the space skipping dimension and will be implicitly interleaved.

For sub-volume 1, the voxel run pointer starts at voxel 1 and will iterate to the right, i.e. it will be incremented. For sub-volume 2, the voxel run pointer starts at 3 and will iterate to the left, i.e. it will be decremented. For sub-volumes 3 and 4 the voxel run pointers will start at voxel 4 and 6 respectively.

Voxel 1 is most distant from the vertical partitioning plane and no other voxels are equally distant. Voxel 1 is therefore rendered and the pointer for that voxel run is incremented to end up on voxel 2. Now the most distant voxels are 4 and 3 with equal distance, so they are rendered and their pointers are respectively incremented and decremented. The first of these pointers now points to voxel 5, but the second is deactivated, because it has jumped over the partition. There are three remaining voxel run pointers that point to voxels 2, 5 and 6. These are equally far away from the partitioning plane and so all are rendered. When their pointers are incremented, they all effectively jump over the vertical partitioning plane and are deactivated. Because all pointers are deactivated, the inner loop is complete and the next iteration of the  $y$ -variable can be initiated. The resultant voxel order for the  $y_a$  and  $y_b$  lines is 1, 3, 4, 2, 5 and 6, which is correctly interleaved.

This scheme is simple to implement and makes efficient use of the space skipping encod-





**Figure 7.10:** The same rendering as in Figure 7.3, but with the IP-PBTF ordering applied. The cross artefact has disappeared. In this case, the permutation was already correct, only the interleaving had to be applied.

ing whilst implicitly interleaving voxels. In principle, it should work with any space-skipping technique where the IP-PBTF is applicable.

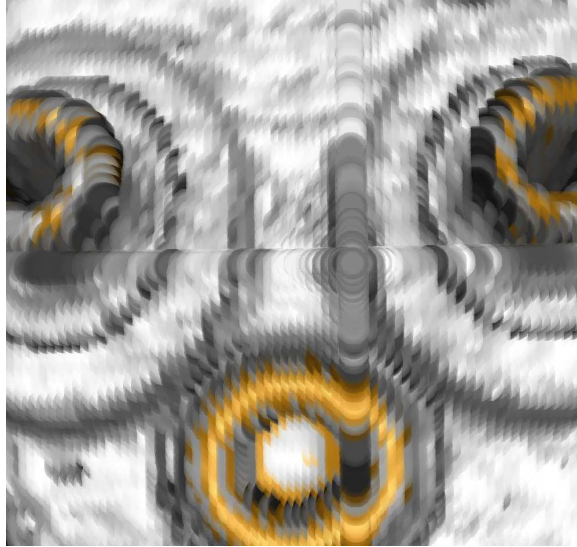
## 7.5 Results

We have implemented the IP-PBTF ordering as part of a specialised hardware-accelerated splatting implementation called “ShellSplatting” [11]. The IP-PBTF ordering is of course applicable to *any* object-order perspective volume rendering implementation where discrete voxels are projected.

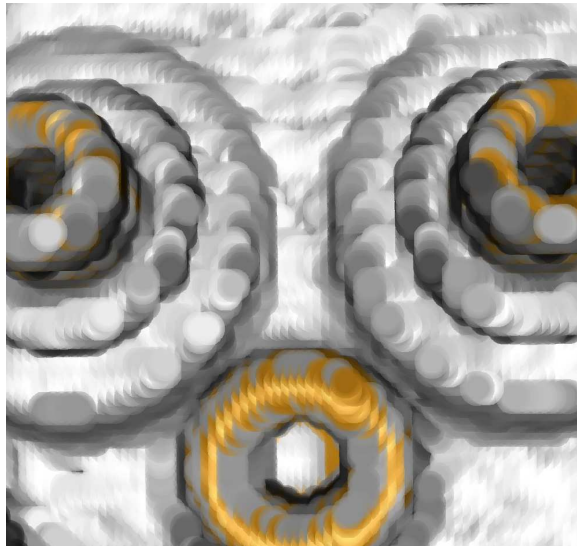
Figure 7.10 shows the same rendering as in figure 7.3, but with the IP-PBTF applied. The cross artefact has clearly disappeared. Figure 7.11 shows an extreme close-up rendering of the engine block with all splats rendered as circles instead of pre-integrated Gaussians in order to emphasise the nature of the cross artefact that occurs with PBTF ordering. Figure 7.12 shows the same close-up, but the IP-PBTF ordering has been applied instead.

Table 7.1 shows some speed timings for three well-known volume datasets<sup>2</sup>. These timings were performed on a 2.4GHz P4 with a GeForce4 graphics card. The image size was  $512 \times 512$  and we zoomed in so that the image was generously filled with the rendered volume. We tested BTF, PBTF and IP-PBTF orderings. The former two were unpermuted. The

<sup>2</sup>The aneurism data is courtesy of Philips Research, Hamburg and the engine block is courtesy of General Electric. Both were downloaded from <http://volvis.org/>. The CT Head is courtesy of Stanford University



**Figure 7.11:** A close-up of a rendering of the engine block dataset with circular splats to illustrate the nature of the cross artefact that occurs with standard PBTF visibility ordering.



**Figure 7.12:** The same close-up rendering as in Figure 7.11, but with the IP-PBTF ordering applied. The permutation has been corrected and the voxels have been rendered in an interleaved fashion.

Frame rate (frames per second)			
	Aneurism	CT Head	Engine Block
Ordering	$256^3$	$256^2 \times 113$	$256^2 \times 128$
BTF	49	12	12
PBTF	45	12	13
IP-PBTF	45	12	13

**Table 7.1:** Optimised splatter rendering frame rates for the different orderings for a  $512 \times 512$  rendered image. The IP-PBTF is just as fast as the PBTF, but yields a more correct ordering.

BTF was of course tested in an orthogonal projection setting. A sequence of 3610 different frames was rendered three times per ordering for all datasets, i.e. the complete sequence was run 27 times in total. For each ordering and dataset permutation, the average of the three tests was taken and then this average was rounded down to yield the final performance figure in frames per second.

The results show that the IP-PBTF extension has no measurable impact on the rendering performance of our splatting implementation. In other words, the IP-PBTF is just as fast as the PBTF *and* yields higher quality results. Our implementation utilises empty space skipping, so we have used implicit interleaving for the inner loop as discussed in section 7.4.4. Experiments with *explicit* interleaving in the inner loop resulted in 57% slower rendering for the “aneurism” dataset. From this we conclude that implicit interleaving is also crucial for an efficient implementation of the IP-PBTF that is compatible with empty space skipping. For the “CT Head” and “Engine Block” datasets, which have far less empty space than “aneurism”, the difference is much smaller, but implicit interleaving still yields the most efficient implementation.

## 7.6 Conclusions

In this chapter, we have discussed existing visibility orderings for object-order volume rendering based on discrete voxel projection. We have shown that, for the common case where voxel projections are larger than a single pixel, *none* of the existing orderings yields a correct BTF ordering for perspective projection.

We focused on the best of the existing orderings, namely the PBTF, and showed how and why it is not valid for cases where each projected voxel affects more than a single pixel. By studying the nature of its failure on constructed simple examples, we were able to determine the changes necessary to remedy these failures. The PBTF fails because it assumes that there is no interaction between the partitioned sub-volumes. By visiting all voxels in an interleaved fashion, i.e. alternatingly from all sub-volumes and by choosing a correct WBTF-like traversal for each sub-volume, the PBTF problems are remedied. We have dubbed the new ordering IP-PBTF, or Interleaved and Permuted Perspective Back-to-Front.

We analysed our changes by looking at circles of equidistance (spheres in 3D) with the

perspective view point as centre. This showed why a permuted interleaving of the PBTF was necessary. Coincidentally, it is also an effective way of illustrating the necessity of partitioning the volume into sub-volumes.

We then presented a possible implementation of iteration through an interleaved split-dimension. We also showed how to implement interleaving efficiently for empty space skipping implementations. The former technique is called explicit interleaving and the latter implicit interleaving.

We presented comparative timings of the BTF, PBTF and IP-PBTF orderings in an accelerated splatting implementation. The results show that the IP-PBTF is just as fast as the PBTF for our empty space skipping implementation, if explicit interleaving is used for the outer loops and implicit interleaving for the inner loop.

For projections no larger than a pixel IP-PBTF is *as good* as the traditional PBTF, as the interleaving and permutation still maintain the PBTF ordering as well. For larger-than-pixel voxel projections, it is clear that interleaving and permuting properly ensures a more strict BTF ordering during object-order perspective projection of discrete voxels, without explicit sorting. This observation, and the fact that these changes have no measurable impact on performance, lead us to conclude that the IP-PBTF is definitely to be preferred for the perspective object-order volume rendering of discrete voxels.

## Acknowledgements

This research is part of the DIPEX (Development of Improved endo-Prostheses for the upper EXtremities) program of the Delft Interfaculty Research Center on Medical Engineering (DIOC-9).

We would like to thank Ed Swan for enlightening and pleasant conversations and for his exceptionally clear exposition of PBTF and its proof.

Jorik Blaas can always be relied on to poke holes in one's theories.

---

## Conclusions and Future Work

---

### 8.1 Conclusions

In chapter 2 of this thesis we presented DeVIDE, a software platform for the rapid prototyping and deployment of visualisation and image processing algorithms. In the subsequent chapter, we discussed three applications where the software has been successfully applied: pre-operative planning for shoulder replacement, reconstruction and visualisation of microscopic sections of chorionic villi and finally, visualisation and measurement of the deformation of the pelvic floor muscles.

