Technical Report:
DeVIDE - The Delft Visualisation and Image processing Development Environment

Charl P. Botha
TU Delft, The Netherlands

May 30, 2005

Preface

This technical report describes DeVIDE, or the Delft Visualisation and Image processing Development Environment. It is an extract of [1], consisting of chapters 2, 3 and a section of chapter 8 from that document.

Contents

List of Figures

List of Tables

Chapter 1

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.

1.1 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, implementation 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 1.1 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

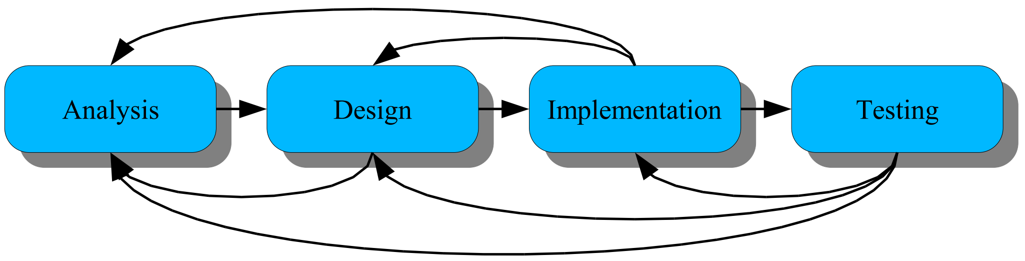


Figure 1.1: The prototyping phase can be seen as a cyclic ADIT trajectory: the problem is analysed and a solution is designed and subsequently implemented. If testing indicates that the solution is suitable, work stops. If not, we return to a previous stage to refine the process with newly-gained information. Such a return to a previous stage for refinement can happen at the design and implementation stages as well.

Environment.

1.2 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.

1.2.1 Visualisation framework models

In [2], 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*.

1.2.2 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 The framework should remain manageable and functional regardless of the amount of functionality (number of modules) that is integrated. 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.

1.2.3 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 1.2 shows an example of a simple network as rendered by the DeVIDE Graph Editor. Figure 1.3 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.

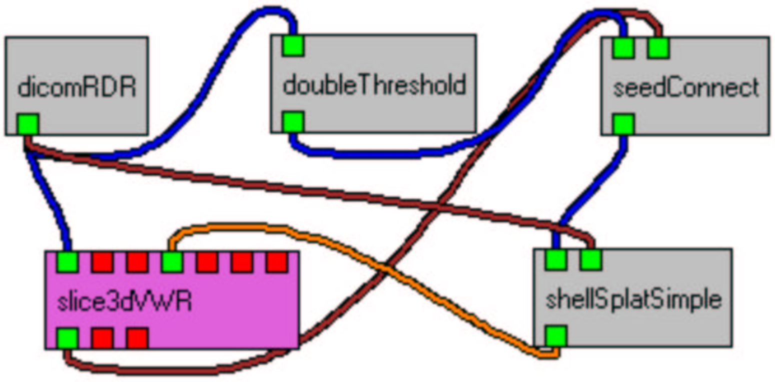
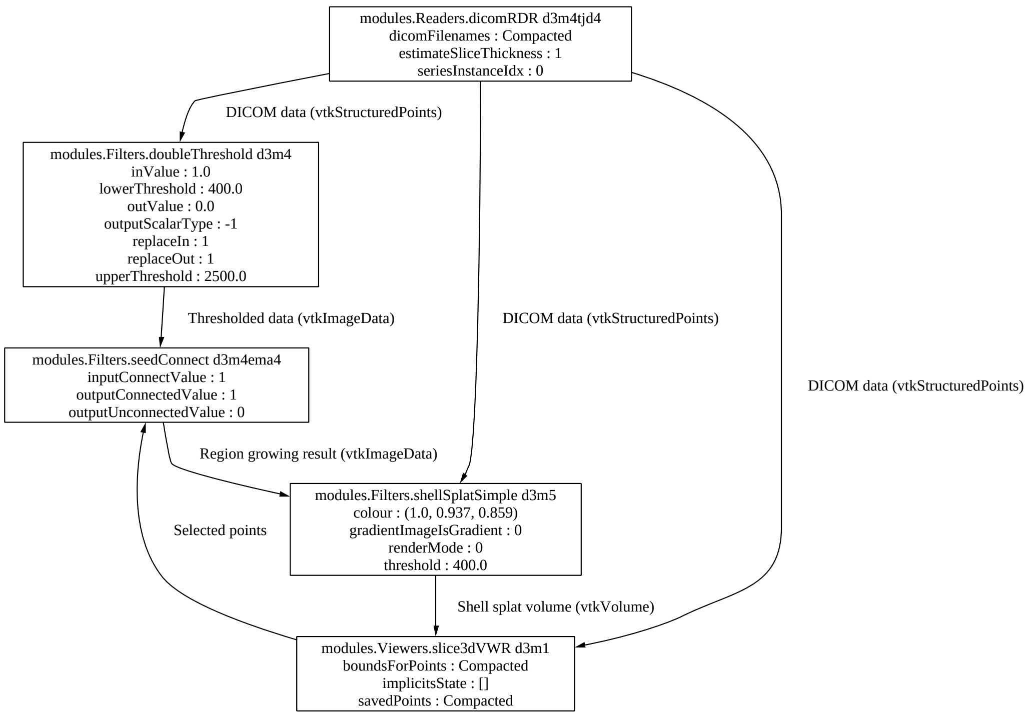


Figure 1.2: Figure showing the DeVIDE visual programming interface to the internal representation of a simple network. DICOM data is being read by the "dicomRDR" module and then thresholded by the "doubleThreshold" module. A 3D region growing is subsequently performed on the thresholded data by the "seedConnect" module, starting from seed points indicated by the user. The result of this region growing is volume rendered with the "shellSplatSimple" module.


```
modules.Readers.dicomRDR d3m4tjd4
dicomFilenames : Compacted
estimateSliceThickness : 1
seriesInstanceIdx : 0

modules.Filters.doubleThreshold d3m4
inValue : 1.0
lowerThreshold : 400.0
outValue : 0.0
outputScalarType : -1
replaceIn : 1
replaceOut : 1
upperThreshold : 2500.0

modules.Filters.seedConnect d3m4ema4
inputConnectValue : 1
outputConnectedValue : 1
outputUnconnectedValue : 0

modules.Filters.shellSplatSimple d3m5
colour : (1.0, 0.937, 0.859)
gradientImageIsGradient : 0
renderMode : 0
threshold : 400.0

modules.Viewers.slice3dVWR d3m1
boundsForPoints : Compacted
implicitsState : []
savedPoints : Compacted
```
DICOM data (vtkStructuredPoints)
Thresholded data (vtkImageData)
DICOM data (vtkStructuredPoints)
DICOM data (vtkStructuredPoints)
Region growing result (vtkImageData)
Selected points
Shell splat volume (vtkVolume)

Figure 1.3: Alternative representation of the simple network shown in figure 1.2. 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.

1.3 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.

1.3.1 VTK and ITK

VTK [3], 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 [4], 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 [5] 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 a 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.

1.3.2 AVS

The Advanced Visualization System, or AVS, is one of the first, if not the first, visualisation-oriented data-flow application builders [6]. 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 [2].

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.

1.3.3 OpenDX

OpenDX, previously known as the IBM Visualization Data Explorer, is a data-flow application builder that focuses on data analysis and visualisation [7]. 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.

1.3.4 SCIRun

SCIRun [8] 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 [9] 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.

1.3.5 VISSION

VISSION [2], 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 [10], 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 1.6.1, 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 [2], VISSION was only available for Unix-like platforms. A porting effort is underway in order to make VISSION available on Windows PCs as well.

1.3.6 ViPEr

ViPEr [11] 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.

1.3.7 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 1.1 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 in Meta-C++, which is interpreted, i.e. not compiled, at run-time. This does simplify the process.

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 1.1: 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.

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 regards 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.

1.4 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 1.5 and 1.6 document respectively architectural and implementation details.

1.4.1 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.

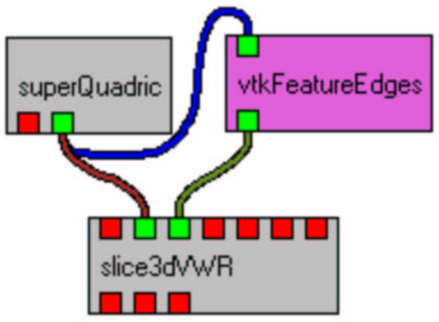


Figure 1.4: DeVIDE network that extracts different types of edges from a super-quadric mesh. The mesh and the extracted edges are visualised together in the *slice3dVWR* module.

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 [5] or Boost.Python[1]. 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 1.5.4 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 [12].

