An Introduction to Programming for Medical Image Analysis with The Visualization Toolkit

Xenophon Papademetris

Draft December 12 2006

A Programming guide for the BioImage Suite Project

This book is an edited collection of class handouts that was written for the graduate seminar "Programming for Medical Image Analysis" (ENAS 920a). This class was taught at Yale University, Department of Biomedical Engineering, in the Fall of 2006. Some the comments in this draft version of the book reflect this fact. For example, see comments beginning "at Yale". Also the word "session" is often used interchangeably with the work "chapter", this is also a leftover from the class handouts. Eventually, such comments will be corrected or removed from the text. Furthermore, many of the references that will appear in the final version are still omitted. It is made available at this stage in the hope that it will be useful.

Contents

Preface 5

I Introduction 7

1. Introduction 8
2. Revision Control With Subversion 13

II Programming with Tcl/Tk 18

3. Introduction to Tcl 19
4. Advanced Topics in Tcl 27
5. An Introduction To Tk 35
6. Tk Part II 43
7. Object Oriented Programming with [Incr] Tcl 53
8. Iwidgets: Object Oriented GUIs 63

III The Visualization Toolkit I – Using Tcl 70

9. An Introduction to the Visualization Toolkit 71
10. Curves and Surfaces in VTK 76
11. Images in VTK 86
12. Displaying Images in VTK 93
13. Transformations 105
14. Some Additional VTK Classes 114

IV Interfacing To BioImage Suite using Tcl 121

15. Leveraging BioImage Suite Components 122

16. Writing your own BioImage Suite Application 132

V C++ Techniques 146

17. Cross-Platform Compiling with CMAKE 147

18. C++ Techniques and VTK 154

VI VTK Programming with C++ and Tcl 166

19. Extending VTK using C++ 167

20. Point-based Registration with ICP 179

21. Intensity Based Segmentation 197

22. A Templated Image to Image Filter 215

23. Copying Data Objects 222

24. The Insight Toolkit 226

VII Appendices 234

A. Final Exam 235

B. Code License 236

References 237

Preface

This book is an edited collection of class handouts that I wrote for the graduate seminar "Programming for Medical Image Analysis" (ENAS 920a) that was taught at Yale University, Department of Biomedical Engineering, in the Fall of 2006. My goal for the class was to provide sufficient introductory material for a typical 1st year engineering graduate student with some background in programming in C and C++ to acquire the skills to leverage modern open source toolkits in medical image analysis and visualization such as the Visualization Toolkit (VTK)[24] and, to a lesser extent, the Insight Toolkit (ITK)[12].

One obviously has to acknowledge that there are many programming books out there that cover the material discussed in this book in more depth (and more correctly perhaps); I list many of my sources in Chapter 1. However, placed one on top of the other, these books form a pile about 1-2 feet high, which can be discouraging to a beginner. This book is meant to be an introductory guide. Many of the definitions are informal, much like one teaches math at 1st grade. As the student progresses and begins to understand the introductory material, then he/she should look at more specialized books which offer a deeper insight into what is going on. The manual pages for many of these toolkits also become useful at this point.

Most of our graduate students – the intended audience for this book and class – while having a strong applied mathematics/signal processing background, are not expert programmers. Frequently, they would have had some programming classes at the undergraduate level and would have been, most likely, exposed to C/C++ at some point. However, with rare exceptions, a dive into the combination of object-oriented and generic programming model used in ITK, for instance, would leave most of them befuddled.

Such students begin their graduate research in semester long projects called "special investigations". This is part of the process of identifying a topic for their research as well as a lab in which they will pursue their dissertation work. In our own research in medical image analysis, the typical product of a doctoral dissertation is a mathematical framework for attacking an image analysis problem which has to be translated into computer code for testing and validation.

Most of the students, in these special investigations, prototype ideas in MATLAB [15]. While MATLAB is a wonderful prototyping tool, it leaves much to be desired in terms of the development of the programming habits needed to write a large, sustainable, and reusable body of code. Unfortunately, many students ending up in the trap of developing the best algorithms that can be implemented in MATLAB as opposed to focusing on what on optimal algorithmic strategy would be. This is especially apparent once large 3D and 4D datasets enter into the picture, and their algorithms end up taking hours and days to run.

At this point in the game, a helpful professor suggests that they should probably look to move to a more efficient language such as C++. However, one look at straight C++ without any of the additional toolkits, makes them realize that switching to C++ is easier said than done. There are very few default

operations for things like linear algebra, image processing, image display etc. Then, perhaps, another helpful person suggests that they take a look at VTK and/or ITK. While now, they can see that there is a ton of functionality out there, they are often lost as to where to begin. VTK and ITK are natural tools once one is used to them but they can be imposing and "scary" to the beginner. While there are some books out there (especially the VTK User's Guide) which are very helpful, they are often only obliquely related to what they really need to learn how to do: implement image analysis methods, learn how to (properly) display their results, and learn how to put a graphical user interface to enable them and their potential users to interact with the methods. The goal of the course, and this book, is to precisely provide the necessary guidance for a new graduate student in order to achieve these goals.

The selection of the material, as well as its presentation, is naturally colored by the author's own experience. I coordinate the BioImage Suite project `www.bioimagesuite.org`, which is a large medical image analysis utility developed in a mixture of C++ and [Incr] Tcl. [1] In part, the motivation for teaching this class and writing this book, is directly derived from the needs of the BioImage Suite project. In particular, the driving question was, how does one get a new programmer up to speed with the skills he/she will need in order to contribute? Perhaps somebody else teaching this class would have used Python instead of Tcl as the scripting language; this is as much a matter of taste (and endless discussions and sadly flame wars in this internet era) and experience with the particular language as anything else. Also, while I mention ITK towards the end – see Chapter 24, the primary toolkit used here is VTK. The template-free interface of VTK makes it easier for beginners, and I find that ITK can be downright user-hostile to the less experienced programmer.

One of my complaints with many introductory texts is that they never attempt to teach how one goes about learning how to put together decent sized applications. They focus too much on "grammar" and too little on "writing stories". However, graduate students need to write "stories" – useful ease-to-use tools that both they and clinical and/or basic science collaborators can use and maintain. Hence, as part of the class, I have also made an attempt to introduce some software engineering tools such as Subversion and CMAKE. As any experienced developer often learns the hard way, these tools can be just as critical as the choice of programming language or toolkit.

Finally, a disclaimer: While VTK is now at version 5.0 we still focus on VTK 4.4 – this is the version in use in BioImage Suite at this point. As we migrate the software to VTK 5.x (a good rule of thumb is never to use .0 release of any toolkit) this book will be updated to reflect this.

Acknowledgments: BioImage Suite is supported by the National Institutes of Health (NIH)/National Institute of Biomedical Imaging and Bioengineering (NIBIB) under grant 1 R01 EB006494-01. The author would also like to acknowledge the valuable help of Thomas Teisseyre in preparing this book.

[1] Interfacing with BioImage Suite is discussed in Chapters 15 and 16.

Part I

Introduction

Chapter 1

Introduction

1.1 Overview

This book consists of six main parts:

I. Introduction: This presents some introductory material including a brief overview of the Subversion revision control system. Subversion was used extensively for the class as a means to upload homework assignments and to download update notes and pdf files.

II. Programming with Tcl/Tk Here, we first introduce the Tcl [10] scripting language and the Tk graphical user interface toolkit. The [Incr] Tcl [13, 27] object-oriented extension of Tcl is then used to introduce the concepts of object-oriented programming (OOP). While OOP could have been introduced in the context of C++, it is first deliberately discussed in a scripting language concept, deliberately, in order lower the learning curve and avoid additional complications such as compiling and linking. The final chapter of this part describes the Iwidgets object-oriented graphical user interface toolkit.

III. The Visualization Toolkit I – Using Tcl: In this part we present a guided tour of those aspects of VTK that are most relevant to medical image analysis using the Tcl language.

IV. Interfacing To BioImage Suite using Tcl Two chapters, chapters 15 and 16, are devoted to explicitly interfacing with the BioImage Suite[20, 19] image analysis package. BioImage Suite provides a large number of additional components, such as complex 3D viewers, that can simplify the task of developing medical image analysis applications.

V. C++ Techniques In this short part of the book, we describe first the CMake program for managing the building (i.e. compiling and linking) of software on multiple platforms.

Next we revisit the concepts of object-oriented programming and translate the original [Incr] Tcl code to C++. Finally we also translate some of the VTK Tcl examples to C++ to demonstrate how to access VTK from C++

VI. VTK Programming with C++ and Tcl This final part of the book is meant to guide students towards implementing their own algorithms in C++ and VTK, while using Tcl/[Incr Tcl] for graphical user interfaces.

Two large case studies, one on point based registration and one on intensity based segmentation form the heart of this part. Here we present complete examples of both algorithm implementation, user interface design and 3D viewer integration. The first case study uses vanilla VTK whereas the second makes use of BioImage Suite concepts explicitly.

The part concludes with three additional chapters. The first describes the implementation of templated image-to-image filters. The second, which was a response to common mistakes in the homeworks, describes how to properly copy data objects so as to save the results of a pipeline for later use. The final chapter briefly touches on the ITK toolkit and demonstrates how to use in conjunction with VTK.

1.2 Software Needed for this Class

The following software is needed to create a proper working environment for this class:

1.2.1 Image Analysis Specific Tools

While one can get the various tools needed individually (Tcl/Tk, ITcl, Iwidgets, VTK, ITK etc.) my suggestion is to simply download and install the Yale BioImage Suite software package `www.bioimagesuite.org`, which includes all of the above properly compiled to work together. It will save you days (if not weeks) of work. In any event, if you can compile and install all of these by yourself, you are most definitely over-qualified for this class (book).

On Linux make sure you install the version that matches the compiler installed on your system.

1.2.2 General Tools

A good text editor: The choice of text editor often has quasi-religious connotations. In the "old days" the two major editors where `emacs` and `vi`, each of which were almost equally powerful and user-hostile. My own preference has always been to use `emacs` were possible and one is willing to put up with the steep learning curve for this, it is highly recommended. Emacs will run on just about anything, a more user friendly version called XEmacs might be more suitable for beginners. (Extra credit assignment: Read about the messy Emacs vs XEmacs "flame war" that led to the split of the XEmacs project from core Emacs).

On unix-like operating systems (which for desktop use means Linux these days), a good alternative is nedit. This is installed on most Linux machines, I can help you install it if needed. Nedit is also available for windows but requires the installation of the cygwin environment which is not a trivial exercise.

For Windows, I have heard good words about WinEdt, PFE (programmers' File editor) & Alpha tk. Wordpad (which comes free with windows) is also decent, avoid notepad though it is too limited.

A C++ Compiler: On Linux, this is usually installed by default (unless you are using one of the more user-friendly modern distributions like Ubuntu). You must know what version of gcc is installed

on your system, simply type 'gcc -v' and note the answer down. Red Hat distributions (and Red Hat like distributions such as Fedora and CentOS) often ship with two compilers, so you have a choice.

On Windows, the best option (given the rest of the software) is Microsoft Visual Studio .NET 2003.

At Yale, this can be downloaded for *free!* from the Yale MSDN Academic Alliance home page (`http://babs.its.yale.edu/msdnaa/`), please register and download this. The Visual Studio editor is also very nice and can be used as a generic text editor if desired. You will need to access this from a machine on the Yale Network, if doing this from home use Yale VPN first to connect to this. (VPN = virtual private network, see
`http://its.med.yale.edu/software/remoteaccess/vpn/` for details/software.)

On Mac OS X (10.4 +) you will need to install the development tools from the system DVD, also install X11 while you are at it. Other than that, Mac OS X, is very similar to Linux.

Subversion: Subversion, described in detail in the Subversion Book – this can be found online at `http://svnbook.red-bean.com/`). Subversion is the revision control system we will use for this class. Please make sure that it is installed on your system.

Windows users should install the more user friendly TortoiseSVN package instead.
See `http://tortoisesvn.tigris.org/`.

CMake: CMake – or cross make `www.cmake.org` is also needed for the C++ portion of this class. This is included with the windows version of BioImage Suite (see above).

1.3 Useful Books

First, a note that many of the books are available on-line from the books24x7 web-page which is at: `http://library.books24x7.com/login.asp?ic=0`. This web-page is only available for machines on the Yale network (or use Yale VPN, see above).

C++

- C++ Demystified: A Self-Teaching Guide by Jeff Kent McGraw-Hill/Osborne. *If you have no experience with C++, this should get you started.*
- C++: A Beginner's Guide, Second Edition (Beginner's Guide) (Paperback) by Herbert Schildt Publisher: McGraw-Hill Osborne Media; 2 edition (December 3, 2003)
- Professional C++ by Nicholas A. Solter and Scott J. Kleper Wrox Press.

Tcl/Tk

- Tcl/Tk, Second Edition: A Developer's Guide (The Morgan Kaufmann Series in Software Engineering and Programming) (Paperback) by Clif Flynt Publisher: Morgan Kaufmann; 2 edition (May 5, 2003)

- Practical Programming in Tcl and Tk (4th Edition) (Paperback) by Brent Welch, Ken Jones, Jeffrey Hobbs Paperback: 960 pages Publisher: Prentice Hall PTR; 4 edition (June 10, 2003)

VTK/ITK/CMake etc

- The VTK User's Guide, Version 4.4 (Paperback) by Kitware Inc. *This is out of print right now.*
- The ITK Software Guide 2.4 (Paperback) by Luis Ibanez; William Schroeder Publisher: Kitware, Inc. *This is freely available as .pdf file*
- Mastering Cmake 2.2 Edition (Paperback) by Ken Martin; Bill Hoffman Paperback: 250 pages Publisher: Kitware, Inc. (February 24, 2006)

Other

- Version Control with Subversion (Paperback) by C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick Paperback: 304 pages Publisher: O'Reilly Media (June 22, 2004) *This is freely available online.*

Assignment

Part I: Ensure that you have access to a workstation with all the necessary software for this class (either your own laptop or access to a networked workstation in the MRRC). If you need help let me know.

Part II: Start BioImage Suite BrainSegment and load one of the default images. This will ensure that everything is in place.

Part III: Enter (in a text editor), save and execute the following trivial Tcl script. This will ensure that you know how to start tcl scripts on your computer. (On Windows use the BioImage Suite Console as a command line terminal.)

```
#!/bin/sh
# the next line restarts using wish \
    exec tclsh "$0" "$@"

set m0 $tcl_platform(user)
set m1 $tcl_platform(os)
set m2 $tcl_platform(osVersion)
set m3 [ info nameofexecutable ]
set m4 $tcl_version
```

```
puts stderr "\nHello User $m0 on [ info hostname ]"
puts stderr "You are using $m1 ($m2)"
puts stderr "Your interpreter is $m3 version $m4\n"
exit
```

Chapter 2

Revision Control With Subversion

The goal of the first session is to introduce the key concepts of revision control and to demonstrate it's use using the subversion package (`http://subversion.tigris.org/`).

2.1 Introduction

In setting out to teach this class, one my goals was not only to teach a programming techniques, but to introduce (however cursorily) additional topics/tools that are useful for large-scale software development. As stated in the Subversion book:[1]

Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing those changes the next day. But the usefulness of version control software extends far beyond the bounds of the software development world. Anywhere you can find people using computers to manage information that changes often, there is room for version control. And that's where Subversion comes into play.

Revision control enables, especially within the context of software development in a research setting, the following:

- The ability to return to an earlier (presumably functional) version of the software following a failed attempt to improve upon it – this can be a real life-saver.
- The ability to return to an older version of the software for the purpose of quantifying improvements.
- The ability to collaborate with other developers on a complex project and easily share code updating, editing.
- A build in backup mechanism for key documents (if the code repository is on a different server).

[1] Much of the contents of this chapter is derived from the book Version Control with Subversion by Ben Collins-Sussman, Brian W. Fitzpatrick and C. Michael Pilato, which is freely available online (`http://svnbook.red-bean.com/`).

2.2 Alternative Revision Control Systems

Subversion is but one of many choices in a revision control systems, although in my opinion it is as good as anything out there. It is fairly user friendly (within the bounds of a complex system), it works on just about any operating system and there are nice user-friendly GUI versions, especially the TortoiseSVN package.
See `http://tortoisesvn.tigris.org/`.

Subversion was started to address issues with the CVS (`http://www.nongnu.org/cvs/`) which is probably the most popular open source revision control system. CVS in turn builds upon RCS (Revision Control System, `http://www.gnu.org/software/rcs/rcs.html`, which was the probably the first freely available source control setup).

The Microsoft Visual Source Safe system

`http://msdn.microsoft.com/vstudio/products/vssafe/default.aspx` is another alternative.

All of the above work much better than no source control setup, but my recommendation is to use Subversion.

2.3 Key Concepts

Repository: The place where the "official" version of the code sits. In the case of subversion, the repository can be either separate location within the local computer filesystem, or accessed remotely from a server. A repository structure can resemble of filesystem and multiple projects can share a single repository.

What differentiates a repository from simply a file server (or another directory) is that it remembers every change ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has (unless one is using a web browser) the ability to view previous states of the filesystem. For example, a client can ask historical questions like, "What did this directory contain last Wednesday?" or "Who was the last person to change this file, and what changes did he make?". These are the sorts of questions that are at the heart of any version control system: systems that are designed to record and track changes to data over time.

Repositories are accessed using their address which has a similar form to a web-page URL. There are two types of repositories, local repositories that live on the filesystem of your own computer that have a prefix (file://) or remote repositories that live on the filesystem of a server and are accessed through a web-server (and have a prefix http:// – an example of this is the class repository) or even a custom subversion server (svn://).