Chapter 4 introduced an approach for deriving accurate and topologically correct surfaces models describing the outer surfaces of the shoulder skeleton from CT data. These models can be used for visualisation, measurement and bio-mechanical modelling. The complete approach was prototyped and implemented with DeVIDE. The automatic surface extraction of a cadaveric scapula from CT data was compared with a manual segmentation by evaluating the Hausdorff distance between the two surfaces. The results show that the automatic method generates satisfactory results.

In the remaining three chapters of the thesis, we presented three techniques that make contributions to the field of volume visualisation, and more specifically direct volume rendering. The first, discussed in chapter 5, was a technique, with two variations, for generating real-time visual feedback during the transfer function specification process. This feedback significantly speeds up the search for a suitable transfer function. Chapter 6 documented Shell Splatting, a hardware-accelerated object-order direct volume rendering technique that is faster than splatting and generates higher quality images than shell rendering. The last of the three chapters presented an improved visibility ordering for perspective projection in

object-order volume rendering. Existing discrete voxel visibility orderings result in disturbing artefacts due to overlapping voxel projections. Our new ordering solved these problems and enabled us to use Shell Splatting in a perspective projection setting as well. All three of these techniques are most applicable to the volume rendering of musculo-skeletal medical data.

The following three subsections deal with three important themes that played an important role during this research.

### 8.1.1 Unifying visualisation and image processing

The three volume rendering techniques form part of the *mapping* and *rendering* stages of the visualisation pipeline that we discussed in the introduction. The shoulder skeleton segmentation approach is an exercise in image processing and can be neatly classified as contributing to the *filtering* stage of the visualisation pipeline. The exercise documented in this thesis leads us to the conclusion that image processing can and should form an integral part of any instance of the medical visualisation process. This integration is mutually advantageous to both visualisation and image processing. Results from the purely visualisation-oriented aspects are used to adjust the image processing aspects and vice versa. A good example of this cooperation is the *histogramSegment* module documented in section 4.5.1. It offers a powerful interactive segmentation tool that is based on both visualisation and image processing concepts. By adding closed curves to a coloured 2D histogram of image intensity and image gradient magnitude, the user can easily classify voxels.

From our experience, the easier it is for information to flow between the visualisation and image processing aspects, the faster the overall process converges to a satisfactory solution. This information flow can be facilitated by a number of factors: one of the easiest to implement is the use of a development platform that integrates visualisation and image processing functionality. In our case, DeVIDE fills this niche quite well. On the one hand, it allows a single operator to prototype and experiment with solutions from both fields. On the other hand, as a common framework, it allows visualisation and image processing experts, as well as researchers from other fields, to work together on the same problem.

### 8.1.2 Keeping the human in the loop

Throughout this work, the importance of semi-automatic methods for complex image processing and visualisation tasks became increasingly prominent. Especially in a clinical setting, there are very good reasons to keep the human in the processing loop. When working with clinicians and complex image processing or visualisation problems, minimal effort by the clinician during use translates to significantly less effort during algorithm development. A few minutes of manual interaction with an effective user interface can save months of development effort. In addition, the accuracy of the results have then been automatically checked by the clinician that makes use of the system. Finally, domain experts often prefer having some sense of control over a complex segmentation or measurement process. In short, it is

prudent to evaluate the possibilities for effective human interaction in complex visualisation problems.

For example, in the registration case study of section 3.2, getting the registration routines to find the optimal set of transforms for the whole volume automatically would have been prohibitively difficult, not to mention the amounts of time and effort it would have taken. Instead, we chose an effective interaction method where image pairs were registered automatically, but the operator could interrupt at any time to help an optimisation routine out of a local minimum or to finalise a promising registration. During the whole exercise, we had to interact with approximately 10% of the image pairs, which is an easily justifiable effort for a registered volume.

### 8.1.3 Using a very high level interpreted language

As explained in section 2.6.1 and in section 2.3, DeVIDE is programmed in Python, a very high-level interpreted language. Only processor intensive code is programmed in faster, lower level languages, for example C++ and Fortran.

By definition, a higher level language has a larger functionality to lines of code ratio. This, added to the fact that Python is robust, dynamically typed and offers introspection functionality, leads to significantly increased programmer productivity and a higher quality system.

Considering the rule that 20% of any program is responsible for 80% of the execution time [6] and that in our case, this 20% is by definition the processor intensive code, we can conclude that the choice of implementing in a very high level interpreted language has only advantages with regards to the traditional philosophy of implementing in a traditional compiled language, with or without an embedded interpreter.

Less than twenty years ago, experienced programmers implemented large systems in C and would, after profiling, implement processor-intensive sections in low level and processor-specific assembly language. The idea behind this philosophy has not changed, only its execution.

## 8.2 Future work

### 8.2.1 Medical visualisation

The segmentation approach described in chapter 4 needs to be refined. Due to the complications of segmenting bony structures from shoulders with joint space narrowing and bone density irregularities, our solution is quite complex. Although the protocol is documented, its execution does require some effort in the DeVIDE software. Parts of the process could be further automated. An interface that encapsulates the whole process and is suitable for use by clinicians will be implemented. The approach will be subjected to more testing.

The pre-operative planning functionality in DeVIDE is an experimental prototype that has been useful in investigating interaction and visualisation aspects of virtual shoulder re-

placement. This prototype will be further refined and developed to result eventually in a clinically-usable solution for the pre-operative planning of shoulder replacements.

The software will be extended with more biomechanical modelling functionality. As a first step, impingement detection will be added. Impingement refers to the phenomenon where different bones “collide” during motion, thus decreasing mobility. This detection will initially be based on the use of conformal prostheses, where the potential range of shoulder motion is relatively easy to calculate. This range of motion can be used along with a patient-specific shoulder skeleton segmentation and collision detection techniques to determine whether a particular prosthesis placement will result in impingement during shoulder motion.

A longer term goal is to integrate the Delft Shoulder and Elbow Model [91], or DSEM, with our pre-operative planning system. The DSEM is a bio-mechanical shoulder model that can be used to help predict the post-operative mobility of a patient shoulder. Currently the DSEM is based on a detailed analysis of a cadaveric shoulder. We will investigate the possibilities of using the results of our patient-specific shoulder segmentation to make the shoulder model patient-specific as well.

We will investigate other methods of giving feedback on the quality of a prosthesis placement. For example, showing, in real time, the amount of contact between cortical bone and the prosthesis would be useful.

The goal of integrating impingement detection, the DSEM and other feedback methods is to assist the surgeon during a pre-operative planning session. At any stage during the pre-operative planning, the surgeon can evaluate the virtual placement according to her own experience, with the help of the visualisation of the placement, but can also get various different kinds of feedback on the quality of the placement. Based on this feedback, adjustments can be made to prosthesis type, size and placement.

Research is still continuing on the exact design that will be used for the mechanical guidance device discussed in section 3.1.3. We will continue our research on using the results of a patient-specific segmentation to generate a patient-specific guidance device design automatically.

As mentioned in the chorionic villi sections reconstruction case study of section 3.2, the process could be greatly improved if the manual segmentation step were integrated with the rest of the automatic processing. This is another example where the integration, or unification, of the various aspects of the visualisation process can improve the results. We will continue extending DeVIDE and its interaction possibilities so that more of these aspects can be unified.

## 8.2.2 General visualisation techniques

Coupled views are visualisations of the same phenomenon or data that are somehow linked. For example, interaction in one view will transparently and immediately affect the visualisation in another coupled view. Coupled views represent a powerful approach to working with complex datasets. The *histogramSegment* module documented in section 4.5.1 is a simple example of this approach, but has proved to be effective. We will expand our investigation of



the coupled view approach by implementing a more generic coupled view architecture within DeVIDE and by applying this to more problem domains.

The DeVIDE architecture will be adapted to support transparent distributed processing. The existing separation between the front-end and the processing components will be increased so that modules can for instance be transparently uploaded to remote machines and executed there. This separation will increase the robustness of the system without affecting its flexibility in the distributed processing context.

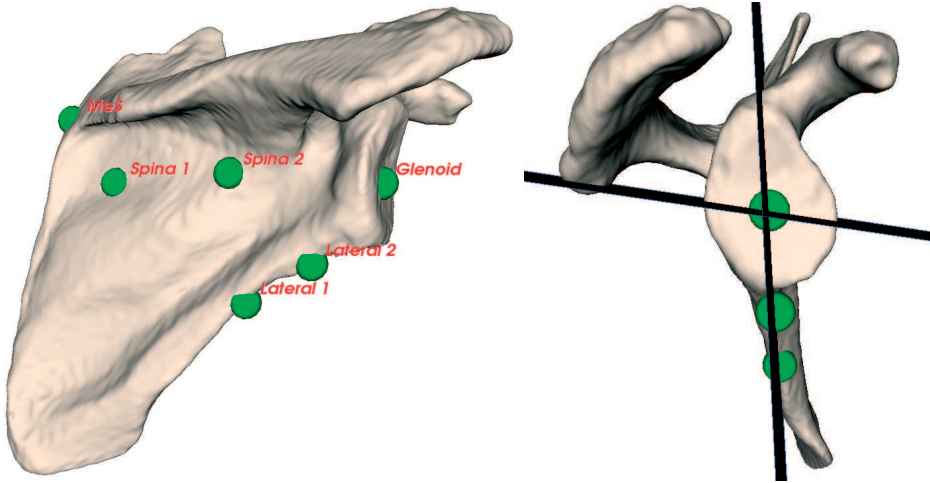
We remain interested in completely alternative approaches to visualisation system architectures and interfaces. We will investigate the possibilities of highly decentralised systems that support distributed processing and the integration of heterogeneous software and hardware components, whilst still remaining as easy to deploy and use as DeVIDE. We will also study ways of hiding the complexity of these systems from the end-user by creating intelligent interfaces that can guide the user in choosing the most suitable processing techniques based on knowledge about the problem or input data.