1.4.2 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 1.4 with the *superQuadric*, *slice3dVWR* and *vtkFeatureEdges* DeVIDE modules. The *vtkFeatureEdges* module is a simple VTK object wrapping, explained in section 1.6.6, 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.

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 1.6.6 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 1.6.2.

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 1.5. 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

[1] `http://www.boost.org/libs/python/doc/`

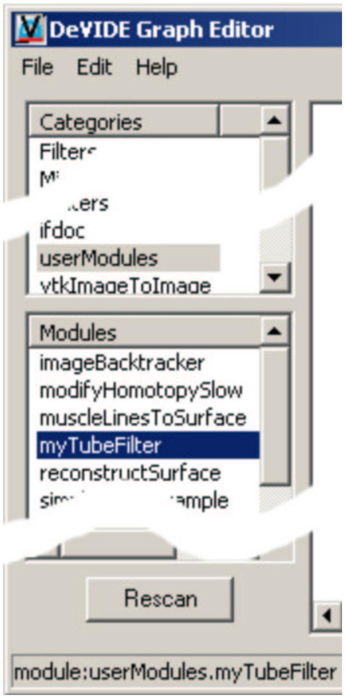


Figure 1.5: The Graph Editor's module palette (shortened) showing the module categories at the top, with the *userModules* category selected. Modules in the currently selected categories are shown in the bottom list. A module can belong to more than one category. Multiple categories can be selected.

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 1.6 shows the resultant network and figure 1.7 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 1.5.5. For example, the user could make use of the introspection facilities to determine better default parameter values for the new module. The visualisation and all other data and parameters throughout the system change immediately to reflect the modifications. Subsequently, the user could mod-

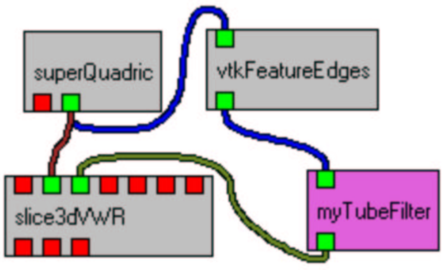


Figure 1.6: The DeVIDE network shown in figure 1.4 with the *myTubeFilter* module integrated.

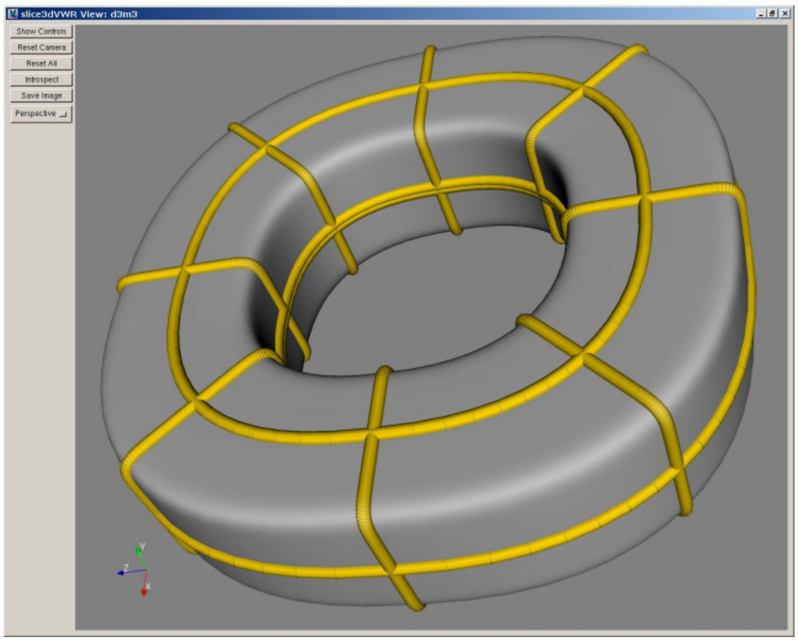


Figure 1.7: A visualisation that makes use of the newly-created *myTubeFilter* module to emphasise discontinuities in the sample mesh.

ify 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.

1.5 Architecture

In 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 [13]. Figure 1.8 illustrates this continuum. In literature, no consistent distinction can be found.

For the sake of this exposition, we will make use of a definition similar to that of Perry and Wolf [14]. 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.

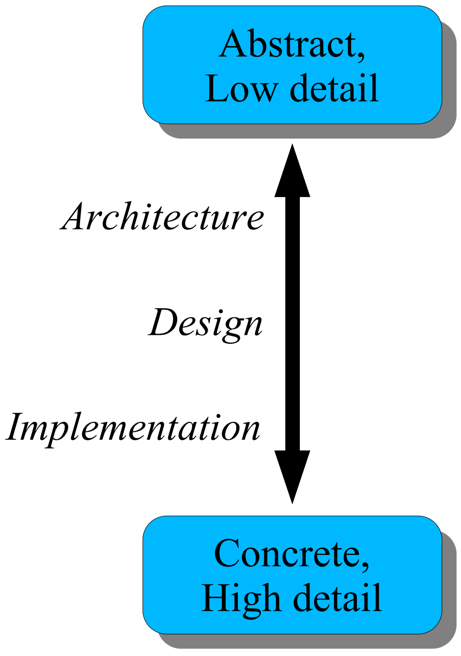


Figure 1.8: The architecture, design and implementation continuum: from concreteness and high-detail to abstraction and low-detail.

We will document design and implementation together in section 1.6. 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 1.9 shows a diagram of the high-level DeVIDE architecture. We will now proceed to document each of the illustrated components.

1.5.1 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 1.6.2 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 1.6.1), 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.

1.5.2 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 1.6.6 has more detail on this.

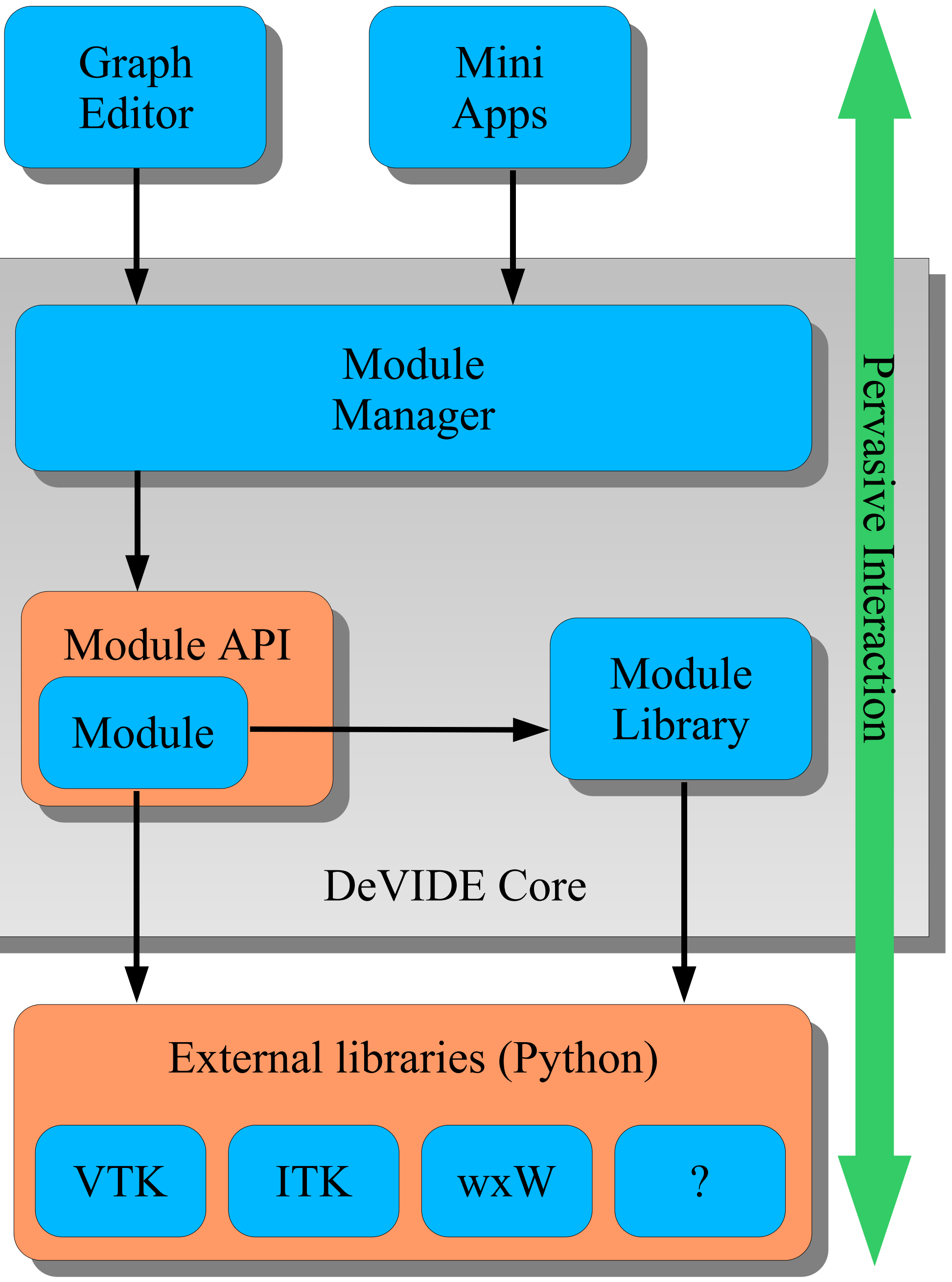


Figure 1.9: Diagram of the high-level DeVIDE architecture. The Module Manager is the main external interface.