Revision Number: Unlike CVS and RCS (for those familiar with these) subversion maintains a single global revision number. Any changes to the repository result in the version number being increased. Each revision number represents a potential version of the files and it can be used to obtain the state of all files at that point in time. For example, consider the case when one is working on a project and at revision 77 they have reached a good working set of code. This is used to then generate results for

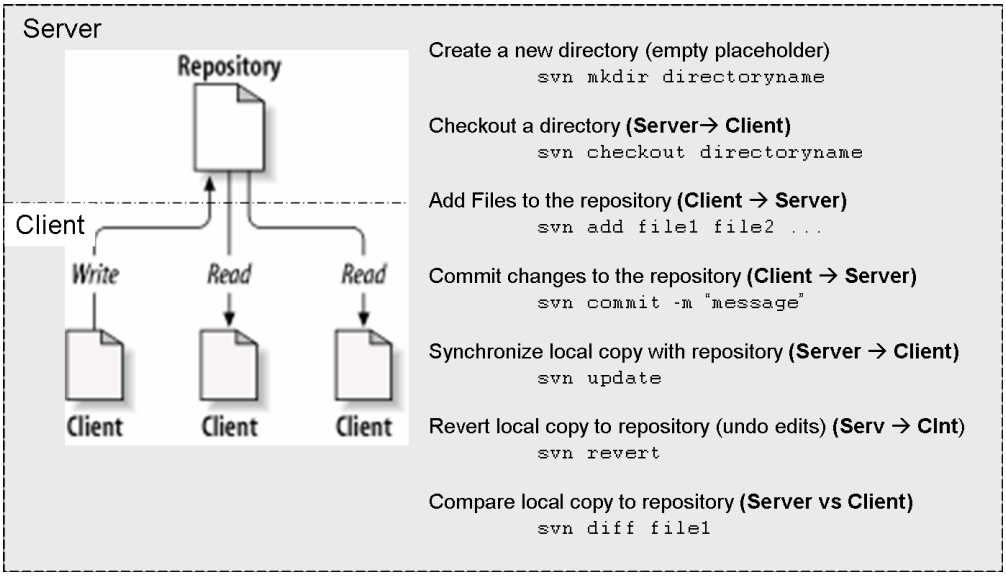


Figure 2.1: **An overview of subversion.** There are additional svn commands such as `svn del`, `svn copy`, `svn status`.

a paper. Subsequently improvements are made to the code (raising the revision number to say 110) and the question is raised as to how effective those changes were. From the repository we can obtain a version of the code from revision 77 and test it against the current version.

When downloading BioImage Suite some of you may have noticed that the comment at the top of the page "SVN repository version 507". This is a reminder of the exact "cut" of code in this release, and it enables us to go back to it, were subsequent improvements to break something!

Access: First prior to starting work on a project a user "checks-out" a copy of the code from the repository. (If this is a new project, then a new directory is made in the repository and then checked out to which files may be added). The key command here is `svn checkout`.

Workflow: Prior to starting work, the user first synchronizes their local copy with the repository (`svn update`). Then he/she performs changes to it (using a text editor presumably) and/or adds or deletes files as needed (`svn add`, `svn delete`). Then he/she (if it all goes well) "commits" (`svn commit`) the changes to the repository.

A different user then "updates" their local copy to re-synchronize with the repository.

Notes: While source control comes into its own for source code, it is also useful for storing and keeping track of other documents (e.g. text), binary objects (images, powerpoint slides).

Users of TortoiseSVN should use the graphical user interface to accomplish these operations. The buttons have the exact same name as the commands described above.

2.4 Repositories used in This Class

In this class each student will need to access (at least) two directories on the class subversion repository. The first is located at:

`http://boreas.med.yale.edu/repos/enas920/00handouts/`

It contains all the "official" class notes, examples etc. Please create an empty directory somewhere (let's call it handouts) and then execute

```
cd handouts; svn checkout http://boreas.med.yale.edu/repos/enas920/00handouts .
```

The trailing dot is critical! This will place all files on the server in this directory. Subsequently as I add files there, you can re-synchronize the files with the server by simply typing `svn update` in the handouts directory. You have no permissions to modify files in this directory!

The second repository is located at:

`http://boreas.med.yale.edu/repos/enas920/‘netid‘/`,

where one should replace 'netid' with their netid. This is your own private directory for uploading solutions to assignments. I suggest you make a directory on your computer (let's call it homework) and then execute as above:

```
cd homework; svn checkout http://boreas.med.yale.edu/repos/enas920/netid .
```

The trailing dot is critical! Once you have a completed solution, place it in your homework directory (i.e. copy the file) and then add it to the repository

```
svn add mysolution.tcl
svn commit -m ‘‘adding solution’’
```

The svn add adds the file the 'todo' list for the next synchronization. This happens when the svn commit command is executed.

You can also create your own repositories on your local machines.

Assignment

Skim Chapters 1 and 2 of the Subversion Book and read **carefully** chapter 3.

Part I: Checkout the handouts repository from the class server

Part II: Checkout your own private homework directory from the class server.

Part III: Create a file called assignment2.txt in your homework directory contain the line.

```
This is the solution to assignment 2
```

Add this to the repository and commit the changes.

Then edit the file to read like:

```
This is the solution to assignment 2.
Here is another line.
```

Compare this version with the version in the repository. Can you tell what has changed?

Change this again to:

```
This is the solution to assignment 2.
This is a replacement for the second line.
```

Repeat the comparison. Commit this new file to the repository.

Delete the file "assignment2.txt" from your local directory. Then type 'svn update'. What happens in this case?

Note: We will use subversion extensively for the rest of the book so please play with it and get comfortable using it! Once you get used to subversion you will never know how you survived without it.

Part II

Programming with Tcl/Tk

Chapter 3

Introduction to Tcl

The goal of the this chapter (and the next one) is to introduce the Tcl (pronounced tickle) scripting language. We will make extensive use of TclTutor, a computer aided instruction package for learning Tcl. The TclTutor can be obtained, either from the class subversion repository or from `http://www.msen.com/ clif/TclTutor.html`. The clearing house on the web for all things Tcl is the Tcl Wiki, which can be found at `http://wiki.tcl.tk`.

3.1 Introduction

Learning a new programming language can be a challenge, however there are few programming languages much easier than Tcl. For the next two chapters we will work first on basic Tcl concepts and more advanced (modular programming) concepts. The main learning tool is the TclTutor which has 44 different lessons covering just about everything in the core Tcl language. In table 3.1 I list the lesson titles and mark some of them as optional – the hint is that you should focus on the other ones first.

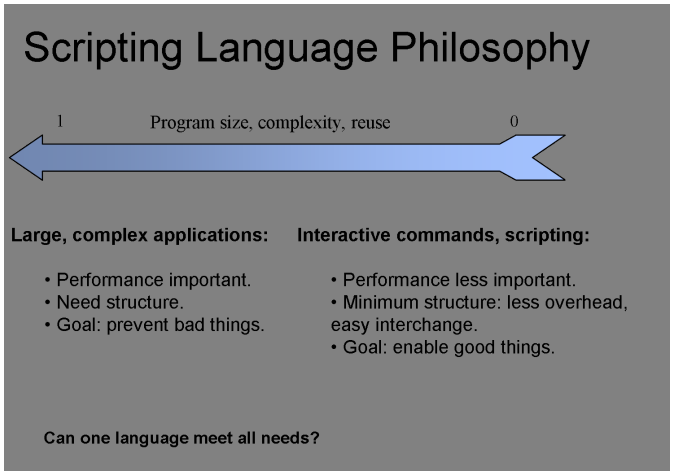
Tcl is an interpreted language. The commands are executed inside a special shell (the interpreter). There are three different shells that we will use for this class:

- **tclsh** – the basic Tcl language shell.
- **wish** – an extended shell which includes the Tk graphical interface toolkit.
- **vtk** – an extended shell that includes the Visualization Toolkit libraries (VTK)

All these shells are available as part of BioImage Suite. (On Windows use the BioImage Suite Console as a command line terminal.)

3.2 Rationale

In this class, I will present a hybrid programming approach that leverages both a scripting language (Tcl) and a more traditional compiled language (C++). The rationale for the hybrid approach is shown in Figure 3.1. While there are other scripting languages, such as Python, Ruby, Perl, Tcl in my mind is



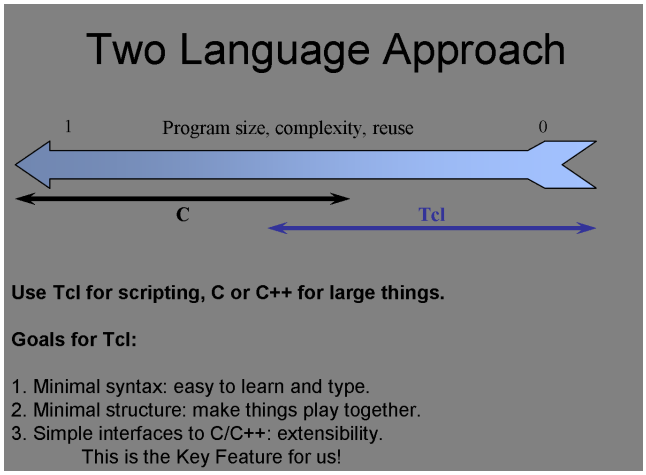


Figure 3.1: Scripting Languages can be powerful complements to more traditional compiled languages (e.g. C++). They enable a more flexible development environment.. See `http://alumni.media.mit.edu/~tpminka/PLE/components.html`.

preferable because it is the one with the best integration with a graphical user interface toolkit (Tk). In fact, most of the other languages, directly or indirectly invoke Tk and hence an understanding of Tcl is helpful in learning these languages as well.

3.3 A Guided Tour of Tcl

This section presents a guided tour of Tcl. It is meant to highlight key features at a glance and to supplement the TclTutor. In all examples below, lines starting with % indicates the commands typed in where as the following line(s) is (are) the expected output.

Printing: `puts` is the print statement in Tcl. It can also be used to print to a file. It is the Tcl equivalent of the C commands printf and fprintf.

```
%  puts "Hello World"
Hello World
```

Variables: Tcl stores data as strings, and only converts to numbers when necessary for calculations or comparisons. The assignment operator in Tcl is set. When set is called with two arguments, as in:

```
% set foo 10
10
% set bar "Hello"
Hello
% puts stderr "$foo $bar"
```

```
10 Hello
% set temp hello$foo
hello10
% set temp foo
foo
```

Flow control: For loops in Tcl have the form:

```
for { set i 0 } { $i < 10 } { incr i 1 } {
  puts stdout $i
}
```

where `incr` is the increment operator which increases the value of the counter variable i by one. An alternative is the 'while' loop which has the syntax:

```
set i 0
while { $i < 10 } {
  puts stdouti "i=$i"
  incr i 2
}
```

Conditionals: Tcl has both an `if .. elseif ... else` and a `switch` construct for conditional operations. These take the form

```
for { set a -3 } { $a<=3 } { incr a } {

    puts stderr "Testing for a=$a"

    if { $a > 2 } {
puts stderr "\t a is greater than 2"
    } elseif { $a >= -2 } {
puts stderr "\t a is between -2 and 2"
    } else {
puts stderr "\t a is less than -2"
    }
}
```

Both the elseif and else constructors are optional, note though that unlike C/C++ you must use `elseif` as one word as opposed to "else if".

The switch statement has the form:

```
switch -exact -- $osmode {
  "unix" {
    set libraryprefix "lib"
    set librarysuffix ".so"
  }
  "windows" {
    set libraryprefix ""
    set librarysuffix ".dll"
  }
}
```

Math Operations: All math operations make use of the `expr` command. The following statement is *not valid* – unfortunately because by default everything is a string!

```
% set i 2+2
2+2
% set i [ expr 2+2 ]
4
```

`expr` parses the operation numerically i.e. not as a string!

Lists: Lists are a complex variable structure, and they are literally everywhere in Tcl. If in doubt as to what something is, it is a list! A new list can be created using the `list` command.

```
% set l [ list 1 2 3]
1 2 3
```

We can access an element of the list (first element has index 0) using the `lindex` command:

```
% set a [ lindex $l 1 ]
2
```

To get the length of the list (i.e. how many items it has) use the `llength` command.

```
% set a [ llength $l ]
3
```

There are other commands such as `lappend`, `lsort`, `linsert` etc. A good understanding of how lists work is critical for understanding Tcl!

Strings: Every variable is implicitly a string – this is the key strength *and* weakness of Tcl. Strings can be manipulated using the `string` command

```
% set l [ string length Hello ]
5
% string range "Hello" 2 4
llo
% string index "Hello" 1
e
```

There are lots of other options.

Associative Arrays: Another complex variable structure in Tcl is the associative array. To create an array (implicitly) use:

```
% set a(1) "Hello"
% set a(2) "Help"
% set a(other) "Complicated"
% puts stdout "$a(1), $a(2), $a(3)"
Hello, Help, Complicated
```

Arrays can be modified using the `array` command.

File Input/Output: Tcl uses the two standard output channels stderr and stdout (standard output – default and standard error). New files can be created using the `open` command.

```
set fileid [open "/winnt/temp/testfile" w]
puts $fielid "This is the first line"
puts $fielid "This is the second line"
close $fileid
```

To read an existing file use:

```
set fileid [open "/winnt/temp/testfile" r]
gets $fielid firstline
gets $fileid secondline
puts stdout "Read\n $firstline \n $secondline"
close $fileid
```

Filename Manipulation: The `file` command can be used to parse filenames. For example consider the following:

```
set fname /home/papad/vtkpxcontrib/CMakeLists.txt
set a [ file extension $fname ]
.txt
% set a [ file rootname $fname ]
/home/papad/vtkpxcontrib/CMakeLists
% set a [ file tail $fname ]
CMakeLists.txt
% set a [ file dirname $fname ]
/home/papad/vtkpxcontrib
% set a [ file size $fname ]
1024
```

There are lots of options for copying files, deleting files, moving files/directories etc.

Arguments to a Script The arguments passed to the script on execution are stored in the global variable `argv` which is a list of all arguments. Unlike C/C++ the first item in the list is NOT the name of the script itself, this is stored in the variable `argv0`.

If a script is executed as:

`tclsh myscript.tcl a.hdr 10.0`

then argv is the list a.hdr,10.0.

Assignment

1. Download the TclTutor package from handouts repository
2. Also download the `tkconsole.tcl` from the same location

   The URL is you need to do a manual download is

   `http://boreas.med.yale.edu/repos/enas920/00handouts/session3/`, if you have successfully installed subversion and checked-out the handouts directory a simple svn update will place

both of these files into the 'session3' subdirectory.

3. Unzip the TclTutor package and execute it using

   ```
   wish TclTutor.tcl
   ```

4. Work through lessons 1-9, 14-19, 22-24. Most of these are fairly short and should not take up too much time. Just try to grasp the basic concepts.

Note: This assignment is really the first half of the assignment printed at the end of the next chapter.

Lesson 0		Introduction
Lesson 1		Simple Text Output
Lesson 2		Assigning values to variables
Lesson 3		Evaluation & Substitutions 1: Grouping arguments with ""
Lesson 4		Evaluation & Substitutions 2: Grouping arguments with
Lesson 5		Evaluation & Substitutions 3: Grouping arguments with []
Lesson 6		Results of a command - Math 101
Lesson 7		Textual Comparison - switch
Lesson 8		Numeric Comparisons 101 - if
Lesson 9		Looping 101 - While loop
Lesson 10		Looping 102 - For and incr
Lesson 11		Adding new commands to Tcl - proc
Lesson 12		Variations in proc arguments and return values
Lesson 13		Variable scope - global and upvar
Lesson 14		Tcl Data Structures 101 - The list
Lesson 15		Adding & Deleting members of a list
Lesson 16		More list commands - lsearch, lsort, lrange
Lesson 17		String Subcommands - length index range
Lesson 18		String comparisons - compare match first last wordend
Lesson 19		Modifying Strings - tolower, toupper, trim, format
Lesson 20	Optional	Regular Expressions 101
Lesson 21	Optional	More Quoting Hell - Regular Expressions 102
Lesson 22		Associative Arrays.
Lesson 23		More Array Commands - Iterating through array
Lesson 24		File Access 101
Lesson 25		Information about Files - file, glob
Lesson 26	Optional	Invoking Subprocesses from Tcl - exec, open
Lesson 27	Optional	Learning the existence of commands and variables ? - info
Lesson 28	Optional	State of the interpreter - info
Lesson 29	Optional	Information about procs - info
Lesson 30		Modularization - source
Lesson 31		Building Libraries of procs - unknown, info library
Lesson 32		Creating Commands - eval
Lesson 33		More command construction - format, list
Lesson 34		Substitution without evaluation - format, subst
Lesson 35		Changing Working Directory - cd, pwd
Lesson 36	Optional	Debugging & Errors - errorInfo errorCode catch error return
Lesson 37	Optional	More Debugging - trace
Lesson 38	Optional	Command line arguments and environment strings
Lesson 39	Optional	Leftovers - time, unset
Lesson 40	Optional	Channel I/O & socket, fileevent, vwait
Lesson 41	Optional	Time and Date - clock
Lesson 42	Optional	More channel I/O - fblocked & fconfigure
Lesson 43	Optional	Child interpreters

Table 3.1: List of Lessons in the Tcl Tutor.

Chapter 4

Advanced Topics in Tcl

The goal of the this chapter is to introduce key modularization concepts in Tcl. In particular I will discuss the following: procedures, packages and namespaces. Just like in the previous chapter, we will make extensive use of TclTutor, a computer aided instruction package for learning Tcl. The TclTutor can be obtained, either from the class subversion repository or from `http://www.msen.com/ clif/TclTutor.html`. A reminder that the clearing house on the web for all things Tcl is the Tcl Wiki, which can be found at `http://wiki.tcl.tk`.

4.1 Introduction

Writing a long program in a single chunk, while doable (I inherited some cardiac code in C that was a single main function that run over 1,000 lines!) is not the best idea in terms of maintenance, flexibility and code reuse. Most programs of any length consist of multiple pieces that can then be reused for similar tasks in ways more elegant than simple copy and paste (although all of us do a fair amount of this too!).

The most elementary form of modularization is the packaging of code into specialized functions that usually perform simple tasks. Functions are termed `procedures` in Tcl and are specified using the `proc` command.

The next step is to move some of these functions into a separate file (or collection of files) to form a reusable library, code that is invoked by multiple programs. These separate files can either be invoked directly (using the `source` command which the rough C-equivalent of #include) or by using the `package` mechanism in Tcl which provides additional functionality.

The use of multiple libraries in large programs can often lead to name conflicts, as for example in the case of using two separate libraries each of which defines a procedure called LoadImage. Namespaces, are an elegant way of avoiding such conflicts. In some respects namespaces are similar to classes in object-oriented languages in that they can provide for code and data encapsulation.

4.2 Procedures

The core common method for modularizing code is the use of procedures. A procedure in Tcl is defined using the proc command. The syntax is:

```
proc myfunction { argument1 argument2 } {
  ... some code
}
```

For example, a trivial script (script4-1.tcl) which contains and addition procedure looks like:

```
proc addtwonumbers { x y } {
  return [ expr $x + $y ]
}

set a [ addtwonumbers 2 3 ]
set b [ addtwonumbers 3 $a ]
puts stderr "a=$a, b=$b"
```

Returning Multiple Values: As a general rule, all complicated issues in Tcl are resolved using lists. This applies to the present case. A procedure can only return "one thing", if you need to return multiple things, make a list and return the list! Consider the following filename parsing procedure (script4-2.tcl has a complete example)

```
proc parsefilename { filename } {
    set path [ file dirname $filename ]
    set tail [ file tail    $filename ]

    set rootname [ file root $tail ]xs
    set extension [ file extension $tail ]

    set output [ list $path $rootname $extension ]
    return $output
}
```

Scoping: The use of procedures (and namespaces as we will discuss later) introduces an additional complication, the issue of scoping. Variables defined inside a procedure are, by definition, local variables to that procedure the main program (global variables) are not available in a procedure, unless explicitly accessed using the `global` command. This is different from other languages, such as C, where global

variables are always available.