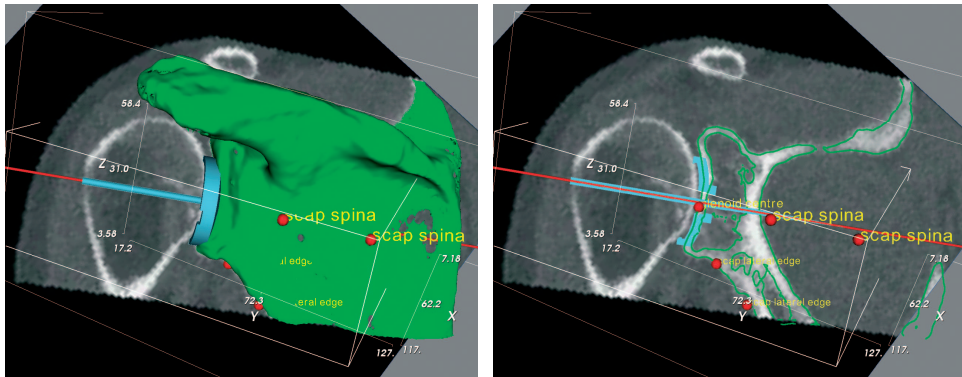
---

## Colour Figures

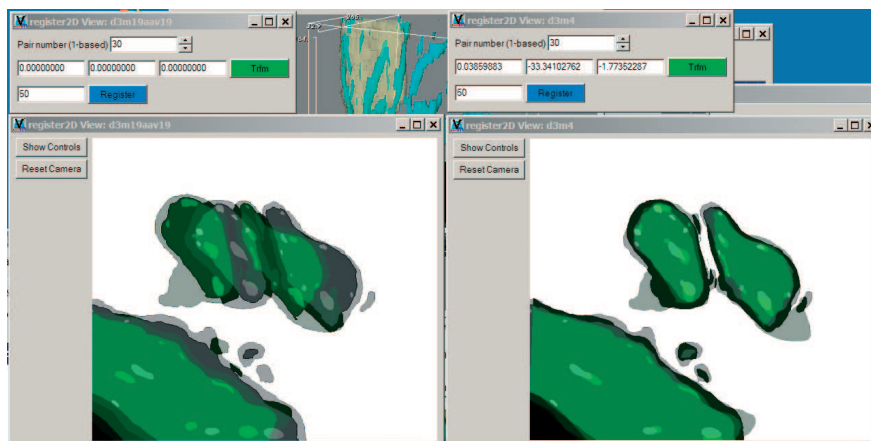
---



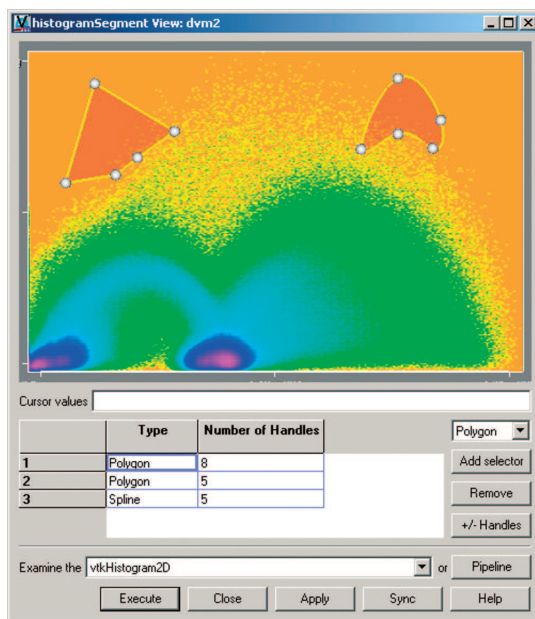
**Figure C.1:** On the left, a scapula is shown with the points that are required to place the two anatomical planes. On the right, the same scapula is shown with the resultant anatomical planes intersecting to form an insertion axis for the glenoid prosthesis.



**Figure C.2:** Part of the glenoid component planning functionality. On the left the complete models are shown and on the right only their contours. The contour view enables the user to judge whether the prosthesis has a good fit and is reminiscent of the conventional template-on-x-ray planning. The slice is anatomically oriented but can be moved to check all parts of the glenoid component.



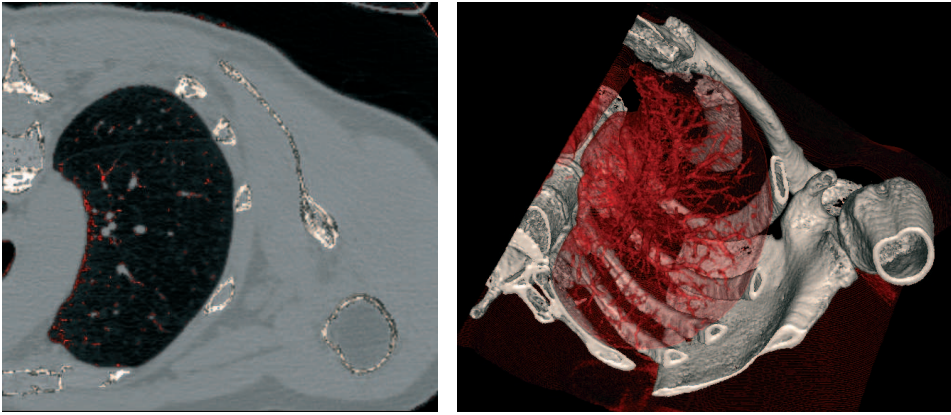
**Figure C.3:** The user-interface of the *register2d* DeVIDE module. On the left an unregistered image pair is shown and on the right the same pair is shown after registration.



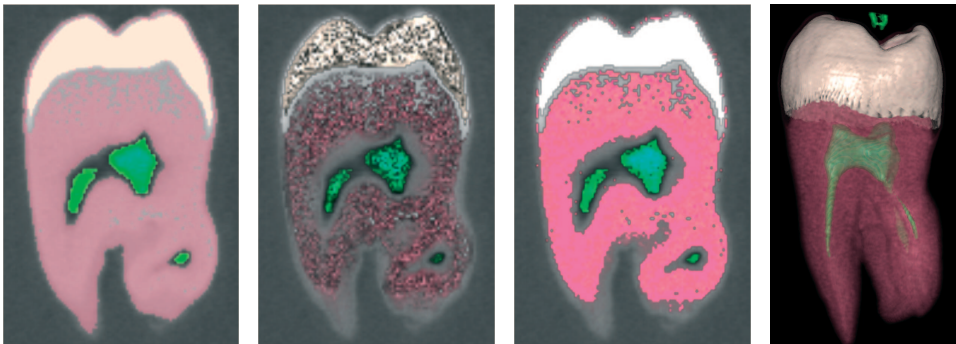
**Figure C.4:** Screen-shot of the Histogram Segmentation DeVIDE module. The user is delineating two areas on the histogram with a polygon and a spline. See section 4.5.1 for an explanation of the arc-like shapes that are visible in the histogram. The colour map is logarithmic and has been optimised to accentuate the different arc-like shapes in order to facilitate the segmentation.

This page intentionally left mostly blank.

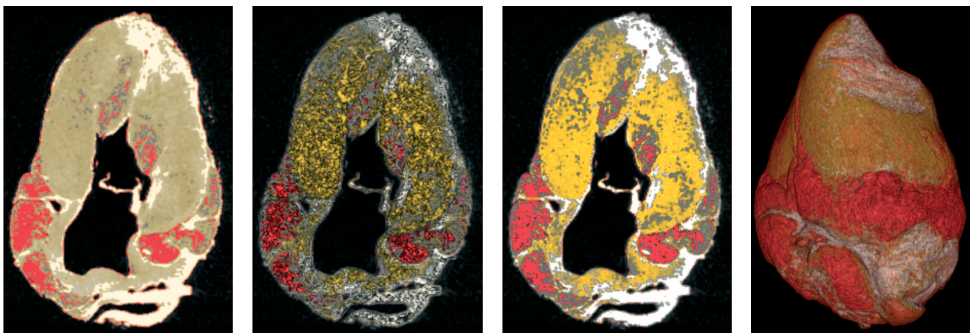
This page intentionally left mostly blank.