1.5.3 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 destroyed, and consumer modules, i.e. modules using data produced by the module that is about to 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 1.6.3.

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 regards 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.

1.5.4 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 [3], the Insight

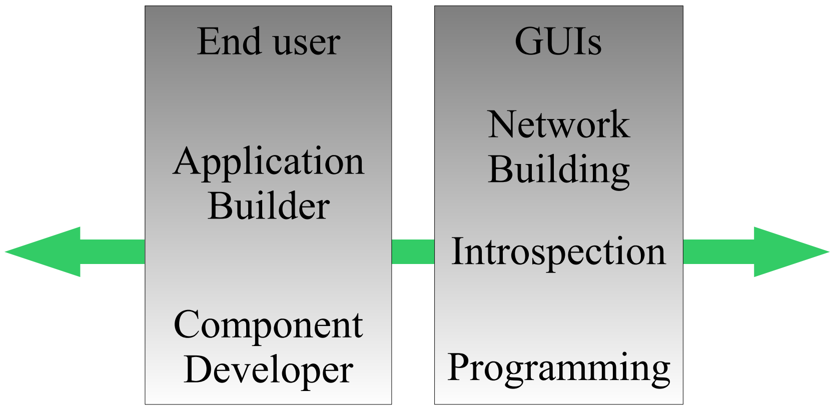


Figure 1.10: The user type and interaction continua. Interaction can take place at any level of abstraction. Each interaction type is more suitable to a specific user type, but no clear partitions can be made. The introspection interaction modality makes interaction at all points possible.

Segmentation and Registration Toolkit, or ITK [4], and wxWidgets/wxPython[2]. 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.

1.5.5 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[3].

In addition to the interaction continuum, one could also imagine a continuum of user types. According to the nomenclature defined by Parsons *et al.* [15], 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 1.10, 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 facilitating the prototyping process. In DeVIDE this interaction breadth is represented by the introspection functionality.

[2]`http://www.wxwidgets.org/` and `http://www.wxpython.org/`.

[3]See section 1.6.4 for more on snippets.

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 1.6.4.

1.5.6 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.

1.5.7 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.

1.6 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.

1.6.1 Python as primary implementation language

Python is an extremely high-level, interpreted, dynamically typed[4] and object-oriented programming language [16]. An important characteristics 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 [17, 18] but, partly because of the well-known rule that 20% of any program is responsible for 80% of the execution time [19], 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.* [15] and by Telea [2]. 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

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

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:

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: Being 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.

1.6.2 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 1.6.6 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 1.1.

Listing 1.1: 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):
```

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 1.1 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 1.1. `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 1.6.5 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[5],
- 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 1.6.6 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 (see section 1.5.2) network state. 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 1.11 illustrates this model of state information flow. The methods that are expected are listed in lines 10 to 13 of listing 1.1 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 1.14 shows an example of such

[5] `http://wxglade.sourceforge.net/`.

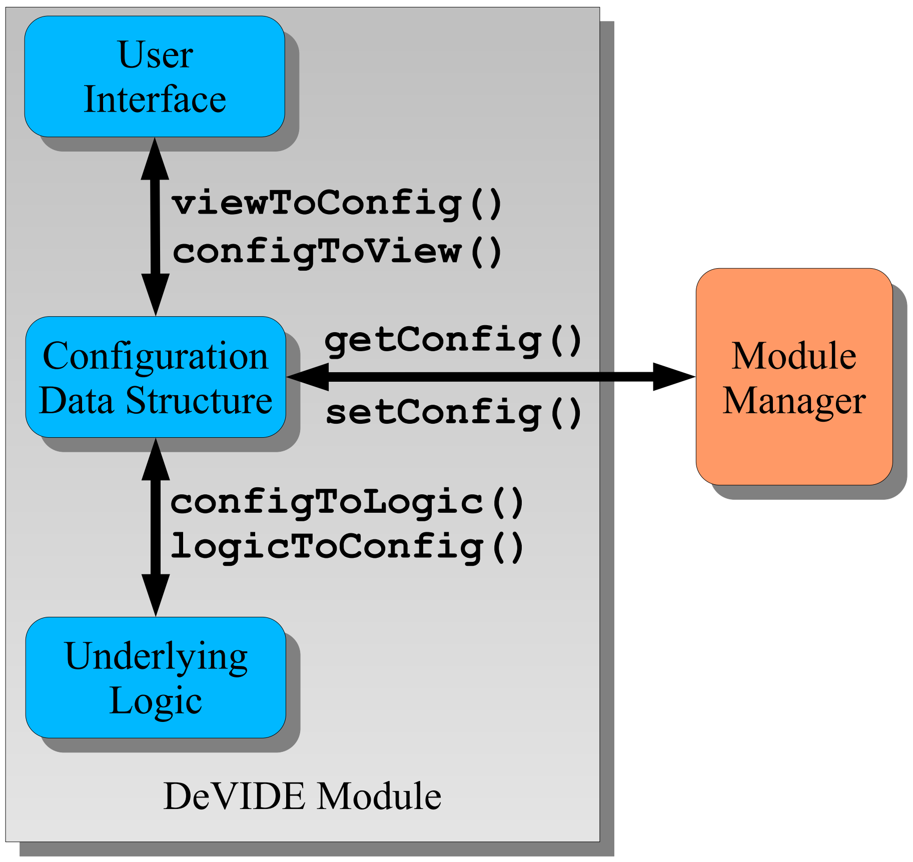


Figure 1.11: A model of the flow of module state information between the user interface, the module configuration data structure and the underlying logic, i.e. the actual implementation of the module algorithm. Communication with the Module Manager is also shown. The module API methods that drive the flow are also indicated.

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 algorithm 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 1.1, 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 1.6.3, 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.

1.6.3 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 1.12.

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 [3] and ITK [4], 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 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

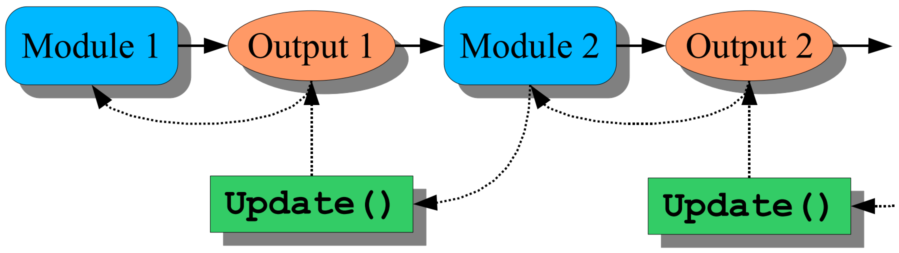


Figure 1.12: The DeVIDE demand-driven execution model. Due to the simple update convention, output data update requests propagate backwards through any given network until the source module is reached. Processing then propagates forwards until it reaches the final output object.

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.

1.6.4 Introspection

As explained in section 1.5.5, 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 1.13 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 1.6.2. 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 visualisations or algorithm outputs. This kind of interaction and real-time feedback make for effective prototyping.

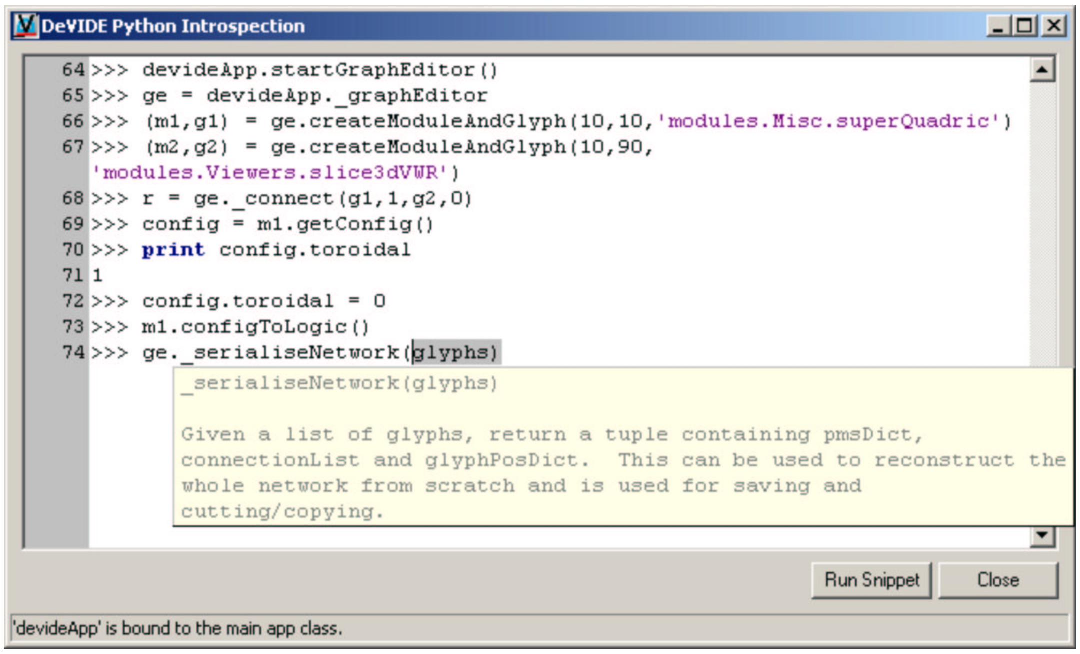
During such a session, the system attempts to assist the dialogue by showing help in the form of tooltips wherever applicable. This help is extracted from the running code. An example of this is shown in figure 1.13. Possible completions of statements are also suggested by querying the member methods and variables of objects. An example is shown in figure 1.15.

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 1.6.6. 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 1.13, 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 a marked 3D object viewer