Consider the following case (script4-2b.tcl)

```
set defaultextension .hdr

proc changeextension { filename } {

    global defaultextension

    set rootname [ file root $filename ]
    set extension [ file extension $filename ]

    set output [ list "${rootname}${defaultextension}" ]
    return $output
}

puts stderr [ changeextension myfile.img ]
```

Here the variable defaultextension is made available to the changeextension procedure using the `global` command, which tells the procedure that the variable defaultextension is the global variable defaultextension, as opposed to a local variable to be used within the procedure.

In general, global variables are a bad idea. They are a prime source of what is often termed "name pollution". A good metric of code quality is the total number of global variables, the fewer the better. If you need to use global variables, I suggest that you group them together in an associative array. For example the script above could have been rewritten as (script4-2c.tcl)

```
set parameters(defaultextension) .hdr

proc changeextension { filename } {

    global parameters

    set ext $parameters(defaultextension)

    set rootname [ file root $filename ]
    set extension [ file extension $filename ]

    set output [ list "${rootname}${ext}" ]
    return $output
}
```

The associative array trick has the advantage of keeping the total number of globally accessible names to one! There are other constructs available (particularly upvar) which I recommend you avoid!

Note: There are ways to make procedures take variable numbers of arguments (using the **args** argument) and to define default values for the arguments.

4.3 Using Multiple Files

The source command Consider the case where the parsefilename procedure is placed in a file `mylib.tcl`. A separate script (script4-3.tcl) can invoke this using:

```
# The next commands includes mylib.tcl
source mylib.tcl

set testfilename "c:/temp/hello.txt"
set results [ parsefilename $testfilename ]
puts stderr "\tThe path name is: [ lindex $results 0 ]"
puts stderr "\tThe main name is: [ lindex $results 1 ]"
puts stderr "\tThe extension is: [ lindex $results 2 ]"
```

Packages An additional level of flexibility (or complexity) is introduced using the `package` mechanism. A package is a collection of procedures, or even more simply a name for the contents of a .tcl file that is separate from the actual filename.

To create a formal package we need two things: (a) the package definition – using the `package provide`, typically the top file in the .tcl file containing the code and (b) to create an index file (`pkgIndex.tcl`) in the directory containing the .tcl file)or files) that matches package names to filenames.

Consider the `mylib.tcl` file. It's first few lines look like:

```
package provide mylibrary 1.0

proc parsefilename { filename }
```

The first line specifies that this file contains the abstract entity mylibrary version 1.0. A different file could contain mylibrary version 2.0 etc.

The pkgIndex.tcl file can either be created automatically using the pkg_mkIndex command (in the tclsh interpreter) or by hand, which is my preferred option. In the simple case of the directory containing a single package this takes the form:

```
package ifneeded mylibrary 1.0 [list source [file join $dir mylib.tcl]]
```

which means: if you need mylib version 1.0, this is located in the file mylib.tcl in the same directory as pkgIndex.tcl. If there are more than package in the directory, each will have an entry in pkgIndex.tcl.

The script (script4-4.tcl) can be rewritten to make use of the package mechanism as:

```
# Append to auto_path the directory containing the current script
lappend auto_path [ file dirname [ info script ]]
package require mylibrary 1.0
```

The variable auto_path contains a list(!) of paths in which the interpreter will look for packages. We append to this the directory containing the current script. If we want to invoke other packages (e.g. from BioImage Suite which are stored in /usr/local/bioimagesuite/base) we could add the command

```
lappend auto_path /usr/local/bioimagesuite/base   # (unix)
lappend auto_path c:\yale\bioimagesuite\base      # (windows)
```

The second line (package require) tells the interpreter to look in all the pkgIndex.tcl files in the directories listed in auto_path to find which one has a match for the requested item mylibrary version 1.0.

The main reason for using packages has to do with the ease with which different versions of the code can be swapped in or out, or moved about. It also makes using other libraries (written by other people easier) as there is no need to know exactly which file to source, only which package to request. In a newer version of the libraries the filenames may change but with an appropriate new pkgIndex.tcl file old code can remain functional. This provides a level of abstraction between the code interface and the code implementation, which is a common theme in many of the more advanced programming techniques that we will encounter. In general the interface (what the user uses to access code), need not directly match the implementation (the underlying code). The interface often acts as a tidy easy to see showroom for a messy storage room (implementation) where the real work is done!

4.4 Namespaces

Just like global variables pollute the name space, procedures can also have the same effect. Consider the aforementioned example of needing to use two separate libraries each of which defines a procedure called LoadImage. In such a case, we have a naming conflict which means that only of the libraries can be used. There are two ways around this: (i) the use of suitably long names for all procedures (i.e. xpMyUtilLoadImage) to ensure unique names (but unreadable code) or the (ii) the more elegant way of using namespaces.

A namespace is a collection of procedures and variables that are grouped together and to which the interpreter adds implicitly a prefix that distinguishes them from other code. An example (newname.tcl)

hopefully will clarify the concept:

```
namespace eval mynewname {

# All code from this line

    variable counter
    set counter 0

    proc newname { } {
        variable counter
        incr counter
        return "new$counter"
    }

    proc longnewname { } {

      set a [ newname ]
      set b long${a}
      return $b
    }

# to this line is enclosed in the mynewname namespace
}
```

A namespace is created using the `namespace eval namespacename` command whose argument (a list!) is the entire code in the namespace. The variable `counter` and the procedure `newname` live inside the namespace.

Inside the namespace we can call the procedures newname and longnewname using their regular names (i.e. newname and longnewname). Variables defined inside the namespace (e.g. counter) can be made accessible inside each procedure using the `variable` command which is the namespace equivalent of the `global` command.

Accessing the code from outside the namespace is a different game (script4-4.tcl).

```
lappend auto_path [ file dirname [ info script ]]
package require mynewname

set firstname  [ ::mynewname::newname ]
set secondname [ ::mynewname::longnewname ]

puts stderr "The firstname is $firstname, the second name is $secondname"
puts stderr "The current value of the counter is $::mynewname::counter"
```

Here all functions and variables defined in the mynewname namespace can be accessed only by adding a ::mynewname prefix to their names. Hence the counter variable becomes ::mynewname::counter and newname becomes ::mynewname::newname etc. While this may seem unwieldy, the general idea is that most functions in a namespace are only meant to be accessed from within the namespace (the implementation!) and only one or two functions are to be accessed from the outside (the interface). This type of functionality is already a step towards object oriented programming, that we will discuss later.

Also note, that (i) symbols can be exported from a namespace into the global scope (a bad idea, so I will not cover this) and (ii) that namespaces can be nested, i.e. a namespace can be defined inside another one. In this case we can have paths like ::mynewname::mysubnumspace::myfunction and so on. Again though, the general idea is that stuff in a nested namespace is not meant to be accessed from outside the namespace itself or at most it's parent namespace.

Assignment

General hint. The construct:

```
while { [ gets $fileid line ] >=0 } {
...
}
```

will read a line at a time from file $fileid and place it into the variable line until there is nothing left to read from the file.

Tuning in assignments: Make a directory session4 and add it to you repository directory. Place all solution in this.

1. TclTutor: Work through lessons 11-13, 24,30,34, in addition to those recommended in the last assignment.

2. File search/replace: Write a script that opens file script4-4.tcl and replaces all stderr with stdout and save the result in script4-4-stdout.tcl. Hint: Take a look at the `regsub` command.

3. Package Use: Write a short library (name it mymath.tcl) with two functions add and subtract. Make this declare a package (package provide mymath) and add this to the index file (pkgIndex.tcl). Then write a second script that reads a set of numbers from a provided file (numbers.txt) and for each line print out the original numbers, as well as their sum and their difference. (Hint: you may want to investigate the `scan` command for parsing each line as it is read from the file.)

4. Arguments: Modify the script in exercise 2 so that the filename of numbers.txt is not hard-wired but read as an argument on the command-line, i.e. it should be executed as `tclsh script3.tcl numbers.txt`.

5. Conditionals: Modify the script in exercise 3 so that the program only prints the difference of the numbers if it is positive, otherwise it prints "the second number is bigger than the first".

In the next chapter, we will examine the creation of graphical user interfaces using the Tk Toolkit.[1]

[1]Incidentally, since somebody asked, the domain Dot TK (.tk), is the domain of the island of Tokelau. Extra Credit: Where is Tokelau?

Chapter 5

An Introduction To Tk

Tk is an open source, cross-platform widget toolkit, that is, a library of basic elements for building a graphical user interface (GUI). This chapter will introduce the basic concepts in Tk, while the next chapter will handle more advanced/esoteric aspects of the toolkit. The clearing house on the web for all things Tcl is the Tcl Wiki, which can be found at `http://wiki.tcl.tk`. A good introduction to Tk is available in chapter 18 of Brent Welch's book Practical Programming in Tcl and Tk – the chapter is freely available online at `http://www.beedub.com/book/2nd/TKINTRO.doc.html`. Much of this handout is based on a set of lecture notes by Marcel Jackowski that was presented in a Yale IPAG seminar.

5.1 Introduction

The task of writing a program with a graphical user interface (GUI) for the first time can be a major challenge. GUI-based programming is a vastly different concept from standard command-line (CLI) programming. In CLI programming the program (and by implication the programmer) is in control. There is only one entry point into the program and the general flow through the program (i.e. order in which statements are executed) is more or less predictable with few permutations. In GUI-programs, in contrast, the user is really in control, as the master. The program (and hence the programmer) is really the slave/butler whose job is to react to events initiated by the master (e.g. clicking on a button) with appropriate responses (often captured in terms of callback procedures). A key part of the job of a programmer in this environment, is to ensure that only valid actions are taken. For example, an image can be displayed once it is loaded but not before. In such a case, clicking the DisplayImage button prior to the LoadImage button should result in an error (a helpful user friendly hand-holding message) as opposed to an attempt to display a non-existent image which will likely crash the program.

The execution of GUI-based program consists of two distinct phases (i) initialization – which includes the creation of the GUI and (ii) the event loop – which for the most part is the process of waiting on the user to generate some event which requires an appropriate response (event handling). Once an event is handled the program just sits there until the next event is generated, or the user issues an exit command which stops the program (or in many cases the program crashes, but one should try to avoid this scenario!)

5.2 What is Tk?

Tk is a freely available open-source GUI toolkit implemented with Tcl and C. It runs on multiple platforms: X/Motif, Win32 GUI, Mac GUI. Its simplicity enables fast development of GUIs with few lines of code. It also allows for easy creation new GUI controls. Tk is used both in commercial packages (e.g. Mayo Clinic's Analyze) and other large scale Medical Image analysis tools (e.g. Slicer, BioImage Suite).

The following are the basic concepts in Tk Programming

- Windows (such as dialog boxes)
- Widgets: windows with a particular look and feel e.g. buttons, labels etc.
- Class commands: create different widgets (e.g. `button .a -text "Button" -command "exit"`)
- Widget commands: configure widgets (e.g. `.a configure -bg black`)
- Geometry management commands: place, pack & grid commands (e.g. `pack .a -side top`) -side top)
- Event bindings (`bind .a <ButtonPress> { puts %b }`)

A First Example: This script (script5-1.tcl) creates two buttons (Print and Exit) and waits for them to be pressed.

```
proc PrintButtonCallback { } {
    puts "Print Button Pressed!"
}

button .b1 -text "Print" -command  { PrintButtonCallback }
button .b2  -text "Exit"  -command { exit }
pack .b1 .b2   -side top
```

The two lines beginning with the button command create two button widgets (Print and Exit). The button command, much like all widget creating commands, has the syntax:

```
button widgetname -optionname optionvalue -optionname2 optionvalue2  ...
```

The widget name has the form `parent.newwidget`, where parent is the parent widget (which contains the new widget) and newwdiget is the particular name of the new widget. The base widget (which is created by default) has the name "." (dot), and in this case the widgetnames of the two buttons are ".b1" and ".b2". The button command has a number of options, two of the most common ones are -text (which specifies the name of the button) and -command (which specifies the action to be taken

when the button is pressed.) While the widgets are created using the button command, they do not appear until they are managed using the pack command which adds them to the display.

5.3 Widgets - Part I

The Core Tk (in a couple of weeks we will also discuss extensions such as the Iwidgets package) has the following core widgets:

1) button **2)** canvas **3)** checkbutton **4)** entry **5)** frame **6)** label **7)** labelframe **8)** listbox **9)** menu **10)** menubutton **11)** message **12)** tk_optionMenu **13)** panedwindow **14)** radiobutton **15)** scale **16)** scrollbar **17)** spinbox **18)** text **19)** toplevel - Create and manipulate toplevel widgets .

In addition to these, Tk also provides the following complex popup dialogs to simplify processes such as filename selection and generating message boxes to alert the user: **i)** tk_chooseColor - pops up a dialog box for the user to select a color. **ii)** tk_chooseDirectory - pops up a dialog box for the user to select a directory. **iii)** tk_dialog - Create modal dialog and wait for response **iv)** tk_getOpenFile - pop up a dialog box for the user to select a file to open or save. **v)** tk_messageBox - pops up a message window and waits for user response. **vi)** tk_popup - Post a popup menu .

Finally, Tk offers three different types of geometry managers, namely: **a)** place - which positions widgets at absolute locations, **b)** grid - which arranges widgets in a grid, **c)** pack - which packs widgets into a cavity, . These managers control how different widgets managed by a single parent appear on the screen. The most common manager is the packer – we will use this exclusively in this chapter.

In the next sections, we will briefly demonstrate the use of these widgets. Each widget can have additional options not covered here, the Tcl Wiki is a good source of information for this, or any Tk book.

5.3.1 Frames and Toplevels

The Frame Widget: The frame widget is a simple widget which simply acts as a container for other widgets. Frames are extremely useful and are used constantly in Tk. For example, consider the case when one would like to create two rows of buttons. The easiest way to do this is to (i) first create two frames, a top frame and a bottom frame and pack them into the parent window. Then (ii) the top row of buttons can be 'packed' into the top frame and the bottom into the bottom frame. Frame widgets are also useful as dividers by explicitly setting their height and/or width and their background color.

Class Command: frame, e.g. frame .top

Common Options: -bg (background), -height, -width

An example of the above settings is given in the script (script5-2.tcl) below.

```
proc PrintButtonCallback { number } {
    puts "Print Button number $number Pressed!"
}

frame .top;
```

```
frame .middle -height 10 -bg black;
frame .bottom
pack .top .middle .bottom -side top -expand false -fill x

button .top.b1 -text "Print 1" -command  { PrintButtonCallback 1 }
button .top.b2 -text "Print 2" -command  { PrintButtonCallback 2 }
pack .top.b1 .top.b2 -side left -expand true -fill x

button .bottom.b1  -text "Exit"  -command { exit }
pack .bottom.b1   -side left -expand true -fill x
```

The configure method can be used to set an option after the widget is created, e.g. to change the color of the frame .middle to blue use

```
.middle configure -bg blue
```

The cget command is used to return the current value of an option. To get the current value of the background color use:

```
set bgcolor [ .middle cget -bg ]
```

The Toplevel widget: This is similar to a frame widget with the key addition that instead of it being placed inside a parent widget, it is created as a standalone window. Consider the example below (script5-3.tcl)

```
wm protocol .dlg WM_DELETE_WINDOW { grab release .dlg ; wm withdraw .dlg}

wm title . "Main Window"
wm title .dlg "Second Dialog"

wm geometry . 800x60
wm geometry .dlg 400x40

pack .top .middle  -side top -expand false -fill x

button .top.b1 -text "Print 1" -command  { PrintButtonCallback 1 }
button .top.b2 -text "Print 2" -command  { PrintButtonCallback 2 }
button .top.b3 -text "Show Exit Dialog" -command  { wm deiconify .dlg }
pack .top.b1 .top.b2 .top.b3 -side left -expand true -fill x
```

```
button .dlg.b1  -text "Exit"  -command { exit }
pack .dlg.b1   -side left -expand true -fill x
wm withdraw .dlg
```

The `wm title` command can be used to set the title of the window and the `wm geometry` command can be used to set the dimensions of the window. The dialog is shown using `wm deiconify` and hidden (minimized) using `wm withdraw`.

Note: The line beginning with `wm protocol` is a critical piece of boiler plate code that ensures that the dialog box is not destroyed when closed. (This is particularly the case in Windows!)

5.3.2 Labels and Buttons

The Label Widget: The label widget is a simple widget that simply places some static text in a window.

Class Command: label, e.g. label .top -text "Some Text"

The Button Widget: This is effectively extension of the label widget which executes a command when clicked. It is important to remember that the command is executed in *global scope*. I strongly suggest avoiding lengthy code in the -command option; instead I recommend creating explicit callback handling procedures and using the -callback to call these.

The Checkbutton Widget: This presents the user with a "check" that can be set to on or off. The key additional option is that of a variable (global scope, hence global variable) whose value (1=on,0=off) reflects the state of the widget.

The Radiobutton Widget: This is similar to the checkbox widget ,but here, the widget variable is shared among multiple radiobuttons, each of which, sets it to a unique value – and only one of which can be on!

The Option Menu Widget: This is a different form of the radiobutton widget.

The following example (script5-4.tcl) demonstrates the use of the widgets mentioned this far.

```
proc Reset { } {
    global parameters
    set parameters(usegradient) 1
    set parameters(usehessian) 0
    set parameters(color) "red"
    set parameters(smoothness) "Low"
}
```

```
proc PrintStatus { } {
    global parameters
    puts stderr "Using Gradient : $parameters(usegradient)"
    puts stderr "Using Hessian  : $parameters(usehessian)"
    puts stderr "Color:           $parameters(color)"
    puts stderr "Smoothness:      $parameters(smoothness)"
}
Reset
frame .top; frame .middle -height 10 -bg black;
frame .middle2; frame .bottom
pack .top .middle .middle2 .bottom -side top -expand false -fill x

button .top.b1 -text "Print Status" -command  { PrintStatus }
button .top.b2 -text "Reset" -command  { Reset }
pack .top.b1 .top.b2 -side left -expand true -fill x
button .bottom.b1  -text "Exit"  -command { exit }
pack .bottom.b1   -side left -expand true -fill x

set w1 [ frame .middle2.left ]
set w2 [ frame .middle2.middle ]
set w3 [ frame .middle2.right ]
pack $w1 $w2 $w3 -side left -expand true -fill both

checkbutton $w1.1 -variable parameters(usegradient) -text "Use Gradient"
checkbutton $w1.2 -variable parameters(usehessian) -text "Use Hessian"
pack $w1.1 $w1.2 -side left -expand true -fill x

radiobutton $w2.a -variable parameters(color) -text "Red"   -value "red"
radiobutton $w2.b -variable parameters(color) -text "Green" -value "green"
radiobutton $w2.c -variable parameters(color) -text "Blue"  -value "blue"
pack $w2.a $w2.b $w2.c -side left -expand true -fill x

label $w3.a -text "Smoothness:"
tk_optionMenu $w3.b parameters(smoothness) "Low" "Medium" "High"
pack $w3.a $w3.b -side left -expand true -fill x
```

Note that all parameters are stored in global scope in the single global associative array `parameters`. This is to keep name pollution down. The original (default) values of the array are set by a call to the `procedure`; this is to allow for easy restoration of default values. Note also that any widget command returns a value which is the same as the actual widgetname. It is best in complex scripts to not hardwire the names of the widgets but to actually use variables.