**Figure C.5:** Preview and direct volume rendering of shoulder CT-data showing bony and bronchial structures.

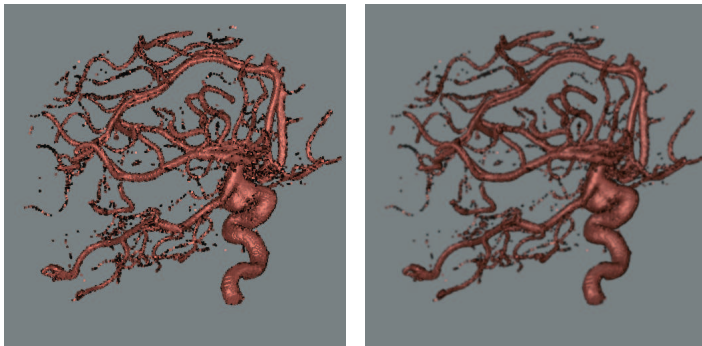


**Figure C.6:** Feedback with simple method, previews with and without shading and direct volume rendering of industrial CT-data of tooth.

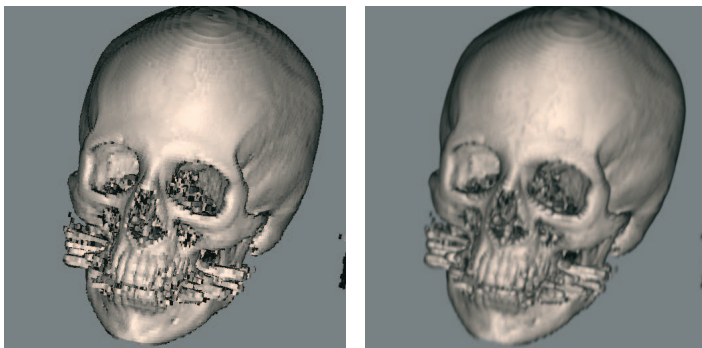


**Figure C.7:** Feedback with simple method, previews with and without shading and direct volume rendering of MRI-data of sheep heart.

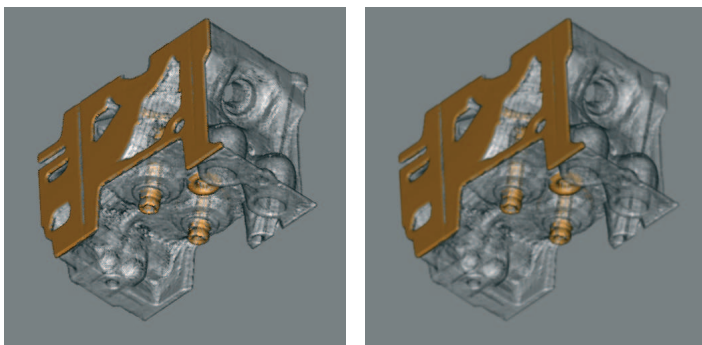




**Figure C.8:** ShellSplat rendering of rotational b-plane x-ray scan of the arteries of the right half of a human head, showing an aneurism. On the left is the fast rendering and on the right is the high quality version. Data from volvis.org courtesy of Philips Research, Hamburg, Germany.



**Figure C.9:** ShellSplat rendering of the Stanford CTHead data set. The fast rendering is on the left and the high quality is on the right. Note that this data set is anisotropically sampled.



**Figure C.10:** ShellSplat rendering of the engine block data set. The grey material has been made transparent. The data set was supplied by volvis.org and originally made by General Electric. Fast rendering on left, high quality on the right.



---

## Bibliography

---

- [1] Greg Abram and Lloyd Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of the 6th conference on Visualization '95*, page 263. IEEE Computer Society, 1995.
- [2] David P. Anderson. Hidden line elimination in projected grid surfaces. *ACM Transactions on Graphics*, 1(4):274–288, 1982.
- [3] N. Aspert, D. Santa-Cruz, and T. Ebrahimi. Mesh: Measuring error between surfaces using the Hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo 2002 (ICME)*, pages 705–708, 2002.
- [4] D. M. Beazley. Automated scientific software scripting with swig. *Future Gener. Comput. Syst.*, 19(5):599–609, 2003.
- [5] Giles Bertrand. Simple points, topological numbers and geodesic neighbourhoods in cubic grids. *Pattern Recognition Letters*, 15:1003–1011, October 1994.
- [6] Barry W. Boehm. Improving software productivity. *Computer*, 20(9):43–57, 1987.
- [7] Gunilla Borgefors, Ingela Nyström, and Gabriella Sanniti di Baja. Connected components in 3d neighbourhoods. In *Proceedings of The 10th Scandinavian Conference on Image Analysis*, pages 567–572, June 1997.
- [8] Charl P. Botha and Frits H. Post. A Visualisation Platform for Shoulder Replacement Surgery. In Silvia D. Olabarriaga, Wiro J. Niessen, and Frans Gerritsen, editors, *Interactive Medical Image Visualization and Analysis (IMIVA) - Satellite Workshop of MICCAI 2001*, pages 61–64, October 2001.

- [9] Charl P. Botha and Frits H. Post. Interactive Previewing for Transfer Function Specification in Volume Rendering. In D. Ebert, P. Brunet, and I. Navazo, editors, *Data Visualization 2002 (Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization)*. ACM SigGraph, 2002.
- [10] Charl P. Botha and Frits H. Post. New technique for transfer function specification in direct volume rendering using real-time visual feedback. In Seong K. Mun, editor, *Proceedings of the SPIE International Symposium on Medical Imaging*, volume 4681 - Visualization, Image-Guided Procedures, and Display, 2002.
- [11] Charl P. Botha and Frits H. Post. ShellSplatting: Interactive Rendering of Anisotropic Volumes. In G.-P. Bonneau, S. Hahmann, and C. D. Hansen, editors, *Data Visualization 2003 (Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization)*, pages 105–112, 2003.
- [12] Charl P. Botha and Frits H. Post. Improved perspective visibility ordering for object-order volume rendering. *The Visual Computer*, 2005.
- [13] Alexandra Branzan-Albu, Denis Laurendeau, Luc Hbert, Helene Moffet, Marie Dufour, and Christian Moisan. Image-guided analysis of shoulder pathologies: Modeling the 3d deformation of the subacromial space during arm flexion and abduction. In Dimitris Metaxas Stephane Cotin, editor, *International Symposium on Medical Simulation (ISMS 2004)*, volume 1 of *Lecture Notes in Computer Science*, pages 193–202, Cambridge, MA, USA, June 17-18 2004. Springer Verlag.
- [14] Ken Brodlie and Jason Wood. Recent Advances in Visualization of Volumetric Data. In *Eurographics State of the Art Reports*, pages 65–84, 2000.
- [15] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.
- [16] G.P. Carnielli, A.X. Falcão, and J. Udupa. Fast digital perspective shell rendering. In *12th Brazilian Symposium on Computer Graphics and Image Processing*, pages 105–111. IEEE, 1999.
- [17] C.Bajaj, V.Pascucci, and D.Schikore. The Contour Spectrum. In *Proc. IEEE Visualization '97*, pages 167–173, 1997.
- [18] Liviu Coconu and Hans-Christian Hege. Hardware-accelerated point-based rendering of complex scenes. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 43–52. Eurographics Association, 2002.
- [19] R.A. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *Proc. IEEE Visualization '93*, pages 261–266, 1993.
- [20] Roger A. Crawfis. Real-time slicing of data space. In *Proc. IEEE Visualization 1996*, pages 271–277, 1996.

- [21] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *Proc. SIGGRAPH '88*, pages 65–74. ACM Press, 1988.
- [22] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th international conference on Software engineering*, pages 149–159. IEEE Computer Society, 2003.
- [23] Gideon Frieder, Dan Gordon, and R. Anthony Reynolds. Back-to-front display of voxel-based objects. *IEEE Computer Graphics and Applications*, 5(1):52–60, 1985.
- [24] J. Goffin, K. Van Brussel, K. Martens, J. Vander Sloten, R. Van Audekercke, and M.H. Smet. Three-dimensional computed tomography-based, personalized drill guide for posterior cervical stabilization at c1–c2. *Spine*, 26(12):1343–1347, 2001.
- [25] J. Goutsias and S. Batman. Morphological methods for biomedical image analysis. In M. Sonka and J. M. Fitzpatrick, editors, *Handbook of Medical Imaging: Volume 2. Medical Image Processing and Analysis*, pages 175–272. SPIE Optical Engineering Press, 2000.
- [26] George J. Grevera, Jayaram K. Udupa, and Dewey Odhner. An Order of Magnitude Faster Isosurface Rendering in Software on a PC than Using Dedicated, General Purpose Rendering Hardware. *IEEE Transactions on Visualization and Computer Graphics*, 6(4):335–345, October-December 2000.
- [27] R.B. Haber and D.A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In G.M. Nielson, G. Shriver, and L.J. Rosenblum, editors, *Visualization in Scientific Computing*, pages 74–93. IEEE Computer Society Press, 1990.
- [28] Xiao Han, Chenyang Xu, and Jerry L. Prince. A topology preserving level set method for geometric deformable models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(6):755–768, 2003.
- [29] Taosong He, Lichan Hong, A. Kaufman, and H. Pfister. Generation of transfer functions with stochastic search techniques. In *Proc. IEEE Visualization '96*, pages 227–234, 489, 1996.
- [30] Paul S Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.
- [31] F. Vega Higuera, N. Sauber, B. Tomandl, C. Nimsky, G. Greiner, and P. Hastreiter. Enhanced 3d-visualization of intracranial aneurysms involving the skull base. In R.E. Ellis and T.M. Peters, editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003*, volume LNCS 2879, pages 256–263. Springer-Verlag Heidelberg, 2003.