```
>>> devideApp.startGraphEditor()
>>> ge = devideApp._graphEditor
>>> (m1,g1) = ge.createModuleAndGlyph(10,10,'modules.Misc.superQuadric')
>>> (m2,g2) = ge.createModuleAndGlyph(10,90,
'modules.Viewers.slice3dVWR')
>>> r = ge._connect(g1,1,g2,0)
>>> config = m1.getConfig()
>>> print config.toroidal
1
>>> config.toroidal = 0
>>> m1.configToLogic()
>>> 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.
```

'devideApp' is bound to the main app class.

Figure 1.13: A sample introspection session with the main DeVIDE Python Introspection window.

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.

1.6.5 Data types

As explained in section 1.6.2, 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.

1.6.6 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[6], 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

[6]Most errors are automatically detected by the system and reported to the user.

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 1.6.2, 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 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 1.14 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 1.15 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 1.13, 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[7].

Scripted interface creation

As explained in section 1.6.2, 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 1.14 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.

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 1.6.2. 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 1.2 shows an example of the use of this class to wrap the *vtkTubeFilter* VTK class.

[7] `http://mayavi.sourceforge.net/vtkPipeline/`.

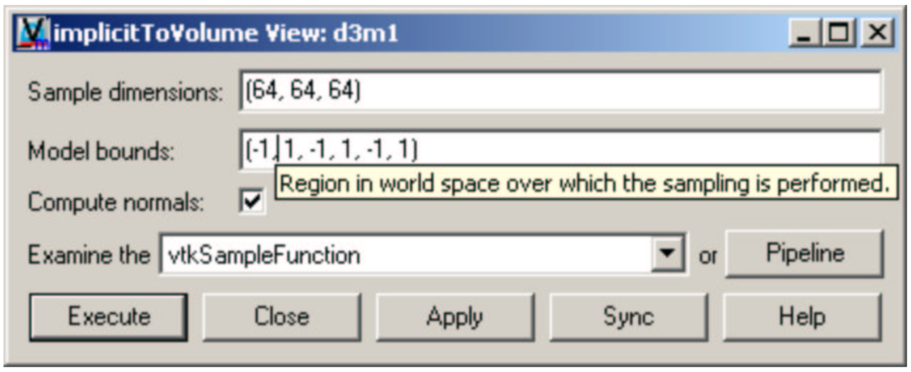
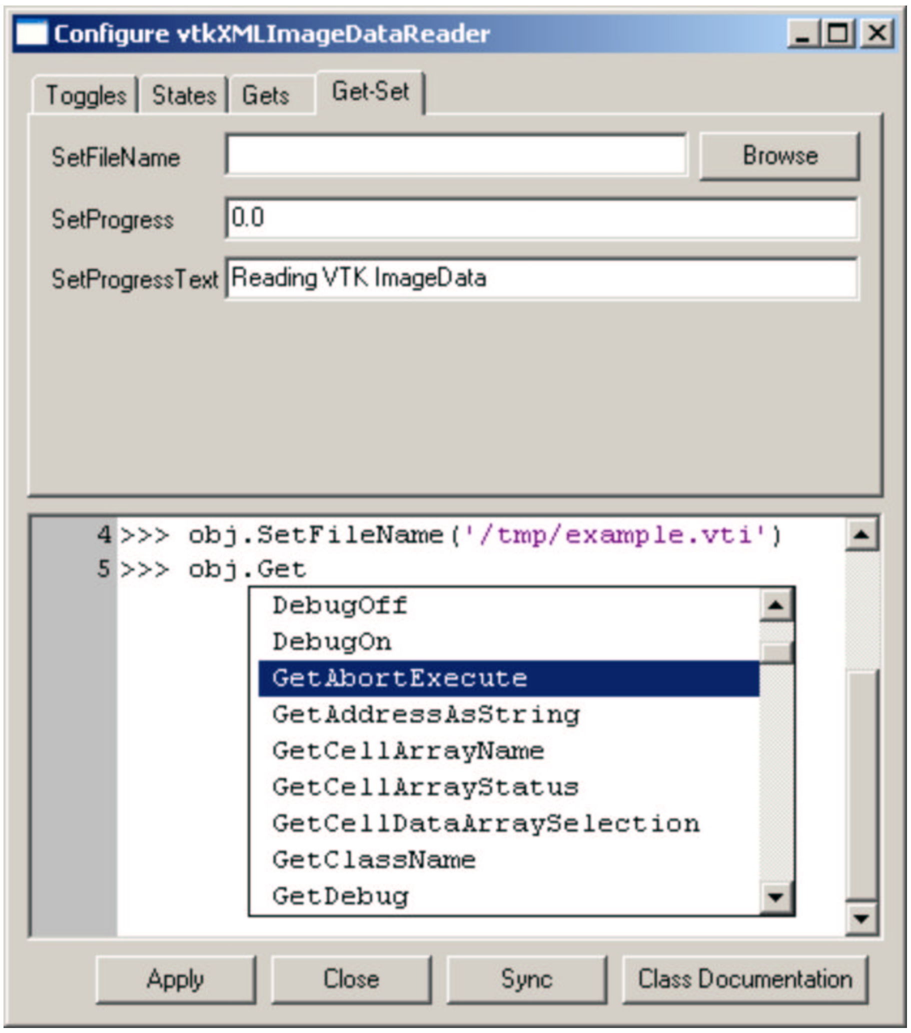

implicitToVolume View: d3m1
Sample dimensions: (64, 64, 64)
Model bounds: (-1, 1, -1, 1, -1, 1)
Region in world space over which the sampling is performed.
Compute normals: ✓
Examine the vtkSampleFunction or Pipeline
Execute
Close
Apply
Sync
Help

Figure 1.14: 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.


Configure vtkXMLImageDataReader
Toggles
States
Gets
Get-Set
SetFileName
Browse
SetProgress 0.0
SetProgressText Reading VTK ImageData

```
>>> obj.SetFileName('/tmp/example.vti')
>>> obj.Get
```

Figure 1.15: 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 1.14.

Listing 1.2: Example of a complete module wrapping of a simple VTK object