The Entry Widget: The entry widget enables the input of a single line of text. The value of the widget is captured in the associated textvariable. Setting the variable changes the widget, and conversely, typing text in the widget changes the value of the variable. We can change the example above to switch from using the tk_optionMenu to an entry widget as follows (script5-5.tcl):

```
entry $w3.b -textvariable parameters(smoothness) -width 10 -relief sunken
```

An often useful modification is to disable text input into an entry widget and use it purely as a place to display the value of a variable. This is accomplished using:

```
$w3.b configure -state disabled; # to disable
$w3.b configure -state normal;   # to enable
```

The Scale Widget: This is a slider widget that allows the setting of a variable by dragging a scale. It has the syntax:

```
scale widgetname -variable variablename -from lowvalue -to highvalue
```

We can modify the previous example to add a scale as follows (script5-6.tcl):

```
scale $w3.b -variable parameters(smoothness) -from 1.0 -to 5.0 -digits 2 \
    -orient horizontal -resolution 0.1
```

Also the default value of parameters(smoothness) is now 1.5, as a scale only takes numerical values!

Note: In the next chapter we will continue with more complex widgets such as the listbox, the text and the canvas widget. We will also take a look at some of the more complex controls available for selecting filenames and colors, as well as providing feedback to the user in the form of messageboxes.

Assignment

Note: A more detailed Tk assignment appears at the end of the next chapter. The goal of this assignment is to get to be familiar with the basic concepts in Tk.

- Download Marcel Jackowski's presentation (jackowski_tk.ppt) and work through it.
- Take script5-3.tcl as a starting point, and rearrange the buttons such that "Print 1" is to the right of "Print 2".
- Take script 5-4.tcl as a starting point and modify the PrintStatus method to save the current parameters in a file parameters.txt in addition to printing them on the screen.
- Take script 5-4.tcl as a starting point, and modify the PrintStatus method to set the color of the .middle variable to be the same as the value of the parameters(color).

- Take script 5-4.tcl as a starting point, and rearrange the widgets so that the graphical user interface is "transposed" i.e. buttons are going top to bottom instead of left to right.

Chapter 6

Tk Part II

In this chapter, building on the foundations laid by chapter 5, we will explore some additional topics in Tk. This chapter is divided into two parts. First we will first examine some additional widgets, such as menus and common dialogs. Then we will look at a fairly complete application, a simple text editor, which demonstrates how a complex GUI-based application looks and illustrates some defensive programming concepts. A reminder that the clearing house on the web for all things Tcl is the Tcl Wiki, which can be found at `http://wiki.tcl.tk`.

- Asking "yes/no" type question and giving warning/error messages – `tk_messageBox`
- Getting a filename to load from – `tk_getOpenFile`
- Getting a filename to save into – `tk_getSaveFile`
- Selecting a directory – `tk_chooseDirectory`
- Selecting a color – `tk_chooseColor`

We examine each of these in turn, next, in the context of an example script (script6-1.tcl) shown below:

```
# Procedure for setting global values
proc Reset { ask } {
    global parameters
    if { $ask == 1 } {
      set ok [  tk_messageBox -type yesno -default no -title "Think again"
                -message "Reseting" -icon question  ]
      if { $ok == "no"  } { return }
    }
    set parameters(readname)  ""
    set parameters(writename) ""
    set parameters(directory) ""
}

# Set Background Color
proc SetColor { widget  } {
   set color  [ tk_chooseColor -title "Set Background Color"  -parent . ]
   if { [ string length $color ] > 0 } {
      $widget configure -bg $color
   }
}
```

```
# Execute one of the three dialogs to set filenames/directory
proc GetName { mode } {
   global parameters

   set filetype1 [ list "Text Files" [ list .txt .tex ] ]
   set filetype2 [ list "All Files" "*"  ]
   set f ""

   switch -exact $mode {
     "directory" { set f  [tk_chooseDirectory -title "Select Current Directory" ] }
     "readname" { set f [tk_getOpenFile -title Load \
                         -filetypes [ list $filetype1 $filetype2 ] ] }
     "writename" { set f [tk_getSaveFile -title Save -filetypes [ list $filetype1 ] ] }

   }
   if { [ string length $f ] < 1 } { return  }
   set parameters($mode) $f
}
Reset
# Create GUI one line at a time in frame.$i
for { set i 1 } { $i <= 3 } { incr i } {
  set parent [ frame .frame$i ]
   pack $parent -side top -expand false -fill x

   # First set the things that are different for each line
   switch -exact $i {
     "1" {  set name "Input File:"; set mode readname  }
     "2" {  set name "Save File :"; set mode writename }
     "3" {  set name "Directory :"; set mode directory }
   }

   # Create the GUI for each line -- note that use of eval
   label  $parent.1 -text $name -width 20
   entry  $parent.2 -textvariable parameters($mode) -width 80
   eval "button $parent.3 -text \"...\" -command { GetName $mode }"
   # Pack in order to make sure what appears and what can stretch
   pack $parent.3 -side right -expand false -padx 5
   pack $parent.1 -side left  -expand false -padx 2
   pack $parent.2 -side left -expand true -fill x
}

# Create bottom button bar
frame .bottom; pack .bottom -side bottom -expand true -fill x -pady 10 -padx 20
button .bottom.1 -text "Reset" -command { Reset }
button .bottom.2 -text "Color" -command { SetColor . }
button .bottom.3 -text "Exit" -command { exit }
pack  .bottom.1 .bottom.2 .bottom.3 -side left -expand true -fill x -padx 2
```

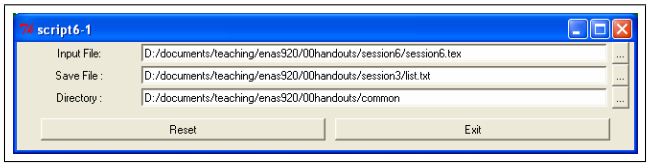


Figure 6.1: A snapshot of script6-1.tcl.

Message Box: The message box control, invoked in the Reset procedure, can be used to inform the user of some event (or error) and ask simple yes/no/cancel questions. It's key options are

- `-type` – this determines which buttons to display. Valid values include: yesno, yesnocancel, ok, okcancel ...
- `-icon` – this determines the icon in the dialog. This is one of: error, info, question ,warning.
- `-default` – name of the default button (e.g. yes)
- `-message` – message to display in the dialog
- `-title` – the tile of the window
- `-parent` – the parent window of the dialog which determines where it will appear.

The output of this command is the name of the button that was pressed (e.g. yes).

Choose Directory: The choose directory control `tk_chooseDirectory` can be used to select a directory. It's options include

- `-parent, -title` – same as above for message box.
- `-initialdir` – specifies the initial directory, if omitted the current directory is used.
- `-mustexist` – whether the directory must exist (set this to true or false)

The output of the tk_chooseDirectory command is the name of the selected directory, or an empty string if cancel was pressed.

Open File: This control (`tk_getOpenFile`) invokes the custom dialog for selecting a filename to open. It's options include:

- `-parent, -title, -initialdir` – same as above for choose directory, same goes for the output.
- `-initialfile` – this can be used to set the selection.
- `-filetypes` – this is a list of extensions allowed. This has to be of the form of a list of lists! Each component list has the format:
  `"Analyze Files" "*.hdr"` where the first argument is the "human-readable" type of the file, and the second specifies the filter to use (e.g. extension .hdr). The second argument can itself be a list(!) to specify multiple extensions as is done in the example.

Save File: This control (`tk_getSaveFile`) invokes the custom dialog for selecting a filename to write to. It's options include:

- `-parent, -title, -initialdir,initialfile, filetypes` – same as above for open file, same goes for the output.
- `-defaultextension` – this let's the program add a default extension (e.g. word's .doc) when none is specified"

Choose Color: This control (`tk_chooseColor`) brings up a dialog for the user to select a color. It has the following options:

- `-parent, -title` – same as above for open file.
- `-initialcolor` – this let's the program specify the initial color. The output of the tk_chooseColor command is the name of the selected color, or an empty string if cancel was pressed.

The "eval" command Towards the end of script6-1.tcl, one would have expected to see a line of the form:

```
button $parent.3 -text \"...\" -command { GetName $mode }
```

as opposed to:

```
eval "button $parent.3 -text \"...\" -command { GetName $mode }"
```

This is a subtle point of great importance. The problem with the first statement (i.e. the one without eval) is that the implicitly defined callback procedure `GetName $mode` is not executed at the time the `button` command is executed, rather it *will be executed* when the button is pressed. At this point in time, the variable `$mode` does not necessarily exist or have the appropriate value (it will always be set to, in this case, directory), which will cause unpredictable problems (Try it!). What we really want to specify, for example, during the first iteration of the loop ($i=0) is something like `GetName readname`. However, this is cumbersome, and sometimes impossible. A way around this is do to a "double parsing" of the code using the `eval` command.

As the manual page states: "Eval takes one or more arguments, which together comprise a Tcl script containing one or more commands. Eval concatenates all its arguments . . . and passes the concatenated string to the Tcl interpreter recursively, and returns the result of that evaluation (or any error generated by it) . . .. In this case the eval command takes "$mode" and replaces it with it's current value ("readfile") before executing the button command, which achieves the desired result. **This will be a critical step when we start using the object-oriented extensions.**.

There is nothing really complex here, other than (i) recognizing that the callback function has variables whose values are either not going to be available or uncertain at the time of the execution and (ii) using the eval command to recursively parse the command (e.g. button) rather than parsing it directly.

6.1 Menubars and Menus

The main menu appears at the top of a window, from which "hang" submenus, each of which contains collections of buttons (both regular buttons, as well as checkbuttons and radiobuttons). Menus are very useful for making a large number of options available in a small amount of screen "real-estate". While there are many subtle issues here, a simple example will clarify things. We will modify the previous example to move the three buttons (Reset, Color and Exit) into a menu structure. The bottom part of

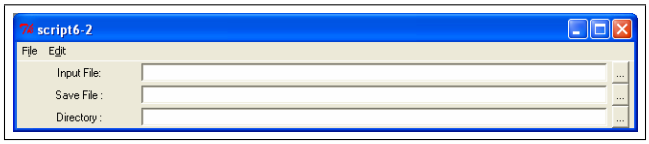


Figure 6.2: A snapshot of script6-2.tcl.

this new script (script6-2.tcl) is shown below:

```
# Create the Main Menubar and attach it to the main widget
set menubase [ menu .top ]; . configure -menu $menubase

# Create two submenus
set filem [ menu $menubase.file -tearoff 0 ]
set filee [ menu $menubase.edit -tearoff 1 ]

# Attach the submenus to the main menubar
$menubase add cascade -underline 0 -label "File" -menu $filem
$menubase add cascade -underline 0 -label "Edit" -menu $filee

# Add buttons to the file submenu
$filem add command -label "Color" -command { SetColor . }
$filem add separator
$filem add command -label "Exit" -command { exit }

# Add buttons to the edit submenu
$filee add command -label "Reset" -command { Reset 1 }
```

Creating a Menu: Both the main menu, and the submenus are created using the `menu` command. A key option of the menu command is `-tearoff` which determines whether the menu can become a standalone dialog if needed for easier access.

The main menu is attached to it's parent widget by specifying it as the option `-menu` of the widget, i.e. `.configure -menu $menubase`, where in this case '.' is the parent widget.

Adding Entries to a Menu: Items (entries) can be added to a menu using the `add` option. The syntax for adding a submenu is:

```
$parentmenu add cascade -label Submenuname -menu $childmenu
```

Similarly other kinds of entries can be added. The most common ones are:

- `command`– which adds a button, e.g.
  `$filem add command -label Color command  SetColor .`
- `separator` – which adds a line to space of items on the menu.
- `check` – which adds a checkbutton, e.g.
  `$filem add check -variable parameters(swap) -label Swap`
- `radio` – which adds a radiobutton, e.g.
  `$filem add radio -variable parameters(displaymode) -label ShowAsSurface`

If your program only has a few "button" like elements, then menus are probably not the best GUI-

element, in fact newer operating systems like Windows Vista seem to be going away from them. On the other hand a complex program with lots of options will probably benefit for having a menu setup.

An additional script (script6-3.tcl) illustrates the use of `add check`, I will not discuss it in any detail in the handout.

6.2 A Complete Application

The following example (script6-4.tcl) illustrates the use of the text widget as the heart of a simple text editor. The text widget can be used to store multiple lines of text (and has other advanced features).

The program consists of the following pieces:

- Initialize Parameters
- Create Scrolled Text Box
- Procedures for Manipulating the Text Box
- Procedures for Loading and Saving the contents of the Text Box
- A Main GUI Generation procedure.
- The main program itself

We examine them in turn, beginning with the initialize parameters function which is straightforward.

```
proc InitializeParameters { } {
  global parameters
    set parameters(enabletimestamp) 0
    set parameters(enableediting) 1
}
```

Creating the Text Widget: Then we create the text using the `text` command. We can also attach a scrollbar created using the `scrollbar` command. Note the use of `eval` to set the name of the scrolling callback.

```
proc CreateScrolledText { basewidget } {
    global parameters

    set w [ labelframe $basewidget.log -text "Text File"  ]
    pack $w -side top -fill both -expand t
    eval "text $w.log -width 80 -height 10  -borderwidth 2 -relief raised \
           -setgrid true  -yscrollcommand {$w.scroll set}"

    set cmd "scrollbar $w.scroll -command { $w.log yview} "
    eval $cmd
```

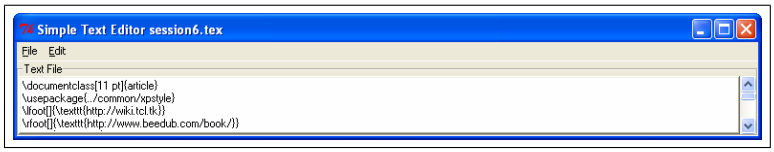


Figure 6.3: A snapshot of the simple text editor (script6-4.tcl)

```
    pack $w.scroll -side right -fill y
    pack $w.log -side left -fill both -expand true
    return $w.log
}
```

Manipulating the Text Widget: The following three procedures illustrating simple clearing the widget, adding text and setting its state. Unlike simpler widgets, there is no variable that automatically contains the contents of this widget, the widget needs to be queried for its contents. The key item here is the concept of index which has the format **line.character** and it starts at 1.0. The word "end" is shorthand for the last character in the file.

```
proc ClearAllText { } {
    global parameters
    set numlines [ expr int([ $parameters(textbox) index end ]) -1 ]
    if { $numlines > 1 } {
        set ok [  tk_messageBox -type yesno -default no -title \
            "Think again ..." -message "This will erase your current file?" \
               -icon question  ]
        if { $ok == "no" } {  return 0 }
    }
    # Delete from start (1.0) to end
    $parameters(textbox) delete 1.0 end
    return 1
}

proc AddANewLine { } {
    global parameters
    set line "Some More Text"
    if { $parameters(enabletimestamp) } {
        set currenttime [  clock format [ clock seconds ] -format "%H:%M:%S on %d %b %Y" ]
        set line "$line at $currenttime"
    }
    $parameters(textbox) insert end "$line\n"
    $parameters(textbox) see end
}

proc SetEditableState { } {
    global parameters
    if { $parameters(enableediting) == 1 } {
        $parameters(textbox) configure -state normal
    } else {
        $parameters(textbox) configure -state disabled
    }
}
```

Loading/Saving the Text: The next two procedures illustrate loading and saving the text in a "defensive programming" setup – where you assume that everything that can go wrong will. The `catch` command is a special tcl construct for handling exceptions (errors). Placing a command inside `catch { }` ensures that no system errors are given. Rather the program has a chance to gracefully respond

to the error (e.g. we have no permission to read or write to a specific file).

To get the number of lines, we explicitly ask for the value of the "end" index, truncate it to its integer part and subtract 1!

```
proc Load { } {
 global parameters

 set ok [ ClearAllText ]
 if { $ok == 0 } { return 0    }

 set fname [tk_getOpenFile -title "Load Filename" \
        -filetypes { { "Text Files" {.tex .txt } }} ]
 if { [ string length $fname ]  < 1 } { return 0   }

 set fileid 0;    catch { set fileid [open $fname r] }
 if { $fileid == 0 } {
     tk_messageBox -type ok  -title "Error ..." \
          -message "Cannot read file $fname" -icon error
     return
 }

 while { [ gets $fileid line ] >=0 } {
     $parameters(textbox) insert end "$line\n"
 }
 close $fileid
 wm title . "Simple Text Editor [ file tail $fname]"
}

proc Save { } {
 global parameters

 set numlines [ expr int([ $parameters(textbox) index end ]) -1 ]
 set fname [tk_getSaveFile -title "Save Filename" \
    -filetypes { { "Text Files" {.tex .txt } }} ]
 if { [ string length $fname ]  < 1 } { return    }

 set fileid 0;     catch { set fileid [open $fname w] }

 if { $fileid == 0 } {
     tk_messageBox -type ok -title "Error ..." \
        -message "Cannot save to file $fname" -icon error  ]
     return
 }

 for { set i 1 } { $i < $numlines } { incr i } {
     set ip [ expr $i + 1 ]
     set txt  [ $parameters(textbox) get $i.0 $ip.0 ]
     puts -nonewline $fileid  "$txt"
 }
 close $fileid
 wm title . "Simple Text Editor [ file tail $fname]"
}
```

Main GUI Generation: This is a fairly straight forward GUI generation procedure, with a menu at the top and a textbox at the bottom.

```
proc GenerateGUI { } {

    global parameters
    set menubase [ menu .menu ]
    . configure -menu $menubase

    frame .top; pack .top -side top -expand true -fill both
    set filem [ menu $menubase.file -tearoff 0 ]
    set filee [ menu $menubase.edit -tearoff 0 ]
    $menubase add cascade -underline 0 -label "File" -menu $filem
    $menubase add cascade -underline 0 -label "Edit" -menu $filee

    $filem add command -label "Clear Text" -command { ClearAllText  }
    $filem add separator
    $filem add command -label "Load" -command { Load  }
    $filem add command -label "Save" -command { Save  }
    $filem add separator
    $filem add command -label "Exit" -command { exit  }

    $filee add check   -label "Enable TimeStamp" -variable parameters(enabletimestamp)
    $filee add check   -label "Enable Editing"   -variable \
           parameters(enableediting) -command { SetEditableState }
    $filee add separator

    $filee add command -label "Add A New Line" -command { AddANewLine  }

    set parameters(textbox) [ CreateScrolledText .top ]
}
```

Main Program: Finally the main program itself, short and sweet.

```
InitializeParameters
GenerateGUI
wm title . "Simple Text Editor"
```

Assignment

- Naturally, read and understand the scripts referred to in this handout. Note that there are slight differences between the actual scripts and the versions presented in this handout, mainly to save space.
- Using script6-1.tcl as a base add a function to print the contents of the file whose filename is specified under Input File.
- Using script6-2.tcl modify the menu to add a Help Menu. Add a function under this labeled

"About" which gives (via a messagebox) some brief information about the application.