- [32] Luis Ibanez, Will Schroeder, Lydia Ng, and Joshua Cates. *The ITK Software Guide*. Kitware Inc., 2003.
- [33] Štěpán Janda, Sjoerd de Blok, Victor P.M. van der Hulst, and Frans C.T. van der Helm. Loading effect of the weight of the internal organs on the pelvic floor in erect and in supine position. *Submitted to: American Journal of Obstetrics and Gynecology*, 2004.
- [34] Štěpán Janda, Sjoerd de Blok, Victor P.M. van der Hulst, and Frans C.T. van der Helm. Pelvic floor muscle displacement in relation to the level of the intra-abdominal pressure and muscle activation. *Submitted to: American Journal of Obstetrics and Gynecology*, 2004.
- [35] Yan Kang, Klaus Engelke, and Willi A. Kalender. A new accurate and precise 3-d segmentation method for skeletal structures in volumetric ct data. *IEEE Transactions on Medical Imaging*, 22(5):586–598, 2003.
- [36] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–333, 1987.
- [37] Gordon Kindlmann and James W. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. In *IEEE Symposium on Volume Visualization*, pages 79–86, 1998.
- [38] John H. Klippel and Paul A. Dieppe, editors. *Rheumatology*. Mosby, 1994.
- [39] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proc. IEEE Visualization 2001*, pages 255–262, 2001.
- [40] Andreas H. König and Eduard M. Gröller. Mastering Transfer Function Specification by using VolumePro Technology. In Toshiyasu L. Kunii, editor, *Proceedings of the 17th Spring Conference on Computer Graphics (SCCG)*, pages 279–286, 2001.
- [41] W. Krueger. The application of transport theory to visualization of 3D scalar data fields. In *Proc. IEEE Visualization '90*, pages 273–280, 481–2, 1990.
- [42] Philippe Lacroute and Marc Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proc. ACM SIGGRAPH '94*, pages 451–458, July 1994.
- [43] David Laur and Pat Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. In *Proc. SIGGRAPH '91*, pages 285–288. ACM Press, 1991.
- [44] O. De Leest, P.M. Rozing, L.A. Rozendaal, and F.C.T. Van der Helm. The influence of glenohumeral prosthesis geometry and placement on shoulder muscle forces. *Clinical Orthopedics*, 330:222–233, 1996.

- [45] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.
- [46] Babette A.M. Lisman and Niek Exalto. Early human nutrition and chorionic villous vascularization. *Middle East Fertility Society Journal*, 4(2), 1999.
- [47] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface construction algorithm. In *Proc. of ACM SIGGRAPH*, pages 163–169. Association for Computing Machinery, 1987.
- [48] H. Malchau, P. Herberts, P. Söderman, and A. Odén. Prognosis of total hip replacement. Update and validation of results from swedish national hip arthroplasty registry 1979–1998. In *Scientific exhibition presented at the 67th AAOS meeting*, 2000.
- [49] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryll, J. Seims, and S. Shieber. Design galleries: a general approach to setting parameters for computer graphics and animation. In *Proc. ACM SIGGRAPH '97*, pages 389–400, 1997.
- [50] Sean Mauch. *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Institute of Technology, 2003.
- [51] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.
- [52] Nelson L. Max. Sorting for polyhedron compositing. In H. Hagen, H. Müller, and G.M. Nielson, editors, *Focus on Scientific Visualization*, pages 259–268. Springer-Verlag, 1993.
- [53] Michael Meissner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A Practical Evaluation of Popular Volume Rendering Algorithms. In *Proc. Volume Visualization and Graphics Symposium*, pages 81–90, 2000.
- [54] Klaus Mueller and Roger Crawfis. Eliminating Popping Artifacts in Sheet Buffer-Based Splatting. In *Proc. IEEE Visualization '98*, pages 239–245, 1998.
- [55] Klaus Mueller, Naeem Shareef, Jian Huang, and Roger Crawfis. High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels. *Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.
- [56] National Electrical Manufacturers' Association. NEMA Standard PS3: Digital Imaging and Communications in Medicine (DICOM), 2000.
- [57] Jeff Orchard and Torsten Möller. Accelerated Splatting using a 3D Adjacency Data Structure. In *Proc. Graphics Interface*, pages 191–200, June 2001.

- [58] S. Osher and J.A. Sethian. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton–Jacobi Formulations. *Journal of Computational Physics*, 79:12–49, 1988.
- [59] J. K. Ousterhout. Scripting: higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
- [60] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Professional, 1994.
- [61] Steven Gregory Parker. *The SCIRun problem solving environment and computational steering software system*. PhD thesis, The University of Utah, 1999.
- [62] David Parsons, Awais Rashid, Andreas Speck, and Alexandru Telea. A ”framework” for object oriented frameworks design. In *Proceedings of Technology of Object-Oriented Languages and Systems, 1999.*, pages 141–151, 1999.
- [63] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.
- [64] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L.S. Avila, K.M. Raghunathan, R. Machiraju, and Jinho Lee. The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21(1):16–22, Jan-Feb 2001.
- [65] Bui Tuong Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, June 1975.
- [66] Thomas Porter and Tom Duff. Compositing Digital Images. In *Proc. SIGGRAPH ’84*, volume 18, pages 253–259, July 1984.
- [67] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.
- [68] Fons Rademakers and Rene Brun. Root: An object-oriented data analysis framework. *Linux Journal*, (51), 1998.
- [69] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.
- [70] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware ’00*, pages 109–118,147, 2000.
- [71] Szymon Rusinkiewicz and Marc Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes . In Kurt Akeley, editor, *Proc. SIGGRAPH 2000*, pages 343–352, 2000.



- [72] Michel F. Sanner, Daniel Stoffer, and Arthur J. Olson. Viper, a visual programming environment for python. In *Proceedings of the 10th International Python Conference*, pages 103–115, February 2002.
- [73] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit*. Prentice Hall PTR, 2nd edition, 1999.
- [74] Iwo Serlie, Roel Truyen, Jasper Florie, Frits Post, Lucas van Vliet, and Frans Vos. Computed cleansing for virtual colonoscopy using a three-material transition model. In R.E. Ellis and T.M. Peters, editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003*, volume LNCS 2879, pages 175–183. Springer-Verlag Heidelberg, 2003.
- [75] J.A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 2nd edition, 1999.
- [76] B. Stenger, P. R. S. Mendonça, and R. Cipolla. Model based 3D tracking of an articulated hand. In *Proc. Conf. Computer Vision and Pattern Recognition*, volume II, pages 310–315, Kauai, USA, December 2001.
- [77] B. Stenger, P. R. S. Mendonça, and R. Cipolla. Model-based hand tracking using an unscented kalman filter. In *Proc. British Machine Vision Conference*, volume I, pages 63–72, Manchester, UK, September 2001.
- [78] J. Edward Swan. *Object-order Rendering of Discrete Objects*. PhD thesis, The Ohio State University, 1998.
- [79] G. Taubin. Optimal surface smoothing as filter design. Technical Report RC-20404 (#90237), IBM T.J. Watson Research Center, 1996.
- [80] G. Taubin. Geometric signal processing on polygonal meshes. In *STAR – State of the Art Report, Eurographics 2000*, 2000.
- [81] Alexandru Telea and Jarke J van Wijk. SMARTLINK: An agent for supporting dataflow application construction. In *Proceedings of Joint Eurographics, IEEE TCVG Symposium on Visualization 2000*, pages 189–198, 2000.
- [82] Alexandru C. Telea. *Visualisation and Simulation with Object-Oriented Networks*. PhD thesis, Technische Universiteit Eindhoven, 2000.
- [83] M.E Torchia, R.H. Cofield, and C.R. Settergren. Total shoulder arthroplasty with the neer prosthesis: long-term results. *Journal of Shoulder and Elbow Surgery*, 6(6):495–505, 1997.

- [84] Marie-Eve Tremblay, Alexandra Branzan-Albu, Denis Laurendeau, and Luc Hbert. Integrating region and edge information for the automatic segmentation for interventional magnetic resonance images of the shoulder complex. In *1st Canadian Conference on Computer and Robot Vision (CRV2004)*, volume 1, pages 279–286, London, Ontario, Canada, May 17-19 2004. IEEE Computer Society.
- [85] Jayaram K. Udupa and Dewey Odhner. Fast Visualization, Manipulation, and Analysis of Binary Volumetric Objects. *IEEE Computer Graphics and Applications*, 11(6):53–62, November 1991.
- [86] Jayaram K. Udupa and Dewey Odhner. Shell Rendering. *IEEE Computer Graphics and Applications*, 13(6):58–67, 1993.
- [87] C. Upson, T Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, pages 30–42, July 1989.
- [88] Edward R. Valstar, Charl P. Botha, Marjolein van der Glas, Piet M. Rozing, Frans C.T. van der Helm, Frits H. Post, and Albert M. Vossepoel. Towards computer-assisted surgery in shoulder joint replacement. *ISPRS Journal of Photogrammetry and Remote Sensing*, 56(5–6):326–337, August 2002.
- [89] E.R. Valstar, K van Brussel, B.L. Kaptein, B.C. Stoel, and P.M. Rozing. CT-based personalized templates for accurate glenoid prosthesis placement in total shoulder arthroplasty. In *Proceedings of The 3rd Annual Meeting of the International Society for Computer Assisted Orthopaedic Surgery*, 2003.
- [90] Marjolein van der Glas, Frans M. Vos, Charl P. Botha, and Albert M. Vossepoel. Determination of Position and Radius of Ball Joints. In Milan Sonka, editor, *Proceedings of the SPIE International Symposium on Medical Imaging*, volume 4684 - Image Processing, 2002.
- [91] Frans C.T. van der Helm. A finite element musculo-skeletal model of the shoulder mechanism. *Journal of Biomechanics*, 27(5):551–569, 1994.
- [92] Guido van Rossum. *Python Reference Manual*. Python Software Foundation, April, 2001.
- [93] Luc Vincent. Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms. *IEEE Transactions on Image Processing*, 2(2):176–201, 1993.
- [94] Luc Vincent and Pierre Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, June 1991.