```
# class generated by DeVIDE::createDeVIDEModuleFromVTKObject
from moduleMixins import simpleVTKClassModuleBase
import vtk

class vtkTubeFilter(simpleVTKClassModuleBase):
    def __init__(self, moduleManager):
        simpleVTKClassModuleBase.__init__(
            self, moduleManager,
            vtk.vtkTubeFilter(), 'Processing.',
            ('vtkPolyData',), ('vtkPolyData',),
            replaceDoc=True,
            inputFunctions=None, outputFunctions=None)
```

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, it is meant 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.

1.7 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 1.1.

1.8 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 1.5.5, 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 1.6.1.
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 1.3, can also be produced. This plays an important role in the documentation of complete algorithms and methods.

Platform independence All 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 2 presents applications that demonstrate how DeVIDE functions in the algorithm development process. In addition, all the work in chapter 2.4 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 [20], but we plan to investigate other possibilities as well.

Chapter 2

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.

2.1 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[1]. Chronic arthritis can lead to irreversible joint damage and disability.

Joint replacement is indicated in cases where rheumatoid arthritis[2] or osteoarthrosis[3], the 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 [22].

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 [23]. In contrast to this, shoulder replacement yields good results with regard to pain relief, but fares significantly worse with regards 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 [24], 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 2.1 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

[1]See the Reuma Stichting website at `http://www.reuma-stichting.nl/`.

[2]**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.

[3]**Osteoarthrosis**, 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 [21]. Osteoarthrosis 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 osteoarthrosis. 90% of adults over the age of 65 suffer from some form of osteoarthrosis (source: `http://www.reuma-stichting.nl/`).

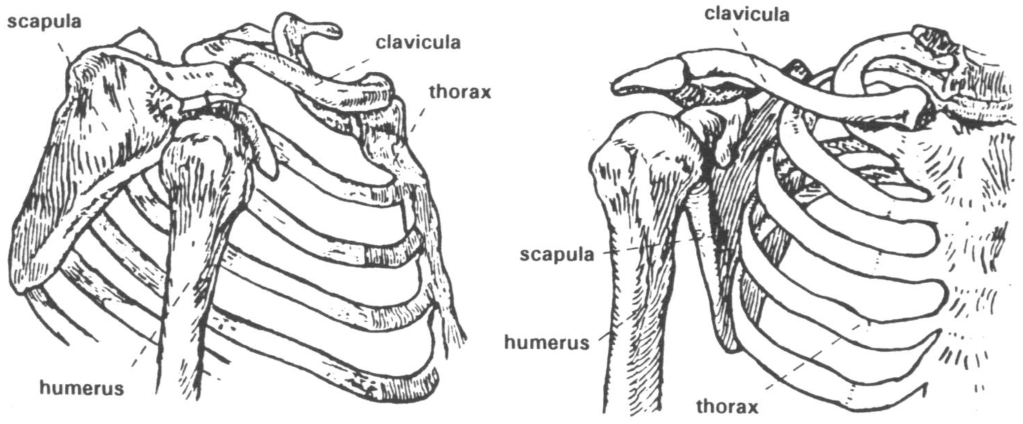


Figure 2.1: The skeletal structures of the shoulder. (Courtesy of the Delft Shoulder Group.)

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 clavicula, 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 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 2.2 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 [25]. As an important facet of this effort, a novel approach to the surgical planning of shoulder replacement procedures is being developed [26].

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 per-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 per-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

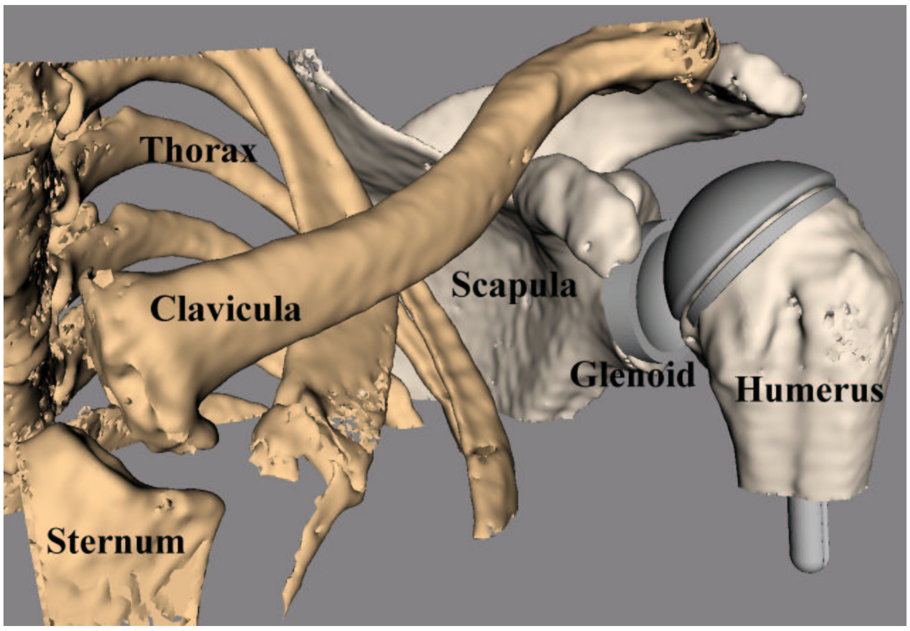


Figure 2.2: CT-derived visualisation of the bony-structures in the human shoulder. Both the humeral head and glenoid prostheses have been virtually implanted, i.e. a total shoulder replacement.

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.

2.1.1 Segmentation

A pre-operative CT-scan of the affected shoulder is made. The resultant DICOM [27] 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 osteo-arthrosis (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 2.4 for more details on this.

After the segmentation, a surface model of the scapula is generated with the Marching Cubes algorithm [28]. 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 [29, 30]. 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 2.1.3 we discuss some of the complications of this fact.

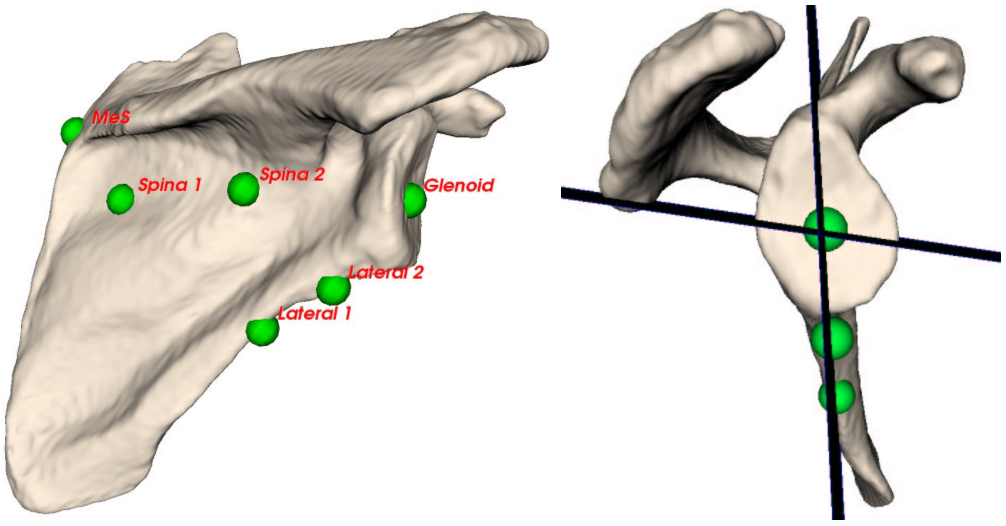


Figure 2.3: On the left, a scapula is shown with the points that are required to place the two anatomical planes. On the right, the same scapula is shown with the resultant anatomical planes intersecting to form an insertion axis for the glenoid prosthesis.