- Using script6.3.tcl modify the GUI to add a status line (an entry widget) at the bottom. Add a function to the edit menu called "Info" that gives information about the current file (e.g. number of lines, and anything else that you can get).

Chapter 7

Object Oriented Programming with [Incr] Tcl

Object-oriented programming (OOP) is a key concept. It represents a paradigm shift from procedural (structured programming). The key difference is that, while in procedural programming variables and procedures are distinct entities, in OOP the functionality (i.e. procedures) is embedded into the data structures (variables) themselves to create what are called classes. We will explore OOP using the [Incr] Tcl extension to Tcl. The webpage for the [Incr] Tcl project is at `http://incrtcl.sourceforge.net/itcl/`. Many new concepts in Tcl, including namespaces, began as part of [Incr] Tcl.

7.1 Introduction

In the evolution of programming some of the main milestones where (i) the transition to structured programming, (ii) the use of object-oriented programming (OOP) and more recently (iii) generic programming (we may get to this at the end of the semester). Each of this transitions was a result of the need to be able to write "bigger" programs with 'less' code. The Visualization Toolkit (VTK), which we will get to in a couple of weeks, is an object-oriented library and understanding how object-oriented libraries work and how to extend them is at the heart of this class. Rather than taking the more conventional approach and describing OOP using C++, I will instead take a detour and use the [Incr] Tcl OOP extension for the Tcl language instead to ease the transition.

Object Oriented Programming: In a Wikipedia article, OOP is distinguished from procedural programming by the short mnemonic that in OOP the verb is always attached to the object! For example consider the following statements:[1]

Subject-oriented: The Sales Application saves the Transaction vs *Object-oriented:* The Transaction saves itself upon receiving a message from the Sales Application

Subject-oriented: The Sales Application prints the Receipt vs *Object-oriented:* The Receipt prints itself upon receiving a message from the Sales Application

[1] In the discussion below, Subject-oriented=procedural for our purposes.

In OOP, instead of having 'simple' variables which store values and are operated on by external procedures, we have objects. Objects are 'active' variables which combine data storage and functionality. Hence objects have the ability to "do stuff" when instructed by the main program.

An example we will (perhaps over) use extensively in this chapter is the distinction between a four-month old infant and four year old child. Consider the all-important (once you have children you will appreciate this more!) concept of "going to the bathroom" (or more likely cleaning up the situation once this is completed!). For all intents and purposes the infant is, a mostly, passive participant. The action is on the part of the parent "Change the baby". On the other hand a four year old (when things are going well), can be told to "Go to the bathroom" and she has the capability to do so and take care of things. In procedural programming, variables are *infants* they can do nothing of themselves. Rather, the program (the adult) operates on them using different procedures. In OOP, variables are *children*, they have the ability to do things as prompted by the main program.

A key first concept is to distinguish between class and object. The class is the generic (e.g. infant) where as the object represents a concrete incarnation or instance of the class (e.g. Alex).

To stretch this metaphor a little bit, infants and four-year-olds have some properties in common (e.g. instances of both groups have a name). In programming terms, we can first design the simpler of the two classes (infant in this case) and then write the second class (fouryearold) as explicitly extending (or deriving from) infant. In this way some common functionality need only be implemented once, thus eliminating potential bugs in the code. Also if down the road we need to add functionality that is common to both (e.g. a social security number), this needs only to be added to the infant class and it will automatically be *inherited* by the *derived* fouryearold class.

Object hierarchies and their design are critical concepts. In programming with VTK one needs to understand how the object hierarchy in VTK is put together for tasks as simple as reading the documentation. Some of the functionality of classes such as Images is simply inherited from parent classes (e.g. DataObject) and one needs to read both "pages" to get a full handle on what the Image class can do and how to interact with it.

What is [Incr] Tcl [Incr] Tcl is probably the most commonly used object-oriented extension for Tcl. The Tcl core language is not object-oriented at its core (unlike say Java or Python), and OOP is added using extensions such as [Incr] Tcl, Other's include Xotcl and snit.

7.2 The Infant Class

A First Class Definition: A class consists of the following:

1. Member variables – these store data (e.g. myName)
2. Member functions – these are the procedures that are embedded in the class. Two special functions are:
   - **The constructor:** This method is automatically called when the object is created.
   - **The destructor:** This method is automatically called when the object is deleted.

An additional feature is that the methods can be defined in one of two ways. (i) Either explicitly in the class definition (we will use this for the constructor and destructor), or (ii) outside the main class definition (we will use this convention for everything else).

A final concept, before looking at a real example, is the distinction between public and protected (and private). Public variables/methods can be accessed from outside the class, where as protected variables/methods can only be accessed from inside the class. It is a good idea to make most variables protected and to provide methods for accessing their value and/or modifying them.

Without further ado, here an example class (infant) from the script file (infant.tcl). First there are the usual directives for package handling. The second line (package require Itcl) loads the OOP extensions.

```
package provide Infant 1.0
package require Itcl 3.2
```

Next comes the class definition itself. This is performed using the `itcl::class` command whose arguments are the name of the class (e.g. Infant) and the class definition.

```
itcl::class Infant {
    # Class Variables
    protected variable myWeight 2.0
    protected variable myName   "Anonymous"

    # Constructor and Destructor
    constructor { newname } {set myName   $newname  }
    destructor { set myName "";  set myWeight ""   }

    # Interface, public methods
    public method GetName { }
    public method GetWeight { }
    public method SetWeight { wgt }
    public method DailyRoutine { }
    public method PrintSelf { }

    # Protected methods
    protected method Cry { }
};
```

The final "};" ends the class definition. The definition consists of four parts:

- The member variables (myWeight and myName) both declared protected.
- The constructor and the destructor.
- The Interface defined in terms of the public methods. These are the methods that outsiders can invoke (e.g. the main program).
- The Protected methods which can only be invoked from within the class.

Finally we move to the implementation both the public and the private methods. This very similar to implementing procedures with two key exceptions: (i) use the syntax itcl::body Classname::methodname as opposed to proc methodname and (ii) all class member variables (e.g. myWeight and myName in this case) are explicitly available as if a "global"-like command had been executed. A final variable

"this" is also always defined, "this" is the name of the class instance or object itself. For example, the method Cry is invoked from within the DailyRoutine method using `$this Cry`.

```
::itcl::body Infant::GetName { } { return $myName }
::itcl::body Infant::GetWeight { } { return $myWeight }
:itcl::body Infant::SetWeight { wgt } { set myWeight $wgt }

::itcl::body Infant::DailyRoutine { } {    puts stderr "infant ... Daily Routine Start"
    $this Cry;     puts stderr "infant ... Daily Routine End"
}

::itcl::body Infant::Cry { } {    puts stderr "infant ... WaWa!"}

::itcl::body Infant::PrintSelf { } {
    puts stderr "infant ... myName =   $myName";
    puts stderr "infant ... myWeight = $myWeight"}
```

Interacting with a Class: The following script (script7-1.tcl) demonstrates the use of the infant class.

```
lappend auto_path [ file dirname [ info script ]]
package require Itcl 3.2; package require Infant

set leanboy [ Infant \#auto "A" ];
$leanboy SetWeight 7.0

set fatboy [ Infant \#auto "B" ];  $fatboy SetWeight 11.0

puts stderr "\nLet's see the details on $leanboy"
$leanboy PrintSelf; $leanboy DailyRoutine
puts stderr "\nLet's see the details on $fatboy"
$fatboy PrintSelf

itcl::delete object $leanboy; itcl::delete object $fatboy
```

The first thing that needs to happen in order to leverage a class is to create a specific instance of the class. (The generic concept 'infant' does not cry, it is only real infants that do!). This is accomplished by calling the constructor as follows:

```
set leanboy [ Infant \#auto "A" ];
```

The constructor is invoked using "Infant" command (which is the name of the class). The constructor takes an implicit argument which is the name of the object (in C/C++ terms this is the address of the pointer). While one can specify this manually, it is best to let the system generate an automatic pointer

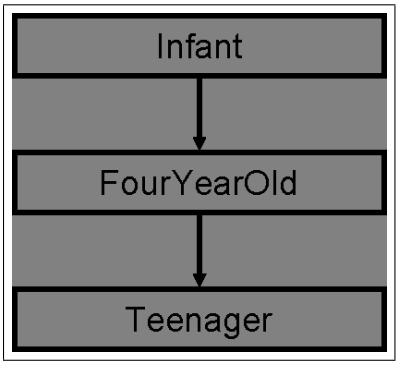


Figure 7.1: An example class hierarchy. The classes get progressively more complex with new functionality being added at each level.

name using the "#auto" construct. The rest of the arguments are those specified in the constructor definition itself, in this case there is only one (newname) which is the (human) name of the infant. In this case we will call this "baby A". The result is stored in the variable "leanboy" which can be used to access the class.

"Baby A" can be made to do all the things defined by its public methods (e.g. GetName, GetWeight, SetWeight, DailyRoutine and PrintSelf). One thing we can do (following script7-1.tcl) is to set its weight, using:

```
$leanboy SetWeight 11.0
```

We can also tell leanboy to print itself using `$leanboy PrintSelf` or to go through his daily routine `$leanboy DailyRoutine`. We can also instantiate a second baby (fatboy) and interact with it similarly.

When an instance of a class is no longer useful, it can be removed from memory, by calling it's destructor. This is invoked using the itcl::delete object command which takes one argument, the (pointer) name of the instance.

7.3 Inheritance Trees

Consider the case where you have some code (either set of procedures or a class) that is almost what you need but needs a slight tweak or some additional functionality. In procedural code doing this would involve: (i) copying the code, (ii) renaming the functions and (iii) change the piece(s) that needs adaptation. A downside of this method is that should you need to add functionality that is common to both the original and the copy, this needs to added twice (in each piece).

The same task in an Object Oriented Programming environment reads: (i) derive a new (child) class from existing class. (we will call this the parent class) (ii) Add new methods and override the methods that needs changing. Crucially if new code, that is common to both classes, needs to be added later, it can simply be added to the parent class and the child class automatically inherits it.

The "inherit" statement: At the top of the class definition there is the usual `class` command and then the key statement `inherit` which specifies that the class `FourYearOld` is derived from the class `Infant`.

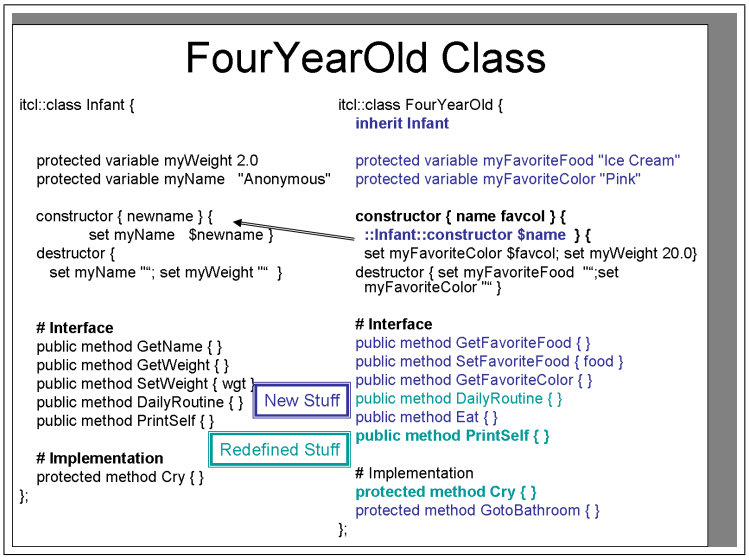


Figure 7.2: A comparison of "infant" and "fouryearold".

```
itcl::class FourYearOld {
    inherit Infant
```

Constructor and Destructor: Following this, there are some new member variable definitions (myFavoriteFood, myFavoriteColor) and then the next key step is the definition of the "nested" constructor as follows:

```
constructor { name favcol } {
       ::Infant::constructor $name
} {
       set myFavoriteColor  $favcol
       set myWeight 20.0
}
```

The constructor for `FourYearOld` first calls the constructor for `Infant` and passes the name argument to it. Then the constructor for Infant itself is executed. The destructor is also "implicitly nested" in the opposite order. First the explicit destructor for the `FourYearOld` class is called followed by the destructor for the `Infant` class.

New Methods: The following methods are new: `GetFavoriteFood`,`SetFavoriteFood`, `GetFavoriteColor`, `Eat` and `GotoBathroom`. These represent functionality that is absent in the `Infant` class (in a 'developmental' interpretation, these are new skills that develop later in life!).

Redefined or Overridden Methods: The methods `DailyRoutine`, `PrintSelf` and `Cry` exist in both `Infant` and `FourYearOld` but the versions in the derived class either redefine or amplify the functionality of the original. Consider the definitions or `Cry` and `PrintSelf` respectively given below. The 'new' `Cry` method simply redefines the 'old' `Cry` method.

```
::itcl::body FourYearOld::Cry { } {
    puts stderr "fouryearold ... Mom I would like to watch a movie!"
}
```

The 'new' PrintSelf method on the other hand explicitly amplifies the 'old' one and it explicitly calls it to do half the work. This is accomplished by the invocation of ::Infant::PrintSelf which prints the values of myName and myWeight. Then the 'new' method adds the information that is new to FourYearOld but not present in infant. In this way if down the road a new member variable (e.g. myHeight) is added to the Infant class, it will automatically printed for an instance of FourYearOld without needing to modify the later's PrintSelf method directly.

```
::itcl::body FourYearOld::PrintSelf { } {
    ::Infant::PrintSelf
    puts stderr "fouryearold ... myFavoriteFood  =   $myFavoriteFood"
    puts stderr "fouryearold ... myFavoriteColor =   $myFavoriteColor"
}
```

The example script below (script7-2.tcl) illustrates the use of the FourYearOld class. It is similar to the previous example script (script7-1.tcl) and will not be described in detail here.

```
}
lappend auto_path [ file dirname [ info script ]]
package require FourYearOld

set girl [ FourYearOld \#auto "M" "Pink" ]
$girl SetWeight 35.0
$girl SetFavoriteFood "PB & J"
puts stderr "\nLet's see the details"
$girl PrintSelf

puts stderr "\nHere is [ $girl GetName]'s daily routine"
$girl DailyRoutine

itcl::delete object $girl
```

A third class in the hierarchy Teenager is also defined which adds additional functionality. The code is in the file teenager.tcl and an example invocation is in the example script7-3.tcl. We will not discuss them in any detail here. (In any event given my general lack of experience with teenagers I would rather not comment on the matter too much at this stage, I may have to revisit this nine years from now!)

7.4 Static/Common Methods and Variables

Often there is a need for functions or variables that are tightly associated with a class. Such helper procedures are often needed by code in the class definition but do not need an "active" instance of the class for them to function. To revisit a previous statement, while the generic concept 'infant' does not cry, but only real infants do, there are such things that are properties of the generic class infant as opposed to individual infants. One such function would be the statistic "How many infants are there?".

While such functionality could be implemented in separate functions/variables, a more elegant way is to implement them within the class definition as explicit properties of the collective. Such methods/variables are called either static (C++ terminology) or common ([Incr] Tcl) terminology.

Consider the example below (script7-4.tcl). To simplify, both the class definition/implementation and the main code are in the same file. At the top we have a variant of the `Infant` class called `IrsInfant` (IRS=Internal Revenue Service) which has an explicitly social security number (mySSN) field. A challenge is to make sure that all instances of `IrsInfant` have unique SSN's (otherwise this would mess-up the tax collection). This is accomplished by maintaining a "common" counter in the class (`NumInfants`) as well as an explicit "common" method (`NewSSN`) for generating a new social security number.

The first part of the class definition is unremarkable and follows closely the original `Infant` definition with some omissions to keep the code short.

```
package require Itcl 3.2

itcl::class IrsInfant {

    protected variable myName   "Anonymous"
    protected variable mySSN    "0"

    constructor { newname } {
set myName   $newname
set mySSN [ ::IrsInfant::NewSSN ]
    }

    destructor {
set myName ""
set mySSN ""
    }

    # Interface
    public method GetName { }
    public method GetSSN { }
    public method PrintSelf { }
```

The common/static methods and variables are defined next. The variables are declared 'common' using the `common` directive in place of the `variable` directive. The methods are declared common using the `proc` directive instead of the `method` directive, as is shown below. Note that the constructor above calls the static function `NewSSN` using the explicit designation `::IrsInfant::NewSSN`. In some respects common/static variables and procedures use the class as an implicit namespace to cut down on name pollution.

```
    # Common
    protected common NumInfants 0
    protected proc NewSSN { }
    public    proc GetNumInfants { }
};
```

The implementation of the non-static methods is very simple:

```
::itcl::body IrsInfant::GetName { } { return $myName }
::itcl::body IrsInfant::GetSSN { } {  return $mySSN  }
::itcl::body IrsInfant::PrintSelf { } {
    puts stderr "myName =   $myName"; puts stderr "mySSN = $mySSN\n" }
```

Finally the static/common methods are implemented. The assigned SSN is equal to 1000 plus the number of infants; the number of infants is incremented each time a request for a NewSSN is issued. The public common method `GetNumInfants` simply returns the current value of the `NumInfants` variable.

```
::itcl::body IrsInfant::NewSSN { } {
    incr NumInfants; return [ expr 1000+ $NumInfants ]
}
itcl::body IrsInfant::GetNumInfants { } {   return  $NumInfants }
```

The main script (below) simply allocates 5 new infants with names C1,...,C5 and prints their description.

```
for { set i 1 } { $i <=5 } { incr i } {
    set baby($i) [ IrsInfant \#auto "C$i" ]
    $baby($i) PrintSelf
}
puts stderr "The Total Number Of Infants = [ ::IrsInfant::GetNumInfants ]"
```

7.5 Additional Notes

Object Hierarchies can also come in tree like structures where two classes can have the same parent class but each is going their separate ways (e.g. my brother is in banking although we share common genes, we also have properties that are different – e.g. salary!)