- [95] Carien Vis, Ellen Everhardt, Jan te Velde, and Niek Exalto. Microscopic investigation of villi from chorionic villous sampling. *Human Reproduction*, 13(10):2954–2957, 1998.
- [96] L. Westover. Footprint evaluation for volume rendering. In *Proc. SIGGRAPH '90*, pages 367–376. ACM Press, 1990.
- [97] Lee Westover. Interactive volume rendering. In *Proceedings of the Chapel Hill workshop on Volume visualization*, pages 9–16. ACM Press, 1989.
- [98] Peter L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.
- [99] M.A. Wirth and C.A. Rockwood. Complications of shoulder arthroplasty. *Clinical Orthopaedics*, (307):47–69, 1994.
- [100] C. Xu and J.L. Prince. Snakes, shapes and gradient vector flow. *IEEE Transactions on Image Processing*, 7(3):359–369, March 1998.
- [101] Roni Yagel, David S. Ebert, James N. Scott, and Yair Kurzion. Grouping Volume Renderers for Enhanced Visualization in Computational Fluid Dynamics. *Transactions on Visualization and Computer Graphics*, 1(2):117–132, 1995.
- [102] Reza A. Zoroofi, Yoshinobu Sato, Toshihiko Sasama, Takashi Nishii, Nobuhiko Sugano, Kazuo Yonenobu, Hideki Yoshikawa, Takahiro Ochi, and Shinichi Tamura. Automated segmentation of acetabulum and femoral head from 3-d ct images. *IEEE Transactions on Information Technology in Biomedicine*, 7(4):329–343, 2003.
- [103] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proc. SIGGRAPH 2001*, pages 371–378. ACM Press, 2001.
- [104] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. EWA splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.



---

## List of Figures

---

2.1	The prototyping phase can be seen as an iterative and incremental ADIT trajectory: the problem is analysed and a solution is designed and subsequently implemented. If testing indicates that the solution is suitable, work stops. If not, we return to a previous stage to refine the process with newly-gained information. Such a return to a previous stage for refinement can happen at the design and implementation stages as well. . . . .	8
2.2	Figure showing the DeVIDE visual programming interface to the internal representation of a simple network. DICOM data is being read by the “dicomRDR” module and then thresholded by the “doubleThreshold” module. A 3D region growing is subsequently performed on the thresholded data by the “seedConnect” module, starting from seed points indicated by the user. The result of this region growing is volume rendered with the “shellSplatSimple” module. . . . .	11
2.3	Alternative representation of the simple network shown in figure 2.2. All explicit algorithm parameters are also shown: this is useful when documenting experiments. . . . .	11
2.4	DeVIDE network that extracts different types of edges from a super-quadric mesh. The mesh and the extracted edges are visualised together in the <i>slice3d-VWR</i> module. . . . .	18
2.5	The Graph Editor’s module palette (shortened) showing the module categories at the top, with the <i>userModules</i> category selected. Modules in the currently selected categories are shown in the bottom list. A module can belong to more than one category. Multiple categories can be selected. . . . .	19
2.6	The DeVIDE network shown in figure 2.4 with the <i>myTubeFilter</i> module integrated. . . . .	19

2.7	A visualisation that makes use of the newly-created <i>myTubeFilter</i> module to emphasise discontinuities in the sample mesh. . . . .	20
2.8	The architecture, design and implementation continuum: from concreteness and high-detail to abstraction and low-detail. . . . .	21
2.9	Diagram of the high-level DeVIDE architecture. The Module Manager is the main external interface. . . . .	22
2.10	The user type and interaction continua. Interaction can take place at any level of abstraction. Each interaction type is more suitable to a specific user type, but no clear partitions can be made. The introspection interaction modality makes interaction possible at all points. . . . .	26
2.11	A model of the flow of module state information between the user interface, the module configuration data structure and the underlying logic, i.e. the actual implementation of the module algorithm. Communication with the Module Manager is also shown. The module API methods that drive the flow are also indicated. . . . .	32
2.12	The DeVIDE demand-driven execution model. Due to the simple update convention, output data update requests propagate backwards through any given network until the source module is reached. Processing then propagates forwards until it reaches the final output object. . . . .	34
2.13	A sample introspection session with the main DeVIDE Python Introspection window. . . . .	35
2.14	Screen-shot of an example dialog created by the scripted configuration module mixin. An example tool-tip is shown as the mouse pointer hovers over the “Model bounds” text input widget. The standard introspection widgets can be seen on the second row from the bottom. The standard module action buttons are on the bottom row. . . . .	38
2.15	Screen-shot of sample introspection session with a VTK object after it has been selected from the object drop-down list on the second to last row of the dialogue box in figure 2.14. . . . .	39
3.1	The skeletal structures of the shoulder. This figure is courtesy of the Delft Shoulder Group. . . . .	46
3.2	CT-derived visualisation of the bony-structures in the human shoulder. Both the humeral head and glenoid prostheses have been virtually implanted, i.e. a total shoulder replacement. . . . .	47
3.3	On the left, a scapula is shown with the points that are required to place the two anatomical planes. On the right, the same scapula is shown with the resultant anatomical planes intersecting to form an insertion axis for the glenoid prosthesis. See colour figure C.1. . . . .	50

3.4	Part of the glenoid component planning functionality. On the left the complete models are shown and on the right only their contours. The contour view enables the user to judge whether the prosthesis has a good fit and is reminiscent of the conventional template-on-x-ray planning. The slice is anatomically oriented but can be moved to check all parts of the glenoid component. See colour figure C.2. . . . .	50
3.5	The latest design of the glenoid drill guide visualised on a surface model of a scapula. Illustration courtesy of Liesbet Goossens and Bart de Schouwer, Katholieke Universiteit Leuven, Belgium. . . . .	52
3.6	An earlier mould design that is in the process of being constructed by DeVIDE according to a set procedure, the patient-specific scapular surface and the final pre-operative planning parameters. . . . .	53
3.7	A digitised serial section of the chorionic tissue. The large structure covering almost half of the left image border is a villus. The circular structures in its interior are vessels. . . . .	55
3.8	The section shown in figure 3.7 after manual delineation and filling with grey levels. . . . .	55
3.9	The DeVIDE modules and network that were used to perform the registration. <i>imageStackRDR</i> and <i>transformStackRDR</i> read collections of input images and image pair transforms respectively. <i>register2D</i> encapsulates the registration user interface. <i>transform2D</i> applies all derived transformations to all input images to generate a reconstructed 3D volume. <i>transformStackWRT</i> writes all derived transforms to disc. <i>vtiWRT</i> saves the resultant 3D volume to disc. <i>slice3dVWR</i> functions as the 3D visualisation user interface. . . . .	57
3.10	The user-interface of the <i>register2d</i> DeVIDE module. On the left an unregistered image pair is shown and on the right the same pair is shown after registration. See colour figure C.3. . . . .	57
3.11	An example of a visualisation of the reconstructed slices. A single villus and two of its vessels are visualised. . . . .	58
3.12	An example coronal slice from one of the MRI datasets. The arrows indicate the pelvic floor muscles. The surface shown in 3.14 describes the inner surface of these muscles in 3D. . . . .	59
3.13	The signed distance field generated from the delineated curve. Above and to the sides the distance values are positive (i.e. the non-hatched areas) . . . . .	60
3.14	Example surface describing inner border of pelvic floor. Image courtesy of Stepan Janda and created with DeVIDE. . . . .	61
4.1	A cadaver scapula. In the centre, it is thin enough to be translucent. Together with the partial volume effect, this leads to extra complications during segmentation. . . . .	65