2.1.2 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[4]. 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 [31].

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 2.3 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 2.3 and 2.4 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 2.3, 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 2.3. 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 2.3.

The constraint system can now automatically align the prosthesis axis with the intersection of the two

[4]Source: Prof. P.M. Rozing, head of the orthopaedics department at the Leiden University Medical Hospital, The Netherlands.

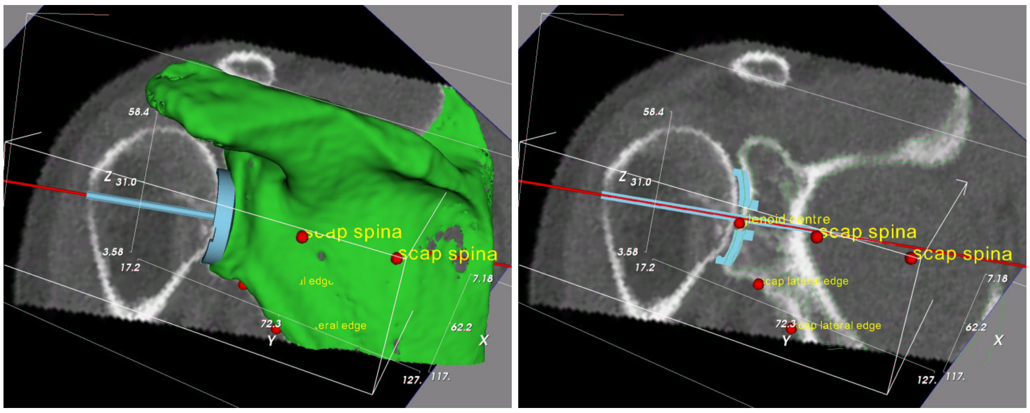


Figure 2.4: Part of the glenoid component planning functionality. On the left the complete models are shown and on the right only their contours. The contour view enables the user to judge whether the prosthesis has a good fit and is reminiscent of the conventional template-on-x-ray planning. The slice is anatomically oriented but can be moved to check all parts of the glenoid component.

planes and optionally constrain all prosthesis motion to the plane intersection. The surface model with the horizontal pin on the left of figure 2.4 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 2.4.

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 2.4. 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.

2.1.3 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 [32] to the design of a patient-specific glenoid drill guide. Figure 2.5 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 [26]. 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

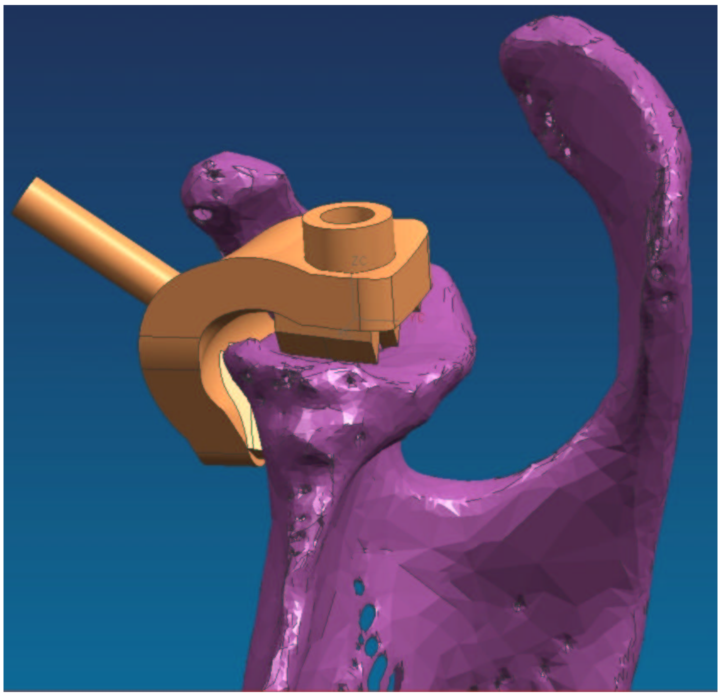


Figure 2.5: The latest design of the glenoid drill guide visualised on a surface model of a scapula. Illustration courtesy of Liesbet Goossens and Bart de Schouwer, Katholieke Universiteit Leuven, Belgium.

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:

- 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 2.4 we present an approach that solves the problem of segmenting affected bone, but the cartilage segmentation problem is still open.

- 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 2.6, 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 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^o 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.

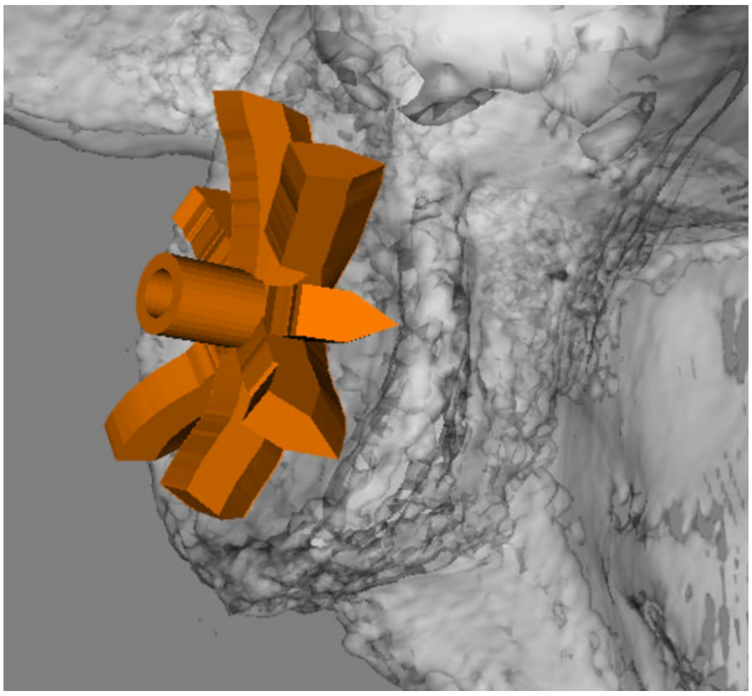


Figure 2.6: An earlier mould design that is in the process of being constructed by DeVIDE according to a set procedure, the patient-specific scapular surface and the final pre-operative planning parameters.

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.

2.1.4 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 2.4 has more details on our planned future work.

2.2 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.

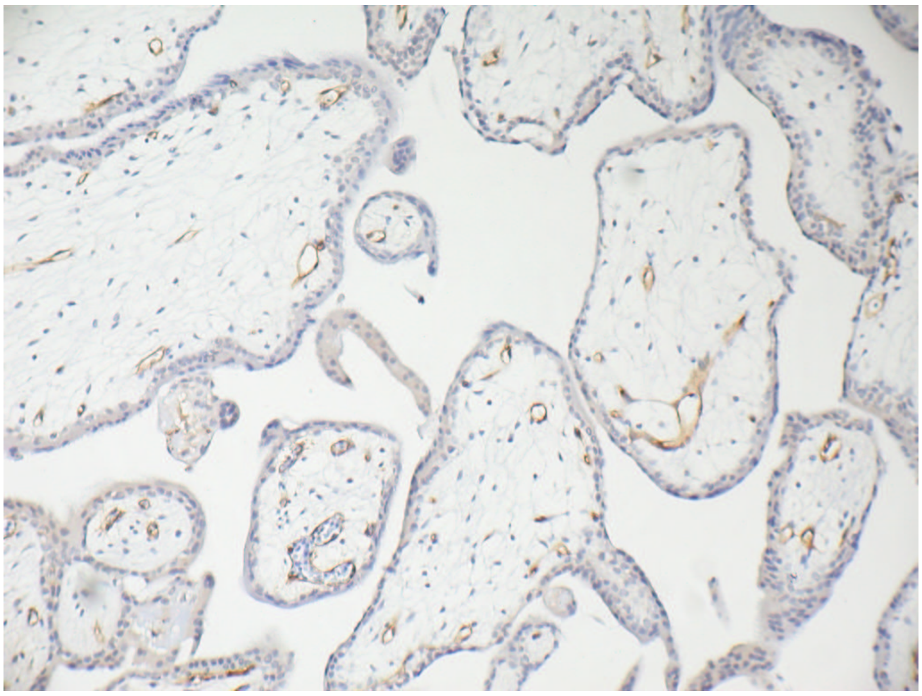


Figure 2.7: A digitised serial section of the chorionic tissue. The large structure covering almost half of the left image border is a villus. The circular structures in its interior are vessels.

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 [33].

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.

2.2.1 Method

In a previous study [34], 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[5]. These microscopic sections were digitised. Figure 2.7 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 2.8 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.

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,

[5]Sections were stained with haematoxylin-eosin, CD31 immunohistochemistry and periodic acid-Schiff.

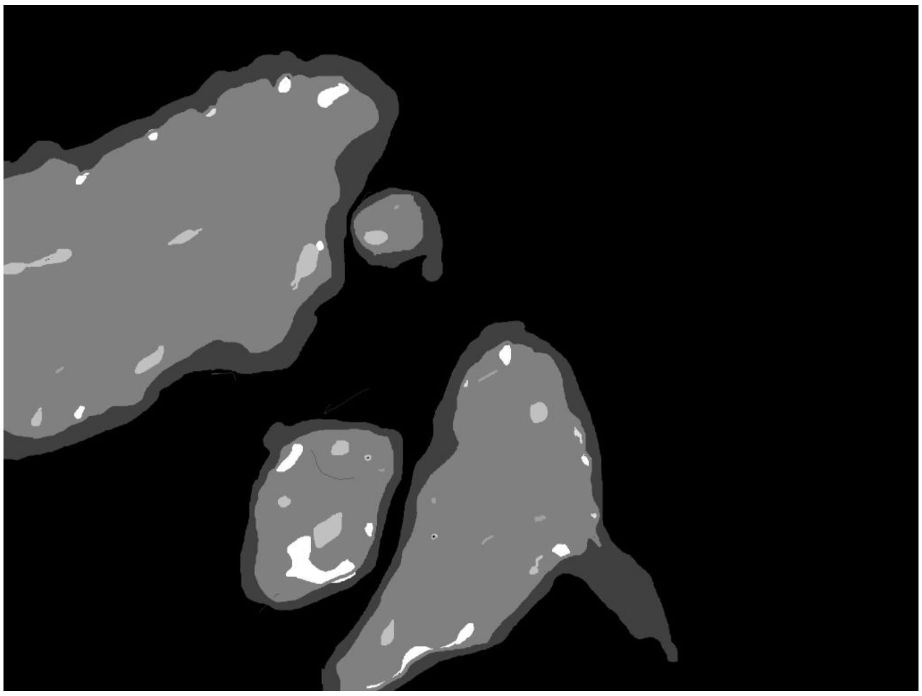


Figure 2.8: The section shown in figure 2.7 after manual delineation and filling with grey levels.

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 [4] calls, were used in this process. Figure 2.9 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 2.10 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 2.9. This module also combines VTK and ITK functionality. The resultant 3-D dataset can be processed and visualised with conventional 3-D techniques.

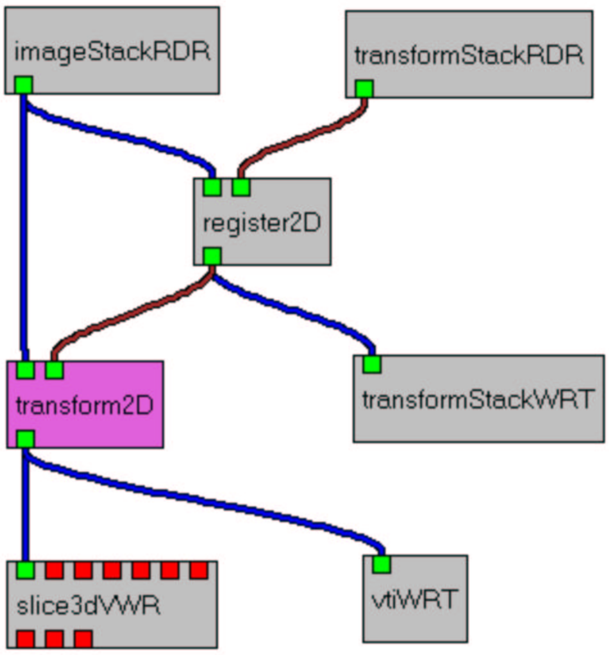
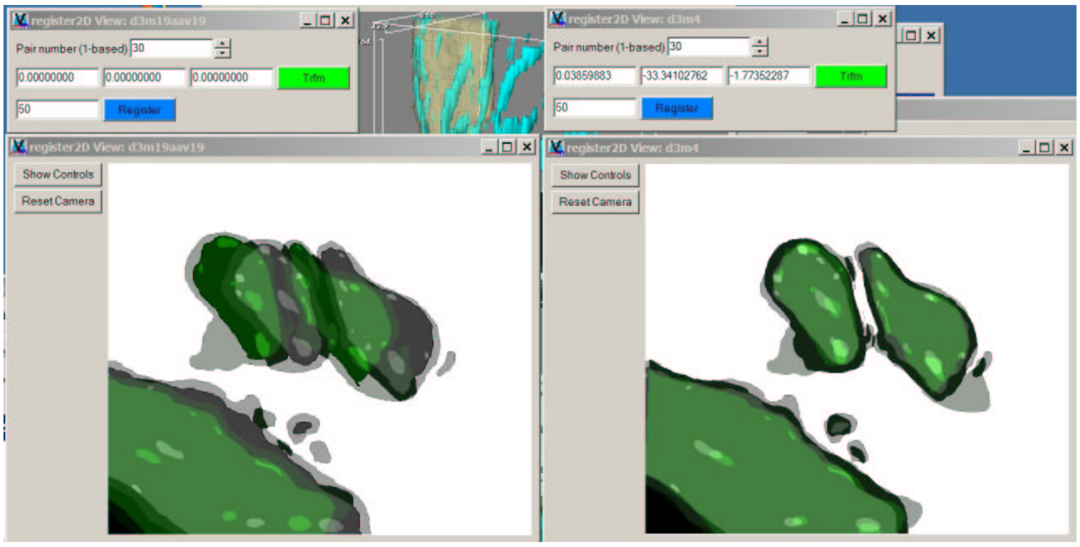