There is also the possibility of 'mixed' inheritance, i.e. a class can have two parents. I suggest you avoid this, it makes things really messy in my opinion.

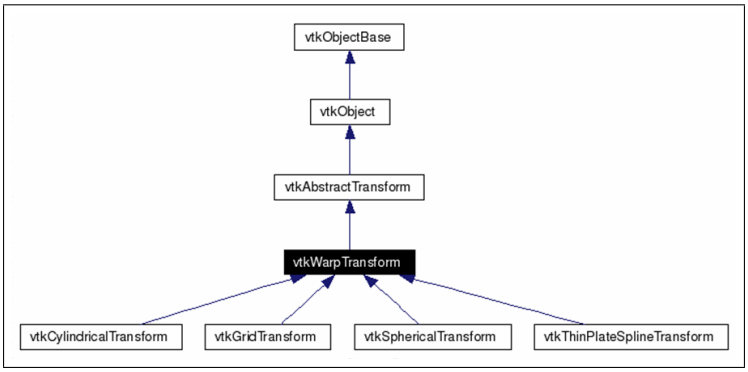


Figure 7.3: An more complex class hierarchy (VTK 4.4 transformation classes).

Assignment

- Make sure you understand the following concepts:
  1. Object Hierarchy
  2. Inheritance
  3. Overriding
  4. Static/Common
  5. Instance vs Class
  6. Constructors and Destructors,
  7. Protected vs Public (read also about protected vs private)

  In particular write a two sentence description of each. Place your answers in a text file and upload this in the repository in the usual way.

- Download the .pdf file itclitk.pdf. This contains two chapters by Michael McLennan who is the original author of [Incr] Tcl. Read chapter 1. The namespace material (pages 47-63) is outdated and to some extend confusing. You can also skip (if you don't have time) anything past page 37. Chapter 2 is interesting but can be skipped for now.

- Taking script7-2.tcl as a base, create an additional fouryearold object and set it's favourite food to something and also it's weight. Use PrintSelf to verify that the properties have been set.

- Merge some of the changes from IrsInfant into Infant (i.e. the social security number stuff) and re-run script7-3.tcl to observe whether these changes also apply to an instance of the Teenager class.

- Create an additional class (collegestudent) derived from Teenager.
  1. Add at least one additional member variable with appropriate Get/Set Methods for modifying and obtaining it's value. Appropriately modify the PrintSelf method to print this also.
  2. In addition change the GotoSchool method to only print "I am going to school" if the current day is a Monday through a Friday. You can get the current date using
     `set a [ clock seconds ]; set d [ clock format $a -format %w ]`
     where the value of d is the weekday as a number (Sunday = 0, Saturday = 6).
  3. Modify pkgIndex.tcl to reflect the presence of the new class and write a script similar to script7-3.tcl to exercise the new class.

Chapter 8

Iwidgets: Object Oriented GUIs

Object oriented libraries are key tools in large programming tasks. Beginning next week we will explore the Visualization Toolkit with its multifaceted functionality for image processing and visualization. This week we explore a smaller but extremely useful library – the Iwidgets toolkit. This provides a number of so-called megawidgets – complex combinations of basic widgets packaged as [Incr] Tcl classes which enable the quick and easy construction of complex graphical user interfaces. Libraries such as Iwidgets form the backbone of most large applications. The iwidgets.pdf file in the repository contains an excellent introduction to the topic, this handout is meant to supplement this material and provide a roadmap.

8.1 Introduction

The Iwidgets library leverages the [Incr] Tcl object oriented extensions to the basic Tcl language. Before launching into a full scale description of Iwidgets, there is one additional concept from [Incr] Tcl that needs to be introduced, namely the use of `cget` and `configure` to read and modify public member variables respectively. Consider the following script (script8-1.tcl) which defines the class PubInfant and invokes it. Note that PubInfant has two member variables, one protected (myName) and one public (myWeight). To get the value of myName we need to invoke the explicitly defined GetName member function. Since, myWeight, is a public variable it can be directly access using the "configure" method to set its value and the "cget" method to get it's value, as illustrated in the script.

```
package provide Infant 1.0
package require Itcl 3.2

itcl::class PubInfant {
    public    variable myWeight 2.0
    protected variable myName   "Anonymous"
    constructor { newname } { set myName   $newname }
    destructor { set myName ""; set myWeight ""   }
    public method GetName { } { return $myName }
};
# -------------------------- Main Code ---------------
set leanboy [ PubInfant \#auto "Alex" ]
$leanboy configure -myWeight 8.0
```

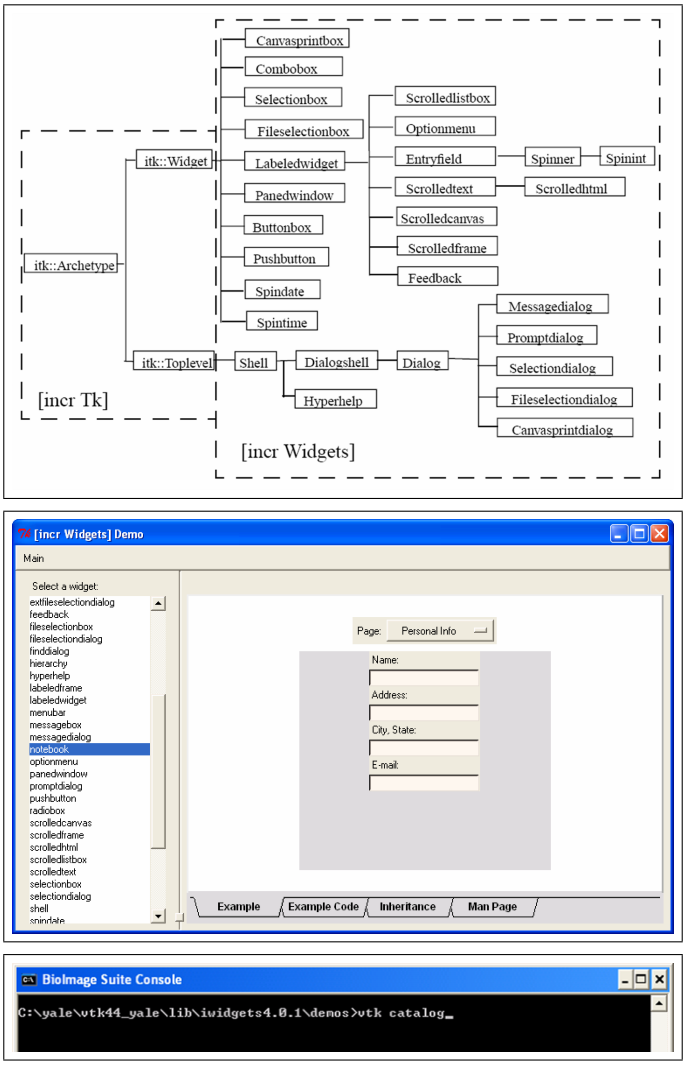


Figure 8.1: *Top* The Iwidgets class hierarchy. *Middle:* The demo program which can be used as a tutorial for all Iwidgets, and *Bottom:* Invoking the demo.

```
puts stderr "The weight of [ $leanboy GetName ] is [ $leanboy cget -myWeight ]"
```

Note also that unlike previous examples, in this case, the code for GetName is placed right in it's definition as opposed to having a separate code block for it. This is often desirable for short methods.

8.2 Introducing Iwidgets

Iwidgets is a collection of so-called mega-widgets. These are combinations of Tk widgets that are used as building blocks to enable the creation of more complex graphical user interfaces. Consider the example of the need for the combination of a label and an entry widget, which is a fairly frequent occurrence (this is illustrated in iwidgets.pdf). In ordinary Tk, this would require three widgets, (i) a frame, (ii) a label and (iii) an entry. Iwidgets provides a single mega-widget called entryfield which presents precisely this combination and some additional features as well.

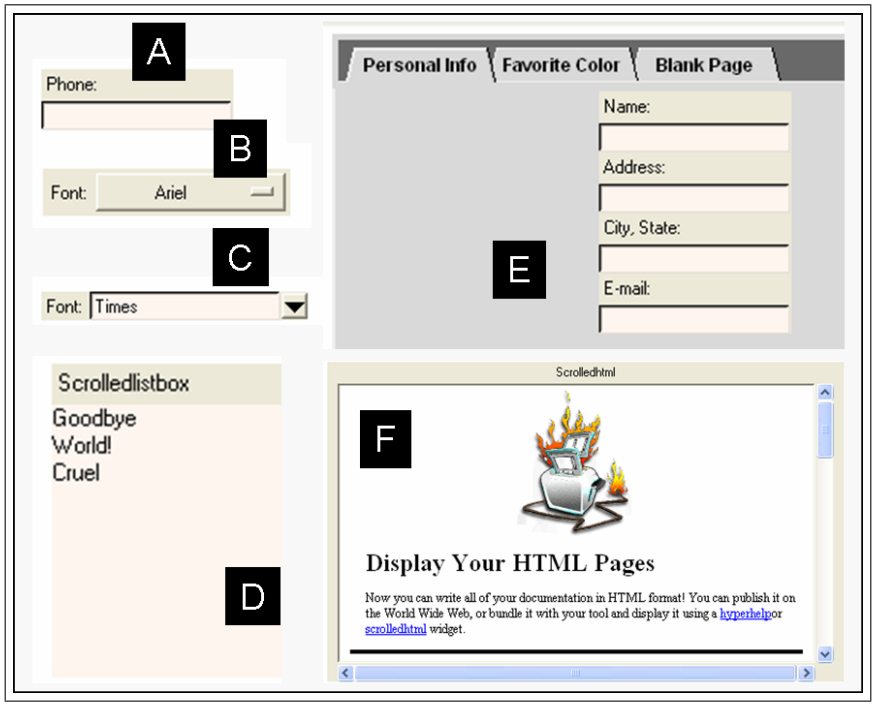


Figure 8.2: Some of the many widgets in Iwidgets.

Figure 8.1(top) presents the hierarchy for the Iwidgets library. A critical skill in Object Oriented Programming and especially when leveraging outside code is the ability to read such hierarchy diagrams. For example, consider the case of the class Entryfield. This is a child of LabeledWidget which in turn is a child of itk::Widget which is finally in turn a child of itk::Archetype. Typically most class libraries have a single (or a very small number) of toplevel parent(s) (e.g. itk::Archetype) which defines functionality common to all classes. Then, progressively functionality is both specialized (via overriding) or added to create new classes. Note also that the parent classes (e.g. itk::Archetype) are often never meant to be instantiated (these in C++ are called abstract classes) but are simply present to encapsulate common functionality by classes further down the hierarchy.

Figure 8.1(middle) shows the "catalog" demo that comes with the Iwidgets package. This can be used to illustrate the individual widgets. On the left the user selects the widget that he/she is interested in. In the right pane, there is a four tab notebook. The four tabs display: (i) the example (ii) the example code (iii) the inheritance tree for the widget and (iv) the detailed manual page description. This is a great learning tool for the Iwidgets package. The bottom part of Figure 8.1 shows how to invoke the catalog demo in Windows. On linux, the path is different, simply change the path to `/usr/local/vtk44_yale/lib/iwidgets4.0.1/demos`.

8.3 A few key Iwidgets

There are lots of mega-widgets in Iwidgets. The following are particularly useful (at least I end up using them a fair amount). For example code for these widgets look at the **catalog** demo described previously. I suggest you read this section with the catalog demo program open.

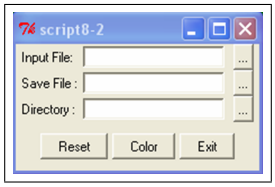


Figure 8.3: A snapshot of script8-2.tcl.

The entryfield widget: This widget combines a label and an entry field together. It also has functionality for "validating" the input in the entry field, such as making sure that it is an integer, a real-valued number, or a string. See Figure 8.2A.

The optionmenu widget: This is essentially a combination of a label and a tk_OptionMenu widget. See Figure 8.2B.

The combobox widget: This is a drop-down list control that again enables the selection of an element from a list. See Figure 8.2C.

The scrolledlistbox widget: This allows the selection of an element (or group of elements) from a list. See Figure 8.2D.

The tabnotebook widget: This is used by the catalog demo itself. It allows for the creation of complex graphical user interfaces using "tabs" in the window. See Figure 8.2E.

The scrolledhtml widget: This is a widget in which you can display HTML formatted text, it could be used to construct a mini web-browser. See Figure 8.2F.

8.4 Object Oriented GUIs

The following example (script8-2.tcl) is an object oriented version of a previous example (script6.1.tcl). Functionally, with the exception of the nice alignment of the entryfields, this is equivalent to script6-1.tcl. However, the encapsulation of this graphical user interface into a class, allows for more than one instance of the class to be created in a program in a clean manner.

The first part of the script invokes both the Itcl and Iwidgets extensions in the usual manner.

```
package require Itcl
package require Iwidgets
```

Key Trick: Next comes the class header. The only novel aspect of this is the declaration of the parameters variable (which will turn out to be a two-dimensional associative array) as "private common". If you go back to script6-2.tcl you will notice that, in that case, a global array called parameters was used to store all

variables that were attached to widgets. In the case of [Incr] Tcl based programming, we use a similar concept. However, this array can not be explicitly a member variable of the class, other than as a static/common variable, because linking a variable to a widget requires that it has a static address. Hence the solution is to use a static associative array attached to the class. Since, we frequently would like to have more than one instance of the class active in a program, the trick here is to have two-dimensional associative array of the form: `parameters(objectidentifier,variable)` where objectidentifier points to the individual instance of the class. This is most easily done using the implicitly defined "this" variable. Note that the constructor simply takes one argument (parent) which is the parent widget and calls CreateGUI which creates the graphical user interface.

```
itcl::class myGUI {
    private common parameters
    protected variable basewidget 0

    constructor { parent } { CreateGUI $parent }
    destructor { }

    public method Reset { ask }
    public method SetColor { widget }
    public method GetName { mode }
    public method CreateGUI { base }
};
```

The next threee methods are practically identical to their non OOP-counterparts in script6-1.tcl with two exceptions: (i) There is no global statement at the top of each procedure since the parameters array, as it is a class variable, is always available (ii) as mentioned before, the parameters array is now a two-dimensional associative array.

```
itcl::body myGUI::Reset { ask } {
    if { $ask == 1 } {
       set ok [  tk_messageBox -type yesno  -message "Reseting filenames" -icon question  ]
       if { $ok == "no"  } {  return }
    }
    set parameters($this,readname)  ""
    set parameters($this,writename) ""
    set parameters($this,directory) ""
}

itcl::body myGUI::SetColor { widget  } {
    set color  [ tk_chooseColor -title "Set Background Color"  -parent $basewidget ]
    if { [ string length $color ] > 0 } { $widget configure -bg $color  }
}

itcl::body myGUI::GetName { mode } {

    set fname ""
    set filetype1 [ list "Text Files" [ list .txt .tex ] ]
    set filetype2 [ list "All Files" "*"  ]

    switch -exact $mode {
```

```
        "readname" { set fname [tk_getOpenFile -title "Load" \
               -filetypes [ list $filetype1 $filetype2 ] ] }
        "writename" { set fname [tk_getSaveFile -title "Filename" \
               -filetypes [ list $filetype1 ] ]}
         "directory" { set fname  [tk_chooseDirectory -title "Select Directory" ] }
    }
    if { [ string length $fname ] < 1 } { return   }
    set parameters($this,$mode) $fname
}
```

Finally, here is the CreateGUI method. This replaces part of the "main program" in script6-1.tcl.

```
itcl::body myGUI::CreateGUI { base } {
    set basewidget $base
    $this Reset 0
    set labellist ""

    for { set i 1 } { $i <= 3 } { incr i } {
        set parent [ frame $basewidget.$i ]
        pack $parent -side top -expand false -fill x
        switch -exact $i {
            "1" { set name "Input File:"; set mode readname }
            "2" { set name "Save File :"  set mode writename }
            "3" { set name "Directory :" set mode directory  }
        }
        eval "iwidgets::entryfield $parent.1 -labeltext \"$name\"  -width 20  \
           -textvariable [ itcl::scope parameters($this,$mode) ] -relief sunken"
        lappend labellist $parent.1
        eval "button $parent.3 -text \"...\" -command { $this GetName $mode }"
        pack $parent.3 -side right -expand false -padx 5
        pack $parent.1 -side left  -expand true -fill x -padx 2
    }

    eval "iwidgets::Labeledwidget::alignlabels $labellist"

    set w [ frame $basewidget.bottom ]
    pack $w -side bottom -expand true -fill x -pady 10 -padx 20

    eval "button $w.1 -text Reset -command { $this Reset 1 }"
    eval "button $w.2 -text Color -command { $this SetColor $basewidget }"
    button $w.3 -text Exit -command { exit }
    pack  $w.1 $w.2 $w.3 -side left -expand true -fill x -padx 2
}
```

There are three elements here that need explanation.

1. *Attaching a class variable to a widget:* The construct
   `-textvariable [ itcl::scope parameters(`$this$`,mode) ]`
   takes a class associative static array "parameters($this$,mode)" and converts it to global scope using the

itcl::scope command. The output of itcl::scope is then used to set the -textvariable field of the widget. This construct is needed everytime a class member variable is attached to a widget. Note also that the variable **must be** static/common. In addition, since the $this variable is not defined in global scope, using **eval** to pre-parse this command is necessary.

2. *Using the iwidgets entryfield class*. This is simply invoked using iwidgets::entryfield. Nothing complex here, other than the fact that it takes arguments related to both the "label" and the "entry" widgets it contains.
3. *Using alignlabels:* Labelwidgets such as the entryfield widget can be aligned so that they look nice using the static class procedure "Labeledwidget::alignlabels". We create a list of the labels to be aligned and use the *eval* command to convert this list into separate arguments.

Finally, the main program. We create and pack a frame (.gui) and then create an instance of the myGUI class into it.

```
}
frame .gui
pack .gui -side top -expand true -fill both
set element [ myGUI \#auto .gui ]
```

Assignment

1. Download the .pdf file iwidgets.pdf. This contains a chapter by Michael McLennan who is the original author of [Incr] Tcl/Iwidgets. This is the best introduction to Iwidgets anywhere.
2. Spend sometime with the catalog demo. This is another key tool in understanding Iwidgets.
3. Using script8-2.tcl as a base, modify it so that two myGUI objects are instantiated. Place these next to one another.
4. Change the last script so that each instance of myGUI is in a separate dialog box.
5. Using script6-2.tcl as a base, rewrite it such that it becomes a class (similar to the conversion from script6-1.tcl to script8-2.tcl).
6. Create some text and save it in HTML format. Write a short script that invokes the scrolledhtml widget to display this text.