- 4.2 An axial slice from a shoulder CT dataset. In the top-right part of the image, the extreme thinness of the scapula and the partial volume effect results in voxels where the intensity is far lower than the expected intensity for bone. In extreme cases, this can result in parts of bony structures becoming almost invisible in CT images. . . . . 65
- 4.3 An axial slice from an arthritic shoulder CT dataset showing a humeral head and a scapula that appear fused together. This is due to the joint space being extremely narrow and the limited resolution of the CT scanner. . . . . 66
- 4.4 A flow chart illustrating our approach to the segmentation of the skeletal structures of the shoulder. Ovals signify data and blocks signify operations. The *A* and *B* blocks show two distinct ways to generate the initial conservative bone mask. *TP* signifies *Topology Preserving*. . . . . 69
- 4.5 Screen-shot of the Histogram Segmentation DeVIDE module. The user is delineating two areas on the histogram with a polygon and a spline. See section 4.5.1 for an explanation of the arc-like shapes that are visible in the histogram. The colour map is logarithmic and has been optimised to accentuate the different arc-like shapes in order to facilitate the segmentation. See colour figure C.4. . . . . 70
- 4.6 An ideal step edge, or material transition, the Gaussian representing the band-limited frequency response of the measurement process and the resultant measured edge as well as its normalised derivative with respect to position. . . . . 72
- 4.7 An example histogram segmentation result. This is the same slice as shown in figure 4.2. . . . . 73
- 4.8 The scapula has been selected from the connected component labeling of an initial bone segmentation shown in figure 4.7. . . . . 74
- 4.9 A simple example of binary dilation and erosion. (a) shows the original binary image *I* (black pixels denote points) with at its bottom right the  $5 \times 5$  rectangular structuring element *B*. (b) shows the dilation of *I* with *B* and (c) shows the erosion of *I* with *B*. In (d), all images are shown together to illustrate how dilation expands *I* at its boundaries and erosion eats *I* away at its boundaries. . . . . 75
- 4.10 An example of morphological reconstruction of a mask image *I* (light grey) from marker image *J* (dark grey). Contiguous objects of the mask *I* that are marked by pixels of the marker *J* are extracted. . . . . 76
- 4.11 A slice of the conservative structure mask of the scapula, as generated by method A. . . . . 77
- 4.12 A slice of an example Canny edges volume. . . . . 79
- 4.13 A slice of the scapula edge features. This is after binary reconstruction of the Canny edges volume from the conservative structure mask of the scapula and inverse masking with the structure mask of the humerus. The isolated structure at the upper left is also part of the scapula. . . . . 79



4.14	Segmented outer surface of the scapula from the data shown in figure 4.2. The transparent outer surface is the initial contour and the inner surface is the final contour delineating the scapula. The hole that is visible is due to the truncation of the CT dataset. . . . .	85
4.15	Segmented outer surfaces of the scapula and humerus from the data shown in figure 4.3. . . . .	85
5.1	Illustration of 2D raycasting showing the role of the transfer function. . . . .	90
5.2	Behaviour of $f(A_t) = 1 - e^{-\frac{A_t}{\tau}}$ for $\tau = 0.25$ . $A_t$ is the instantaneous opacity, $f(A_t)$ is a compensated opacity that can be used in sliced-based previews. $\tau$ in this context is a parameter-less time-constant. . . . .	95
5.3	An illustration of the frequency distribution for a single $(x,y)$ position. The distribution bin containing voxel scalar value $v(x,y)$ is found with a binary search. Subsequently, bins to the left and right of the found bin are merged according to the similarity of the optical properties they represent. After merging, the number of estimated voxels $N$ can be calculated by adding up the number of voxels in all merged bins. . . . .	98
5.4	Our implementation of a DVR transfer function editing widget. The user can add and manipulate an arbitrary number of points in the piecewise continuous lines that represent the hue, saturation, value, scalar opacity and gradient opacity transfer functions. Our method does of course extend to other more complex transfer function representations, e.g. Bézier curves. . . . .	100
5.5	Preview and direct volume rendering of shoulder CT-data showing bony and bronchial structures. See colour figure C.5. . . . .	101
5.6	Feedback with simple method, previews with and without shading and direct volume rendering of industrial CT-data of tooth. See colour figure C.6. . . . .	101
5.7	Feedback with simple method, previews with and without shading and direct volume rendering of MRI-data of sheep heart. See colour figure C.7. . . . .	101
6.1	Illustration of the $P$ and $D$ data structures. $P$ contains, for every $(y,z)$ tuple, an index and a run-length into $D$ . $D$ contains at each position the information to render a single shell voxel. Voxels that do not contribute to the rendering are not stored at all. . . . .	109
6.2	Illustration of the calculation of the reconstruction function bounding function in voxel space, transformation to world space and projection space and the subsequent “flattening” and transformation back to voxel space. . . . .	111
6.3	ShellSplat rendering of rotational b-plane x-ray scan of the arteries of the right half of a human head, showing an aneurism. On the left is the fast rendering and on the right is the high quality version. Data from volvis.org courtesy of Philips Research, Hamburg, Germany. See colour figure C.8. . . . .	113
6.4	ShellSplat rendering of the Stanford CTHead data set. The fast rendering is on the left and the high quality is on the right. Note that this data set is anisotropically sampled. See colour figure C.9. . . . .	114

- 6.5 ShellSplat rendering of the well-known engine block data set. The grey material has been made transparent. Data set supplied by volvis.org, originally made by General Electric. Fast rendering on left, high quality on the right. See colour figure C.10. . . . . 114
- 7.1 A perspective splatting of the Stanford CT Head dataset from the same viewpoint with three different orderings: traditional BTF, PBTF and our IP-PBTF. 119
- 7.2 Diagram illustrating BTF and WBTF ordering for parallel projection. . . . . 121
- 7.3 Close-up of splatting of “engine block” dataset showing the cross artefact caused by PBTF ordering. The cross artefact occurs at the volume subdivision boundaries due to the fact that splats overlap in screen space. . . . . 122
- 7.4 A simple 2D example showing how the PBTF visibility ordering is incorrect for cases where the voxel projection is larger than a single pixel. Each numbered block represents a voxel. . . . . 126
- 7.5 Interleaving the ordering of the voxels in the sub-volumes solves the problem shown in Figure 7.4. . . . . 126
- 7.6 Sub-volume interleaving by itself is not sufficient to remedy the PBTF visibility problems. In this simple case, in spite of interleaving, voxels 3 and 4 have been incorrectly composited before voxels 5 and 6. . . . . 127
- 7.7 Interleaving between sub-volumes *and* selecting a suitable axis permutation remedies the PBTF problems in this example. . . . . 128
- 7.8 Circles (spheres in 3D) of equidistance from the view point. It is clear that the  $y$  should be partitioned and that each partition should be traversed in the opposite direction in order to maintain a BTF ordering. This illustration also helps to show why interleaving and permutation are required. See the text for more detail. “SV 1” and “SV 2” refer to “Sub-volume 1” and “Sub-volume 2” respectively. . . . . 129
- 7.9 Illustration of an example single inner loop of implicit interleaving. “SV” refers to sub-volume. In lines  $y_a$  and  $y_b$ , only the numbered voxels are stored in the space skipping data structures. The non-numbered voxels are completely discarded during a pre-processing step. . . . . 134
- 7.10 The same rendering as in Figure 7.3, but with the IP-PBTF ordering applied. The cross artefact has disappeared. In this case, the permutation was already correct, only the interleaving had to be applied. . . . . 135
- 7.11 A close-up of a rendering of the engine block dataset with circular splats to illustrate the nature of the cross artefact that occurs with standard PBTF visibility ordering. . . . . 136
- 7.12 The same close-up rendering as in Figure 7.11, but with the IP-PBTF ordering applied. The permutation has been corrected and the voxels have been rendered in an interleaved fashion. . . . . 136

- C.1 On the left, a scapula is shown with the points that are required to place the two anatomical planes. On the right, the same scapula is shown with the resultant anatomical planes intersecting to form an insertion axis for the glenoid prosthesis. . . . . 146
- C.2 Part of the glenoid component planning functionality. On the left the complete models are shown and on the right only their contours. The contour view enables the user to judge whether the prosthesis has a good fit and is reminiscent of the conventional template-on-x-ray planning. The slice is anatomically oriented but can be moved to check all parts of the glenoid component. 146
- C.3 The user-interface of the *register2d* DeVIDE module. On the left an unregistered image pair is shown and on the right the same pair is shown after registration. . . . . 147
- C.4 Screen-shot of the Histogram Segmentation DeVIDE module. The user is delineating two areas on the histogram with a polygon and a spline. See section 4.5.1 for an explanation of the arc-like shapes that are visible in the histogram. The colour map is logarithmic and has been optimised to accentuate the different arc-like shapes in order to facilitate the segmentation. . . . 147
- C.5 Preview and direct volume rendering of shoulder CT-data showing bony and bronchial structures. . . . . 150
- C.6 Feedback with simple method, previews with and without shading and direct volume rendering of industrial CT-data of tooth. . . . . 150
- C.7 Feedback with simple method, previews with and without shading and direct volume rendering of MRI-data of sheep heart. . . . . 150
- C.8 ShellSplat rendering of rotational b-plane x-ray scan of the arteries of the right half of a human head, showing an aneurism. On the left is the fast rendering and on the right is the high quality version. Data from volvis.org courtesy of Philips Research, Hamburg, Germany. . . . . 151
- C.9 ShellSplat rendering of the Stanford CTHead data set. The fast rendering is on the left and the high quality is on the right. Note that this data set is anisotropically sampled. . . . . 151
- C.10 ShellSplat rendering of the engine block data set. The grey material has been made transparent. The data set was supplied by volvis.org and originally made by General Electric. Fast rendering on left, high quality on the right. . . 151



---

## List of Tables

---

2.1	A selection of existing frameworks and some of their distinguishing characteristics. <i>Module Spec.</i> refers to module specification, and is an indication of how new modules are created for the relevant framework. In the <i>Interpreter</i> column, <i>ext</i> and <i>emb</i> signify <i>extended</i> and <i>embedded</i> respectively. . . . .	16
4.1	Symmetric Hausdorff distance between the manually and automatically segmented meshes. . . . .	86
6.1	ShellSplatter rendering frame rates for three example data sets. Each data set's resolution is shown along with the percentage and number of voxels that are actually stored in the shell rendering data structures. . . . .	115
7.1	Optimised splatter rendering frame rates for the different orderings for a $512 \times 512$ rendered image. The IP-PBTF is just as fast as the PBTF, but yields a more correct ordering. . . . .	137



---

## Summary

---

This thesis presents a flexible software platform for medical visualisation and image processing, a technique for the segmentation of the shoulder skeleton from CT data and three techniques that make contributions to the field of direct volume rendering.