Figure 2.9: The DeVIDE modules and network that were used to perform the registration. *imageStackRDR* and *transformStackRDR* read collections of input images and image pair transforms respectively. *register2D* encapsulates the registration user interface. *transform2D* applies all derived transformations 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.


register2D View: d3m19aav19
Pair number (1-based) 30
0.00000000 0.00000000 0.00000000
50

register2D View: d3m4
Pair number (1-based) 30
0.03859883 -33.34102762 -1.77352287
50

Figure 2.10: 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.

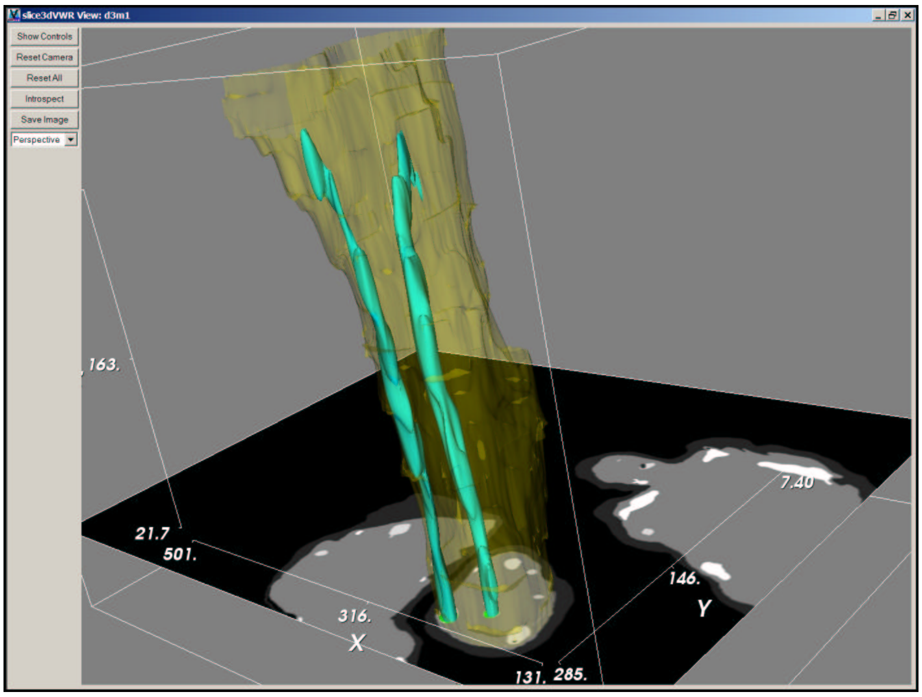


Figure 2.11: An example of a visualisation of the reconstructed slices. A single villus and two of its vessels are visualised.

2.2.2 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 2.11 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 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 $4\mu m$ 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.

2.3 Pelvic floor displacement from MRI

DeVIDE was used in two studies on the female pelvic floor [35, 36] 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.

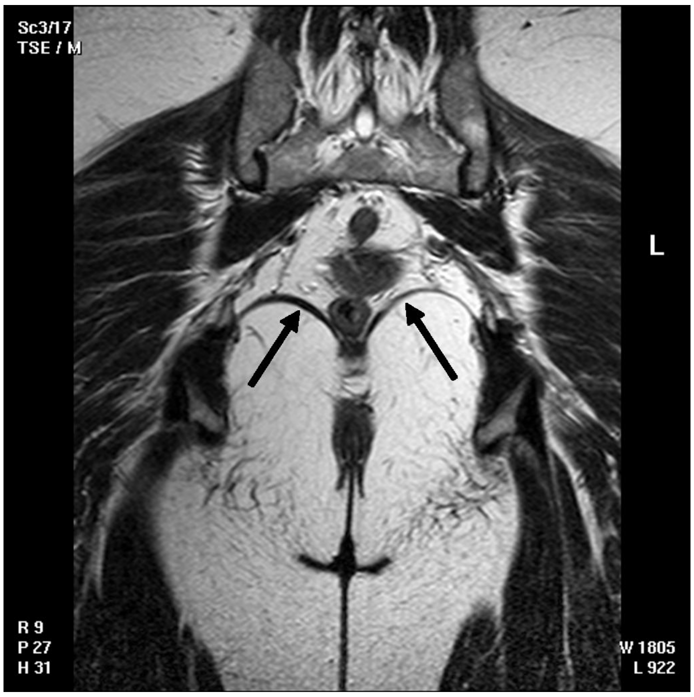


Figure 2.12: An example coronal slice from one of the MRI datasets. The arrows indicate the pelvic floor muscles. The surface shown in 2.14 describes the inner surface of these muscles in 3D.

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.

2.3.1 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 2.12 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, a 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 straight-forward. 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 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

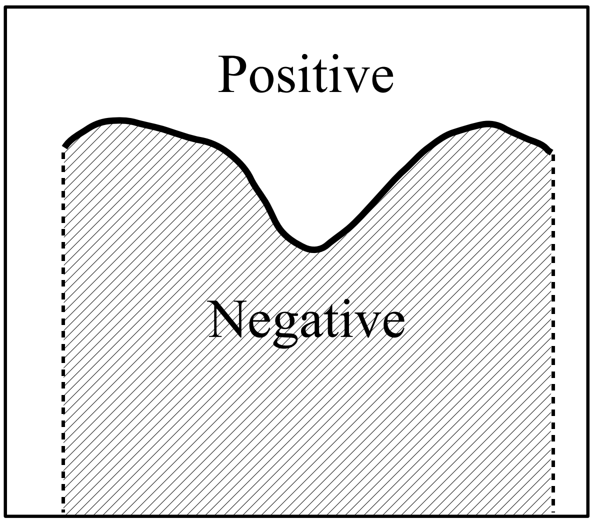


Figure 2.13: The signed distance field generated from the delineated curve. Above and to the sides the distance values are positive (i.e. the non-hatched areas)

arbitrarily chosen origin, as long as this is consistent for all images in a dataset. Figure 2.13 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 2.13 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 2.13. 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 2.14. 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.

2.3.2 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 [37] between the different pelvic floor surfaces derived for each patient. First the surfaces were divided into two anatomical parts. In figure 2.14, 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 [37]. Mesh was used to calculate the Hausdorff distances.

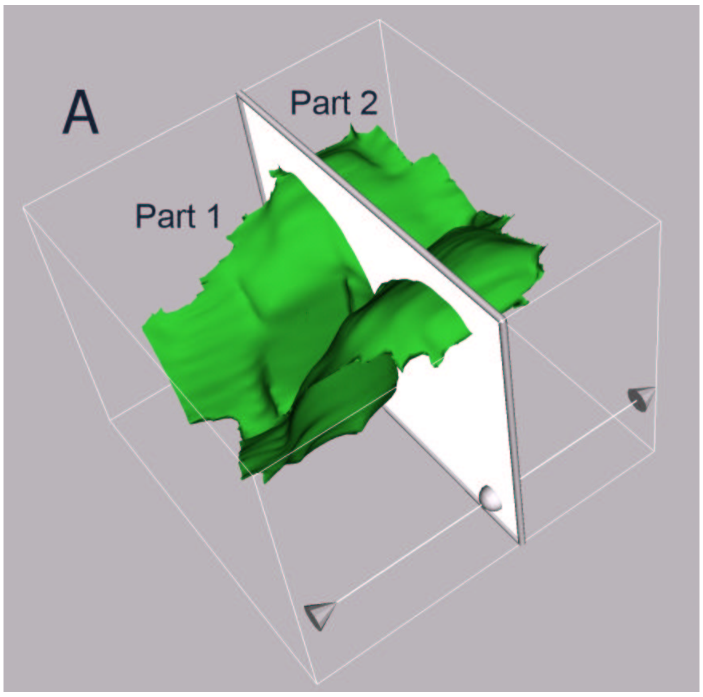


Figure 2.14: Example surface describing inner border of pelvic floor. Image courtesy of Stepan Janda and created with DeVIDE.

2.3.3 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 [35] and [36].

2.4 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 2.1.4 and 2.4 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 2.9, 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.

Further Reading

The segmentation techniques documented in this technical report are quite basic: Chapter 4 of [1] details a more advanced approach that also deals with shoulder data of patients that have been affected by bone-changing diseases.

Chapter 8 of [1] contains more information on planned future work.

Bibliography

[1] C. P. Botha, *Techniques and Software Architectures for Medical Visualisation and Image Processing*. PhD thesis, Delft University of Technology, 2005. To appear.

[2] A. C. Telea, *Visualisation and Simulation with Object-Oriented Networks*. PhD thesis, Technische Universiteit Eindhoven, 2000.

[3] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*. Prentice Hall PTR, 2nd ed., 1999.

[4] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*. Kitware Inc., 2003.

[5] D. M. Beazley, "Automated scientific software scripting with swig," *Future Gener. Comput. Syst.*, vol. 19, no. 5, pp. 599–609, 2003.

[6] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schleigel, J. Vroom, R. Gurwitz, and A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, pp. 30–42, July 1989.

[7] G. Abram and L. Treinish, "An extended data-flow architecture for data analysis and visualization," in *Proceedings of the 6th conference on Visualization '95*, p. 263, IEEE Computer Society, 1995.