Part III

The Visualization Toolkit I – Using Tcl

Chapter 9

An Introduction to the Visualization Toolkit

The Visualization Toolkit (VTK) has become one of the *de facto* standard tools in modern programming for image analysis. It has rich functionality for both image/surface processing and visualization. It's design pioneered the use of the combination of scripting and compiled languages in a single piece of software through the use of wrapping techniques for making available the underlying C++ classes in VTK as new commands in scripting languages such as Tcl and Python. The keys to successfully using VTK are (i) understanding the structure of the object-oriented hierarchy and (ii) understanding it's pipeline architecture. This document and those following it will focus on VTK version 4.4.2. Much of this material is also applicable to the newly released version 5.0.

9.1 Introduction to 3D Graphics

Most students' first experience to image display consists of simply displaying a two-dimensional image slice (most likely in Matlab). While there are some issues, such as the relative orientation (row/column vs column/row) and the position of the origin, it is for the most part a fairly straightforward and intuitive procedure. The same applies to rendering curves and point landmarks on the image. The key behind this simplicity is the fact that, unsurprisingly, two-dimensional computer monitors are well suite to displaying two-dimensional content!

The move to 3D graphics and visualization requires, in some way, getting around the fact that for the most part our display units are two-dimensional. The most common "illusion" used in 3D graphics – and the one that is employed by the Visualization Toolkit (VTK) – is that the program first generates a 3D world consisting of three-dimensional entities – actors/props in VTK parlance – which have various appearance properties. The world is illuminated by a set of lights and the "user" looks at the world through the eyes of a virtual camera. The image that gets displayed on the computer monitor is precisely the output of such a virtual camera. To recap, a world consists of (i) Actors, (ii) Lights and (iii) a camera. The exact output naturally depends critically, in addition to the Actors themselves, on the position and orientation of both the lights and the cameras. If a camera is looking away from an actor, that actor will not be visible in the displayed "image".

9.2 An Introduction to VTK

VTK is a large, complicated, powerful but often surprisingly easy to use toolkit for 3D image processing and visualization. It is an object-oriented library. Many operations in VTK are performed using a pipeline architecture where multiple elements are attached together to perform a complex task. A typical pipeline takes the form shown in Figure 9.1. This is broken into two parts. The first part (shown in Figure 9.1(left) consists of:

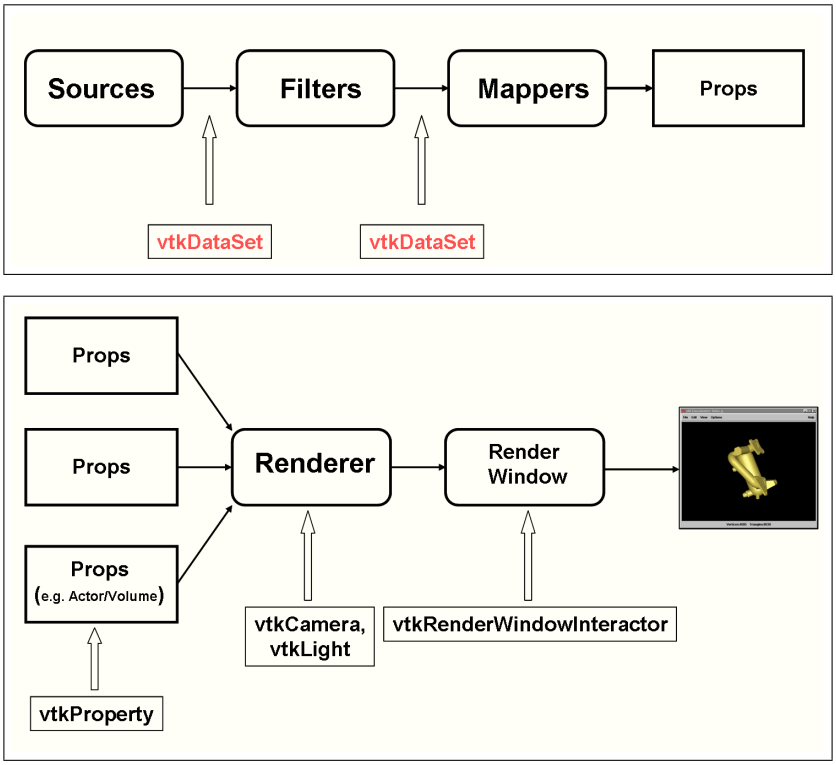


Figure 9.1: The VTK Pipeline Parts 1 and 2.

1. *Sources* – these classes produce data. For example, vtkJPEGReader reads a jpeg image from a file and generates an image output.
2. *Filters* – these operate on some data to produce a modified version. For example, vtkImageGaussianSmooth acts on an image to perform Gaussian smoothing and to produce a new smoothed image.
3. *Mappers* – these define the interface between data (e.g. images) and graphics primitives or software rendering techniques. A special kind of "mapper" like class are the writers which output the data to files (e.g. vtkJPEGWriter). Multiple mappers may share the same input, but render it in different ways.

The second part of the pipeline consists of the elements that make up the virtual 3D world, namely:

1. *Props/Actors* – these take as input the output of a mapper and 'know' how to generate the visible representation of data. The type of rendering produced is governed by an auxiliary data structure known as a property (e.g. color, showing a surface as a wire-frame vs a full surface etc.). Props take as their input the output of a mapper. Mappers *should not* be shared among props.

   Volumes – these are special kinds of props that are used to display volume rendered images.

2. *Renderer* – Renderers are the classes that generate a 2D image from a 3D scene. They have attached actors as well as lights and cameras. More formally, from the man page, "renderer is an object that controls the rendering process for objects. Rendering is the process of converting geometry, a specification for lights, and a camera view into an image. vtkRenderer also performs coordinate transformation between world coordinates, view coordinates (the computer graphics rendering coordinate system), and display coordinates (the actual screen coordinates on the display device). Certain advanced rendering features such as two-sided lighting can also be controlled."

3. *Render Window* – the Render Window is the piece of screen real estate in which the virtual camera image is displayed. An important auxiliary item attached to a render window is an interactor which handles mouse/keyboard input to the window.

9.3 Data Objects

VTK stores key items such as images, surfaces and meshes in classes derived from vtkDataObject. Data objects consist of combinations of points (nodes) which define the location of the data and cells (elements) which define the topology. For example a 20x20 2D image can be thought off consisting of 400 regularly spaced points connected into squares. Similarly, triangulated surfaces consist of a set of points and a set of interconnecting triangles.

Consider the examples shown in Figure 9.2. The first is a polygonal curve, which consists of five points – whose locations define the position of the curve. The topology (cells) are the five lines connecting the points. In this case they would take the form $(0,1),(1,2),(2,3),(3,4),(4,0)$ where $(0,1)$ is the line connecting points 0 and 1. The second example is that of a triangulated surface. This is similar to the curve, with the replacement of "line" cells with "triangle cells".

The case of the image is more interesting in that, in this case, both the topology and location of the individual points are implicitly defined once the position of the corner of the image (the origin) and the dimensions of the individual element (pixel or voxel spacing) are defined. Hence in the definition of an image (vtkImageData), the only things that need to be defined in terms of geometry and topology are the dimensions (number of points), the origin (position of the corner node) and spacing (the distance between the nodes), the rest are automatically defined from these.

For many data objects, especially images there is an additional critical element – the data associated with each point (or cell in some cases). Such data is used to store information such as the image intensity etc. Two data structures attached to each DataObject are used to store such information. These are the PointData and CellData structures which contain in turn data arrays derived from vtkDataArray. In the case of surfaces, additional information may include surface normals in addition to surface color etc.

9.4 Data Arrays

Ultimately most data, in VTK, is stored in Data Arrays. The parent class vtkDataArray (see hierarchy in figure 9.3) provides a generic interface to all arrays which are specialized forms of it for different data types. The common data objects simply assume that their data is stored in a vtkDataArray and rely only on the generic interface. The actual data is stored in derived classes of vtkDataArray (e.g. vtkFloatArray) which are specialized for the specific data type. In this way, one can have image data structures that are independent of the data type, which is dynamically defined during the runtime of the program.

A vtkDataArray is essentially a two dimensional array, the first dimension (the row dimension) is the *Tuple* dimension and second dimension (the column dimension) is the *Component*. dimension. In the case of vtkDataObjects, the contents of each tuple (row) are used to specify the data associated with each point (or cell in CellData). For example a 16×16 color image (RGB) would have its intensity information in a vtkDataArray consisting of 256 tuples, with 3 components per tuple. In this way the color information for the fourth voxel (in raster scan order) is in the fourth tuple of the array (indexing begins at zero in VTK, hence this will have index 3) with the red color stored in component 0, the green in component 1 and the blue in component 2. In fact, the very name component originated in the need to store color images, vtkDataArrays prior to version 4 could only have a maximum of 4 components. This has been since relaxed, and we can use the components to store the time component of a 4D Image (in the case of fMRI, this can be a number greater than 100).

9.5 Concluding Remarks

In this chapter, I have tried to focus on the fundamental components of VTK; understanding these is key to learning how to use the toolkit. In the next session we will begin to use VTK for surface and image rendering and manipulation. In lieu of an assignment, take a look at the VTK man pages at `http://noodle.med.yale.edu/vtk/`.

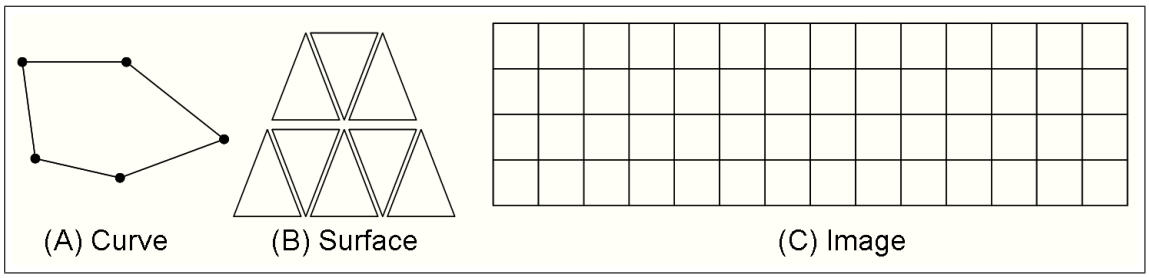


Figure 9.2: Geometry and Topology of Common Data Objects.

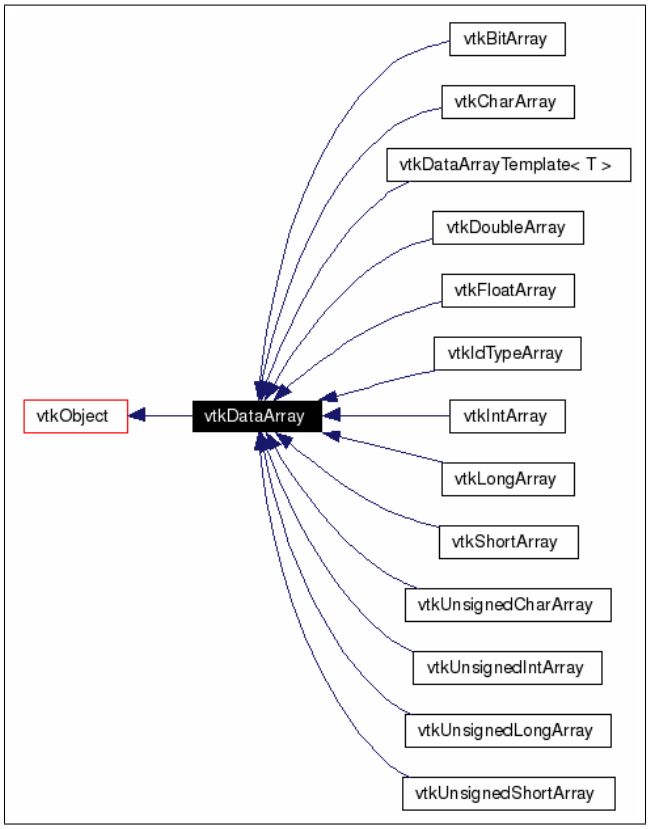


Figure 9.3: The Data Array Hierarchy.

Chapter 10

Curves and Surfaces in VTK

The Visualization Toolkit (VTK) has become one of the *de facto* standard tools in modern programming for image analysis. It has rich functionality for both image/surface processing and visualization. In this Chapter we describe creating and visualizing surfaces and space curves. This document and those following it will focus on VTK version 4.4.2. Much of this material is also applicable to the newly released version 5.0.

10.1 Introduction

The standard introduction to VTK tends to have the following pattern: (i) create a simple object, e.g. a cone[1] (ii) display it and (iii) manipulate the display in some way. While this approach has some merit, it is more directed towards users who are simply interested in visualizing existing data (i.e. read in from a file) as opposed to more common tasks in image analysis in which we need to somehow create this data – perhaps as the output of some segmentation algorithm.

While visualization remains a critical task, in image analysis, we are often less interested in "cosmetic" effects such as lights, perspective etc. and more at getting a handle of how a specific algorithm is performing. Hence for most of these lectures, we will focus on more "boring" visualization tasks. In any event, once one is comfortable with VTK, "Hollywood" like effects of high quality are not that hard to generate.

10.2 Tcl and VTK Objects

A common confusion when programming in VTK using the Tcl scripting language is the concept of object name. A new object (e.g. an image) is generated using the syntax (remember to use the `vtk` interpreter from here on):

```
vtkImageData myimage
```

where "vtkImageData" is the class type and "myimage" is the object name. Later code can modify this object using its name explicitly, in standard OOP usage, e.g. the example below changes "myimage" to have dimensions 100×100, sets its storage type to float and allocates memory.

[1] In the repository, I copy the files from one of the tutorials included with VTK, this is the Cone series and consists of six scripts, `Cone.tcl`,`Cone2.tcl`,...,`Cone6.tcl`. These are worth looking through.

```
myimage SetDimensions 100 100 1
myimage SetScalarTypeToFloat
myimage AllocateScalars
```

The object name "myimage" must be *absolutely unique* in the whole program. If an object of the name "myimage" exists, then any attempt to create a second object with the same name will fail. The object name is, in effect, a super-global variable[2] and while explicitly naming objects in this way works for small scripts, it is a recipe for disaster in *large programs*, especially when leveraging code written by other people.

The solution to this problem is to have a function that generates unique names (like the #auto construct in [Incr] Tcl) and using this to set the object name. Such a function could be similar to that used to generate unique SSN's in the `IrsInfant` class (Chapter 7).

In BioImage Suite we have a package (pxvtable) to perform this task. [3]. A slightly modified version of this code (newname.tcl) has the form:

```
package provide newname 1.0
namespace eval newname {
  variable counter
  set counter 0
  proc vnewobj { } {
    variable counter
    incr counter
    return "myvtkobj$counter"
  }
}
```

While inelegant, the only modification this code needs for use in your own code is to replace myvtkobj with some unique indentifier (using your initials might help). Naturally if you are using BioImage Suite as a base for your own work, you can always invoke the BioImage Suite pxvtable package instead (pxvtable::vnewobj). We will look at writing BioImage Suite extensions in a few weeks. Using the newname package we can rewrite the original code snippet as:

```
package require newname
set myimage [ vtkImageData [ newname::vnewobj ]]
$myimage SetDimensions 100 100 1
$myimage SetScalarTypeToFloat
$myimage AllocateScalars
```

In this case, myimage is an ordinary variable which contains the object name (probably something like myvtkobj1). The usual scoping rules apply etc. You can now access the object the sane way, using $myimage.

[2] In C++, this would be the equivalent of accessing variables directly using their memory location. While it can be done, it is asking for trouble.

[3] This, once upon a time, had pretensions to be a fully OOP-extension to .tcl. Some of that code is still there but it is not worth discussing any more.

10.3 Data Arrays

All data in VTK is stored ultimately in one of the many derived classes of vtkDataArray. vtkDataArray is an abstract superclass for classes representing arrays of vectors called tuples (or numbers treated as vectors of length 1). Each tuple consists of a set of numbers or components. Derived Classes of vtkDataArray include vtkUnsignedCharArray, vtkShortArray, vtkFloatArray, vtkDoubleArray etc. An abstract class is one which is never instantiated itself, it is rather a parent class which captures the common interface for a set of derived classes.

vtkDataArray-derived classes can function either as a dynamic array (lower performance) or a fixed length arrays. For example, in the case of vtkImageData, the intensities are stored in a vtkDataArray having dimensions equal to the number of voxels and vector length typically equal to 1 (3 for color images. In the case of multiframe data such as fMRI or cardiac data, the number of components is equal to the number of frames.

Fixed Length Array: The following script (script10-1.tcl) creates a fixed length array and manipulates it's contents.

```
# Prelimaries, load the newname package and withdraw the gui
lappend auto_path [ file dirname [ info script ]]
package require newname
wm withdraw .

# Create the Array
set arr [ vtkFloatArray [ newname::vnewobj ]]
$arr SetNumberOfComponents 2
$arr SetNumberOfTuples    12
# Set some defaults -- always a good idea
$arr FillComponent 0  0.0
$arr FillComponent 1 10.0
# Set some values
$arr SetComponent 10 0 3.0
$arr SetTuple2    11 9.0 2.0
$arr SetComponent 4 4 -2.1
# Print some info and then the array
puts stdout "The array has [ $arr GetNumberOfTuples ] tuples,
             [ $arr GetNumberOfComponents ] components"
puts stdout "Here are its contents"
set nt [ $arr GetNumberOfTuples ]
set nc [ $arr GetNumberOfComponents ]
for { set i 0 } { $i < $nt } { incr i } {
    puts -nonewline stdout "Tuple $i : ("
    for { set j 0 } { $j < $nc } { incr j } {
      puts -nonewline stdout "\t [ $arr GetComponent $i $j ]"
    }
    puts stdout "\t)"
}
# The exit command is explicitly needed in a vtk script
exit
```

We can access the elements in this array by using the SetComponent and GetComponent methods. All indices start at 0. The value of a whole component can be set using the FillComponent method as shown above. The

SetTuple commands (SetTuple2, SetTuple3 etc.) can be used to set whole vectors and the corresponding GetTuple commands can be used to get whole vectors as lists.