Our primary goal was to investigate the use of visualisation techniques to assist the shoulder replacement process. This motivated the need for a flexible environment within which to test and develop new visualisation and also image processing techniques with a medical focus. The Delft Visualisation and Image processing Development Environment, or DeVIDE, was created to answer this need. DeVIDE is a graphical data-flow application builder that combines visualisation and image processing techniques, supports the rapid creation of new functional components and facilitates a level of interaction with algorithm code and parameters that differentiates it from similar platforms.

For visualisation, measurement and pre-operative planning, an accurate segmentation from CT data of the bony structures of the shoulder is required. Due to the complexity of the shoulder joint and the fact that a method was required that could deal with diseased shoulders, existing techniques could not be applied. In this thesis we present a suite of techniques for the segmentation of the skeletal structures from CT data, especially designed to cope with diseased shoulders.

Direct volume rendering, or DVR, is a useful visualisation technique that is often applied as part of medical visualisation solutions. A crucial component of an effective DVR visualisation is a suitable transfer function that assigns optical characteristics to the data. Finding a suitable transfer function is a challenging task. We present two highly interactive methods that facilitate this process.

We also present a method for interactive direct volume rendering on ubiquitous low-end graphics hardware. This method, called ShellSplatting, is optimised for the rendering of bony structures from CT data and supports the hardware-assisted blending of traditional surface rendering and direct volume rendering. This characteristic is useful in surgical simulation

applications.

ShellSplatting is based on the object-order splatting of discrete voxels. As such, maintaining a correct back-to-front or front-to-back ordering during rendering is crucial for correct images. All existing real-time perspective projection visibility orderings show artefacts when splatting discrete voxels. We present a new ordering for perspective projection that remedies these artefacts without a noticeable performance penalty.

This research was performed within the DIPEX (Development of Improved Prostheses for the upper EXtremities) program at the Delft University of Technology, in cooperation with the Leiden University Medical Centre.



---

## Samenvatting

---

Dit proefschrift beschrijft een flexibel software platform voor medische visualisatie en beeldbewerking, een methode voor de segmentatie van het schouder-skelet vanuit CT-data en drie nieuwe technieken op het gebied van direct volume rendering.

Hoofddoel van dit werk was het gebruik van visualisatie technieken te onderzoeken die ondersteuning zouden kunnen bieden bij het operatief vervangen van het schoudergewricht. Hiervoor was een flexibele software omgeving nodig waarmee nieuwe medische visualisatie en beeldbewerkingstechnieken snel ontwikkeld en getest konden worden. We hebben DeV-IDE, of de Delft Visualisation and Image processing Development Environment, gemaakt om deze functie te vervullen. DeV-IDE is een grafische data-flow application builder waarmee visualisatie *en* beeldbewerkingstechnieken gecombineerd kunnen worden. Het ondersteunt het snel creëren en integreren van nieuwe functionele componenten en het maakt het de gebruiker mogelijk om direct en interactief met algoritme implementaties en parameters te experimenteren, een factor waardoor DeV-IDE zich van soortgelijke pakketten onderscheidt.

Voor visualisatie, meting en pre-operatieve planning is een accurate segmentatie van het schouder-skelet vanuit CT data nodig. Omdat de schouder een complex gewricht is en omdat we een methode zochten waarmee door artritis aangetaste schouders gesegmenteerd konden worden, konden we bestaande technieken niet gebruiken. In dit proefschrift beschrijven we een verzameling technieken voor het segmenteren van het schouder skelet uit CT data van schouder gewrichten. Onze technieken kunnen ook omgaan met aangetaste schouders.

Direct volume rendering, of DVR, is een nuttige techniek die vaak toegepast wordt als onderdeel van medische visualisatie oplossingen. Een cruciale component van een effectieve DVR visualisatie is de transfer functie die optische eigenschappen toekent aan een dataset. Het vinden van een geschikte transfer functie is een uitdagende taak. We beschrijven twee interactieve technieken die helpen bij dit probleem.

We beschrijven ook een techniek voor interactieve direct volume rendering op algemeen beschikbare en goedkope graphics kaarten. De techniek, ShellSplating genoemd, is vooral

goed met het renderen van botstructuren vanuit CT data en ondersteunt ook het combineren van traditionele surface rendering en direct volume rendering, iets wat handig is tijdens het simuleren van chirurgische procedures.

ShellSplatting is gebaseerd op het in object-volgorde splatten van discrete voxels. Hierdoor is het snel vinden van een correcte back-to-front of front-to-back volgorde heel belangrijk om correcte renderings te genereren. Alle bestaande visibility algoritmen voor het in real-time perspectief projecteren van discrete voxels veroorzaken zichtbare fouten. We beschrijven een nieuw algoritme dat de fouten grotendeels opheft zonder een meetbare snelheidsverlies.

Het onderzoek dat in dit proefschrift is beschreven was een onderdeel van het DIPEX (Development of Improved Protheses for the upper EXtremities) programma van de Technische Universiteit Delft in samenwerking met het Leids Universitair Medisch Centrum.

---

## Curriculum Vitae

---

Charl Pieter Botha was born on August 25, 1974 in Worcester, South Africa. He matriculated at the Boys' High School in Paarl in December of 1992, and started studying for a Bachelor's degree in electrical and electronic engineering (B.Eng.) at the University of Stellenbosch the year after. This degree was obtained in 1997. After two more years, he was awarded a master of science (M.Sc.) in electronic engineering.

During the last year of his masters, he started working for Crusader Systems in Stellenbosch. After graduation, he continued in Crusader's employ, cooperating on the development of a successful embedded industrial image processing solution for the mining industry. In parallel to this employment and thereafter, he also worked for Stone Three Signal Processing in Somerset-West developing industrial image processing solutions. Here he learnt much by observing how experts go about setting up a successful startup company.

In September of the year 2000, he moved to the Netherlands and started as a Ph.D. student in the Visualisation group of the Delft University of Technology. Since September of 2004, he has been working as post-doctoral researcher in the same group.



---

## Acknowledgements

---

The work described in this thesis was performed in the Data Visualisation group of the Computer Graphics & CAD/CAM section of the Delft University of Technology. It formed part of the Development of Improved endo-Prostheses for the upper EXtremities, or DIPEX, research programme.

During this time, I had the privilege of being surrounded and interacting with a number of fantastic people, some of whom I wish to thank on these pages.

- Frits, my advisor, thank you for the years of Visualisation knowledge and your sage advice.
- Erik, my promotor, for always having things organised quickly and for the fast and accurate feedback on my thesis and other writings.
- All the DIPEX AIOs and PostDocs: Marjolein, Joanne, Dennis, Ed, Andriy, Gerard, Javad, Rogier, Wojciech, Edward. It was a pleasure working together on a worthwhile project.
- Albert Vossepoel, you were one of the major reasons I enjoyed going to DIPEX meetings. Thank you for all the finer insights into image processing, culture and obscure Dutch wordplay.
- Frans van der Helm, your enthusiasm for and practical approach to the role of visualisation and image processing in biomechanical engineering are inspiring.
- Professor Rozing, thank you very much for all your time, effort and insights. I'm looking forward to cooperating with you on building future shoulder replacement technology!

- Ed Chadwick, thank you for stimulating conversation about shoulders, *The Guardian*, PNAC, good Guinness and other matters of international importance.
- Edward Valstar, for strengthening links, showing how to bring engineering to clinical practice and for being Edward.
- Dave Weber, thank you for the example you've set, for the wise words you've given me also during the past few years and for inspiring me to go on this trip in the first place.
- Students I've had the pleasure of working with: LingXiao, Jorik, Joris, Peter Krekel, Victor, Peter Kok and Robin, thank you for the challenging discussions and for keeping me on my toes. LingXiao and Jorik, you're both now working as AIOs in this group: here's to at least another few years of fruitful cooperation!
- All staff members, AIOs and students of the Computer Graphics section: Rafa, Paulos, LingXiao, Gerwin, Benjamin, Eric, Frits, Wouter, Michal, Wim, Erik, Toos, Bart and Ruud, thank you for making this group so much fun to work in.
- Ruud and Bart, thank you very much for keeping all things technical running smoothly. Thank you also for always having the time for a chat.
- Toos, without you, the group would come to a complete standstill.
- To all my friends and family, thank you for doing what friends and families do.
- Paul, I could have made a separate chapter to thank you for all the conversations, "debates", advice, help with all things written, trips to buy hardware, movies I'm too scared to watch, late nights in Delft exploring culinary heights and beery depths, strangely lengthened cycle trips back to Den Haag and so much more, but instead, I'll just thank you for being such a great friend.
- Mom and Barry, thank you for always believing in me. Barry, thank you for making it all happen. You will always live on in my thoughts.
- Stella, thank you for sharing your life with me, you've made mine wonderful.