[8] S. G. Parker, *The SCIRun problem solving environment and computational steering software system*. PhD thesis, The University of Utah, 1999.

[9] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Addison-Wesley Professional, 1994.

[10] F. Rademakers and R. Brun, "Root: An object-oriented data analysis framework," *Linux Journal*, no. 51, 1998.

[11] M. F. Sanner, D. Stoffler, and A. J. Olson, "Viper, a visual programming environment for python," in *Proceedings of the 10th International Python Conference*, pp. 103–115, February 2002.

[12] S. Mauch, *Efficient Algorithms for Solving Static Hamilton-Jacobi Equations*. PhD thesis, California Institute of Technology, 2003.

[13] A. H. Eden and R. Kazman, "Architecture, design, implementation," in *Proceedings of the 25th international conference on Software engineering*, pp. 149–159, IEEE Computer Society, 2003.

[14] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[15] D. Parsons, A. Rashid, A. Speck, and A. Telea, "A "framework" for object oriented frameworks design," in *Proceedings of Technology of Object-Oriented Languages and Systems, 1999.*, pp. 141–151, 1999.

[16] G. van Rossum, *Python Reference Manual*. Python Software Foundation, April, 2001.

[17] J. K. Ousterhout, "Scripting: higher level programming for the 21st century," *IEEE Computer*, vol. 31, pp. 23–30, March 1998.

[18] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[19] B. W. Boehm, "Improving software productivity," *Computer*, vol. 20, no. 9, pp. 43–57, 1987.

[20] A. Telea and J. J. van Wijk, "SMARTLINK: An agent for supporting dataflow application construction," in *Proceedings of Joint Eurographics, IEEE and TCVG Symposium on Visualization 2000*, pp. 189–198, 2000.

[21] J. H. Klippel and P. A. Dieppe, eds., *Rheumatology*. Mosby, 1994.

[22] M. Wirth and C. Rockwood, "Complications of shoulder arthroplasty," *Clinical Orthopaedics*, no. 307, pp. 47–69, 1994.

[23] 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.

[24] M. Torchia, R. Cofield, and C. Settergren, "Total shoulder arthroplasty with the neer prosthesis: long-term results.," *Journal of Shoulder and Elbow Surgery*, vol. 6, no. 6, pp. 495–505, 1997.

[25] E. R. Valstar, C. P. Botha, M. van der Glas, P. M. Rozing, F. C. van der Helm, F. H. Post, and A. M. Vossepoel, "Towards computer-assisted surgery in shoulder joint replacement," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 56, pp. 326–337, August 2002.

[26] E. Valstar, K. van Brussel, B. Kaptein, B. Stoel, and P. 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.

[27] National Electrical Manufacturers' Association, "NEMA Standard PS3: Digital Imaging and Communications in Medicine (DICOM)," 2000.

[28] W. Lorensen and H. Cline, "Marching cubes: a high resolution 3d surface construction algorithm," in *Proc. of ACM SIGGRAPH*, pp. 163–169, Association for Computing Machinery, 1987.

[29] G. Taubin, "Optimal surface smoothing as filter design," Tech. Rep. RC-20404(#90237), IBM T.J. Watson Research Center, 1996. `http://http://www.research.ibm.com/people/t/taubin`.

[30] G. Taubin, "Geometric signal processing on polygonal meshes," in *STAR – State of the Art Report, Eurographics 2000*, 2000.

[31] O. D. Leest, P. Rozing, L. Rozendaal, and F. V. der Helm, "The influence of glenohumeral prosthesis geometry and placement on shoulder muscle forces," *Clinical Orthopedics*, vol. 330, pp. 222–233, 1996.

[32] J. Goffin, K. V. Brussel, K. Martens, J. V. Sloten, R. V. Audekercke, and M. Smet, "Three-dimensional computed tomography-based, personalized drill guide for posterior cervical stabilization at c1–c2," *Spine*, vol. 26, no. 12, pp. 1343–1347, 2001.

[33] B. A. Lisman and N. Exalto, "Early human nutrition and chorionic villous vascularization," *Middle East Fertility Society Journal*, vol. 4, no. 2, 1999.

[34] C. Vis, E. Everhardt, J. te Velde, and N. Exalto, "Microscopic investigation of villi from chorionic villous sampling," *Human Reproduction*, vol. 13, no. 10, pp. 2954–2957, 1998.

[35] 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.

[36] 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 in supine position,” *Submitted to: American Journal of Obstetrics and Gynecology*, 2004.

[37] 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)*, pp. 705–708, 2002.