Dynamic Arrays: These are useful when one does not know ultimately how much data is coming. In this case we only specify the type of the array and the number of components. The following script (script10-2.tcl) illustrates this concept:

```
lappend auto_path [ file dirname [ info script ]]
package require newname
wm withdraw .

set arr [ vtkFloatArray [ newname::vnewobj ]]
$arr SetNumberOfComponents 1
$arr InsertNextTuple1 5
$arr InsertNextTuple1 10

set b [ $arr GetComponent 1 0 ]
puts stderr "b=$b"
```

There are additional functions (InsertNextTuple2, InsertNextTuple3,4,9) for multi-component arrays.

Data Types: The following constants are often defined, as shorthand for the numerical value that VTK uses to identify each data-type.

set VTK_VOID 0	set VTK_SHORT 4	set VTK_LONG 8
set VTK_BIT 1	set VTK_UNSIGNED_SHORT 5	set VTK_UNSIGNED_LONG 9
set VTK_CHAR 2	set VTK_INT 6	set VTK_FLOAT 10
set VTK_UNSIGNED_CHAR 3	set VTK_UNSIGNED_INT 7	set VTK_DOUBLE 11

In C++ one can use the static member function vtkDataArray::CreateDataArray(type) to create an array of type to be specified at run-time. In Tcl there is a less elegant solution as vtkDataArray can not be accessed directly. Here one creates a specific empty array (e.g. vtkFloatArray) and then uses this to create another array, e.g. ,

```
set emptyshell [ vtkFloatArray  [ newname::vnewobj ]]
set shortarray [ emptyshell CreateDataArray VTK_SHORT ]
```

10.4 Creating Curves and Surfaces

In VTK, we can create curves and surfaces using the vtkPolyData class, which is a derived class of vtkDataObject for storing polygonal data. The general structure of a vtkDataObject (see Chapter 9 for more details) is described in figure 10.1.

vtkPolyData is a complex class which has many members. The key ones are:

- Points of type vtkPoints represents the geometry of the surface (i.e. the points)
- Polys of type vtkCellArray represents part of the topology of the surface (i.e. the polygons)
- Lines of type vtkCellArray represents another part of the topology (the elements which are simple lines as opposed to faces).
- PointData of type vtkPointData represents data associated with the points (e.g. normals, colors etc)
- CellData of type vtkCellData represents data associated with the points (e.g. again normals, colors etc)

Creating a Polygonal Curve: A polygonal curve consists of a set of points and the lines that connect these. First some basic preliminaries (script10-3.tcl):

```
lappend auto_path [ file dirname [ info script ]]
package require newname
wm withdraw .
set pi 3.1415
```

Next we create the points. These are stored in a vtkPoints data structure which is very similar to a DataArray (it contains a data array and adds convenience methods such as GetPoint/SetPoint for manipulating this array). Our curve is a planar circle consisting of eight segments.

```
# Creating a circle with 8 segments
# First the points
set pts [ vtkPoints [ newname::vnewobj ]]
$pts SetNumberOfPoints 8
for { set i 0 } { $i <= 7 } { incr i } {
   set rad [ expr 2.0*$i*$pi/8.0 ]
   set x [ expr 10.0* sin($rad) ]
   set y [ expr 10.0* cos($rad) ]
   $pts SetPoint $i $x $y 0.0
   puts stdout "Point $i = [ $pts GetPoint $i ]"
}
```

From the geometry, to the topology. The circle has eight line segments which need to be specified. These are stored in a vtkCellArray data structure, which is a very powerful (=complex) object. This uses dynamic storage and the second command "Allocate 10 5", creates an initial memory allocation for 10 cells and instructs the array to allocate any additional memory in increments of 5. Inserting a Cell in Tcl is probably more complex than C++ given the lack of pointers. First we specify the size of the cell (InsertNextCell 2) and then we insert the individual point indices (InsertCellPoint). The point indices will link to points in the Points array (pts) above, with 0 being the first point. Since this is a closed curve, the last line segment consists of points 7 and 0.

```
# Next the line segments
set lines [ vtkCellArray [ newname::vnewobj ]]
$lines Allocate 10 5
for { set i 0 } { $i <= 7 } { incr i } {
   set p1 $i
```

```
   set p2 [ expr $i + 1 ]
   if { $p2 > 7 } { set p2 0 }
   $lines InsertNextCell 2
   $lines InsertCellPoint $p1;
   $lines InsertCellPoint $p2
   puts stdout "Set Line Segment $i = ($p1, $p2)"
}
```

Once we create the points and the lines, we can place them in a surface structure. We create this using the vtkPolyData command and attach the points and the lines to it.

```
# Create the curve object and set the points and lines
set curve [ vtkPolyData [ newname::vnewobj ]]
$curve SetPoints $pts
$curve SetLines $lines
```

The pts and the lines structures can now be deleted. VTK uses reference counted memory allocation (we will talk about this more later), and by setting the pts and lines as parts of the surface structure their reference count is increased, so issuing the Delete command does not actually delete them!

```
}
# Since VTK uses reference counting delete pts and lines
# as these are counted inside curve
$pts Delete
$lines Delete
```

Finally we can output the curve to a file, using the vtkPolyDataWriter class. All vtk writer classes have similar structure. The SetInput method is used to specify the data-structure to be saved, the SetFileName method specifies the filename and the Write method executes the operation.

```
# Save the curve to a file
set writer [ vtkPolyDataWriter [ newname::vnewobj ]]
$writer SetInput $curve
$writer SetFileName curve.vtk
$writer Write

# Clean Up all objects and exit
$writer Delete
$curve  Delete
exit
```

Creating a Surface: Our simple surface will be a cone which will consist of a circle and an apex. This is accomplished by modifying the previous script to yield a new script (script10-4.tcl) as follows. Each triangle

consists of a line segment on the curve as a base with the apex as the third point. The key changes from the previous file are:

```
...
set pts [ vtkPoints [ newname::vnewobj ]]
$pts SetNumberOfPoints 9
for { set i 0 } { $i <= 7 } { incr i } {
 ...
}
# Set apex point as point 8
$pts SetPoint 8 0.0 0.0 10.0

# Next the line segments
set triangles [ vtkCellArray [ newname::vnewobj ]]
$triangles Allocate 10 5
for { set i 0 } { $i <= 7 } { incr i } {
   set p1 $i
   set p2 [ expr $i + 1 ]
   if { $p2 > 7 } { set p2 0 }
   $triangles InsertNextCell 3
   $triangles InsertCellPoint $p1;
   $triangles InsertCellPoint $p2
   $triangles InsertCellPoint 8
   puts stdout "Set Line Segment $i = ($p1, $p2)"
}

set surface [ vtkPolyData [ newname::vnewobj ]]
$surface SetPoints $pts
$surface SetPolys $triangles
```

The key change is that we now have triangles as opposed to lines which are attached to the vtkPolyData object surface using the SetPolys method.

10.5 Displaying Polygonal Data

To display the curve and/or surface constructed previously we need to create a pipeline of the form shown in figure 10.2. The following script (script10-5.tcl) is similar to script10-4.tcl up to the point where vtkPolyDataWriter is created. At that point in the code we replace the file output operation with the display pipeline as shown below:

The first step is to create the mapper (mappers define the interface between data and graphics primitives)

```
# Create the
 mapper set map [ vtkPolyDataMapper [ newname::vnewobj ]]
 $map SetInput $surface
```

The next step is to create the actor which appears in the scene:

```
#Create the actor and set it to display wireframes
set actor [ vtkActor  [ newname::vnewobj ]]
$actor SetMapper $map
$map Delete
```

Next we tweak the display properties of the actor, using it's property element. We set the color to white (1,1,1) and the display to wireframe.

```
set property [ $actor GetProperty ]
$property SetColor  1 1 1 ; $property SetAmbient 1.0
$property SetDiffuse 0.0;   $property SetSpecular 0.0
$property SetRepresentationToWireframe
```

The next step is to create the renderer and add the actor to it. While we are at it we switch the renderer to parallel projection as opposed to perspective which is the default.

```
set ren [ vtkRenderer [ newname::vnewobj ]]
$ren AddActor $actor
[ $ren GetActiveCamera ] ParallelProjectionOn
```

We next create a render window, and attach the renderer to it.

```
set renWin [ vtkRenderWindow [ newname::vnewobj ]]
$renWin AddRenderer $ren
$renWin SetSize 300 300
```

Finally we create an interactor to handle mouse/keyboard input.

```
set iren [ vtkRenderWindowInteractor [ newname::vnewobj ]]
$iren SetRenderWindow $renWin
$iren Initialize
$iren AddObserver SetExitMethod { exit }
```

Pressing the "e" key invokes the exit command of the renderer which exits the program.

10.6 Reference Counting Memory Allocation/Deallocation

When a new object is instantiated, memory is allocated e.g. `set pts [ vtkPoints [ newname::vnewobj ]]`. As part of the construction of the new object, a counter called the reference count of the object is set to 1.

Each time the object is used (contained) by another object e.g. `$curve SetPoints $pts`, the reference number of the object `$pts` is increased by 1.

When the object `$curve` is deleted it will decrement the reference count of `$pts` by 1. If at this point in time `$pts` has a reference count of 0 it will get removed from memory, if it not it will stick around.

Calling the delete method of an object, e.g. `$pts Delete`, also decreases the number of references by 1. When the number of references is equal to zero, the memory is then released. Hence calling delete does not always result in (immediately) deleting the object.

If an object (e.g. `$pts`) is not meant to have an independent existence from its container object (e.g. `$curve` in this case), then once assigned to the container its delete method should be called to decrease its reference count back to 1. That way, when the container object is deleted, the dependent object is also deleted.

Assignment

1. Review the contents of this and the previous handout.
2. Play with the "Cone" scripts in the repository.
3. Using script10-3.tcl as a base, change the code to create a helix, i.e. a curve where the z-coordinate increase with each point as opposed to a constant 0.0. Save the output in a file helix.vtk
4. Using the last assignment as a base and the display code from script10-5.tcl (or your own version) write a script that creates and displays a helix.
5. Modify the last assignment to display both a helix and a cone. Hint: you will need two actors.

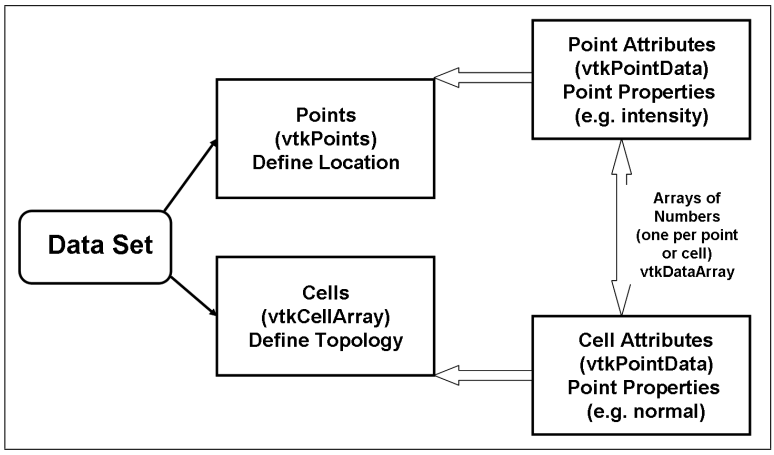


Figure 10.1: Constituent parts of a Data Object. A data object consists of a set of points which define its geometry (these are implicitly defined in the case of images), and a set of cells (e.g. lines, triangles, etc.) which define the topology (again this is implicitly define in the case of images). Both the points and the cells can have associated attributes (e.g. the intensities in an image are stored as point-data associated with each point).

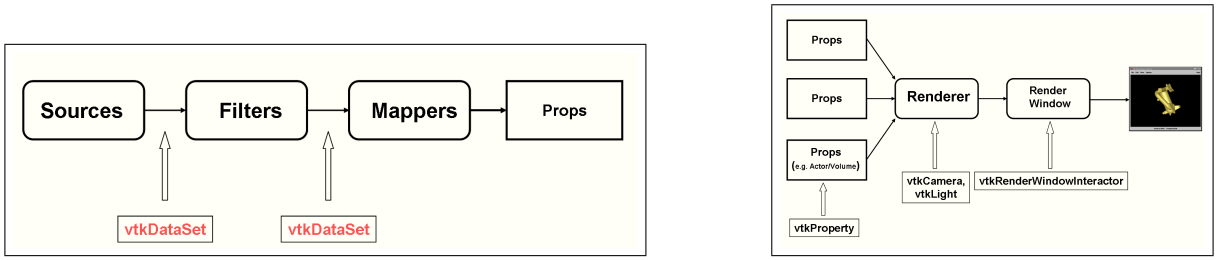


Figure 10.2: The VTK Pipeline Parts 1 and 2.

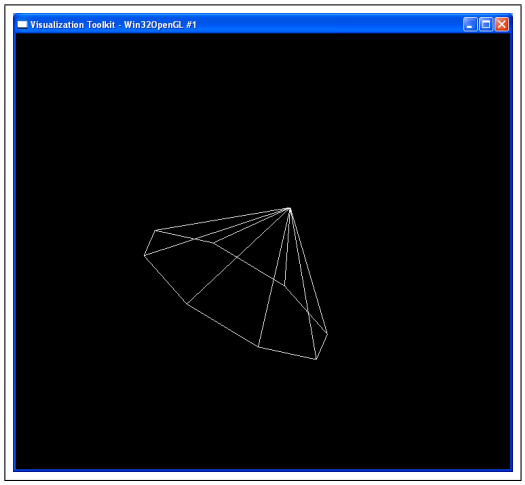


Figure 10.3: A screenshot of the script10-5.tcl.

Chapter 11

Images in VTK

Finally, after a number of chapters into a book on programming for *image* analysis we come to the point we talk about images per se for the first time. In particular, we set the foundations for creating and manipulating images. This chapter aims to set the stage for hands-on examples that will be described in the next chapter.

11.1 Introduction

In their early steps in programming for image processing/analysis, most people tend to regard and store an image as essentially a matrix of numbers. The only complication, at this stage, is the relationship between the row/column indices of the matrix and the x- and y-axes of the image, respectively.[1]

This type of radical simplicity disappears, unfortunately, once we have issues with mapping images. Examples include, image registration applications when we are looking to estimate a transformation between two images, statistical shape model building etc.

In general, in medical imaging, we need to keep track not only the image intensities but additional attributes such as voxel dimensions (voxels need not be isotropic), image orientation (the relationship of the x-,y- and z-axis orientation to the human body – see Figure 11.1), the position of the voxels etc.

In VTK, images are stored in a complex class `vtkImageData`. This shares much of the general structure of other VTK data objects, but because of it's regularity much of its geometry and topology can be specified using a very small number of parameters. It is also worth pointing out that in earlier versions of VTK (earlier than 4.0), the primary class for storing images was called `vtkStructuredPoints` – which is now, for the most part, an empty subclass of vtkImageData. If you see vtkStructuredPoints anywhere in the VTK documentation, you can safely substitute vtkImageData at this point.

11.2 The vtkImageData Class

In VTK, images are stored using the class vtkImageData. VTK, implicitly at least, treats an image as a function which takes values equal to those specified at the center of each voxel and interpolates in-between. The standard interpolation scheme used is tri-linear interpolation. In some cases (vtkImageReslice) other interpolation schemes are available. In addition to the image intensities themselves there are three key variables (arrays) in vtkImageData, which are used to define an image, namely:

[1] Often a problem in MATLAB, when often image matrices need to be transposed before display!

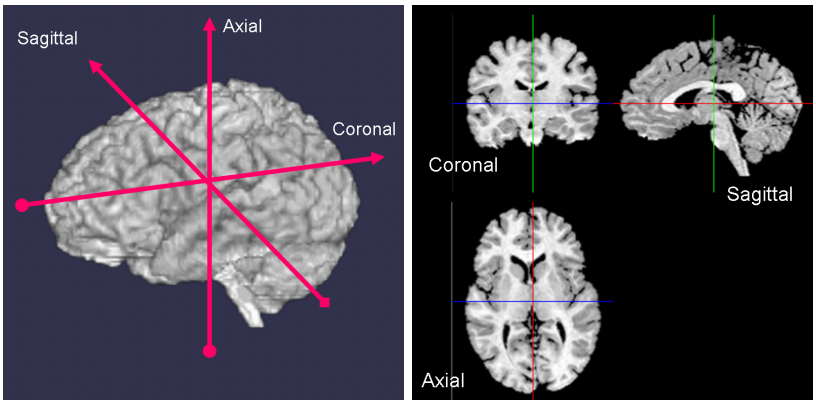


Figure 11.1: **Left:** Standard Image Acquisition Orientations, Axial or Transverse, Coronal and Sagittal. The arrows indicate the z-axis direction (which may be inverted depending on the acquisition protocol), the x-axis and the y-axis are perpendicular to this. **Right:** Axial, coronal and sagittal slices. Oblique acquisitions are also sometimes used, in which the z-axis is rotated, or *obliqued* away from one of the standard acquisitions, e.g. coronal oblique.

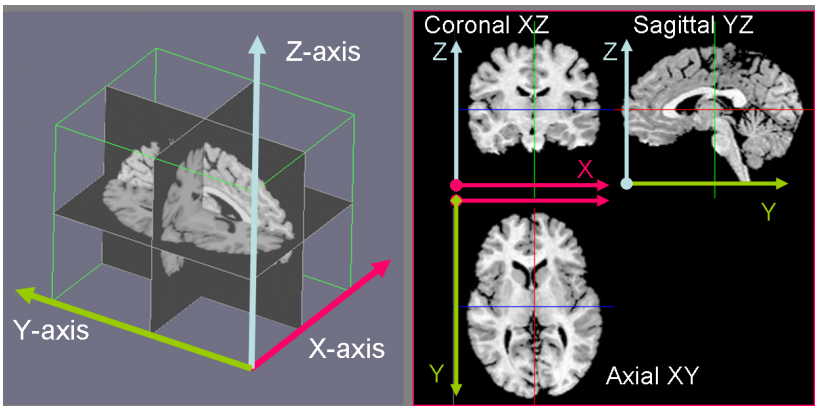


Figure 11.2: Axes orientation for a typical axial acquisition.

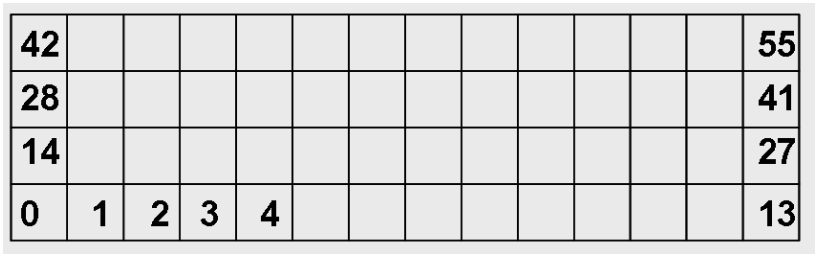


Figure 11.3: Voxel indices in a 14×4 image. Voxels are ordered in raster-scan order.

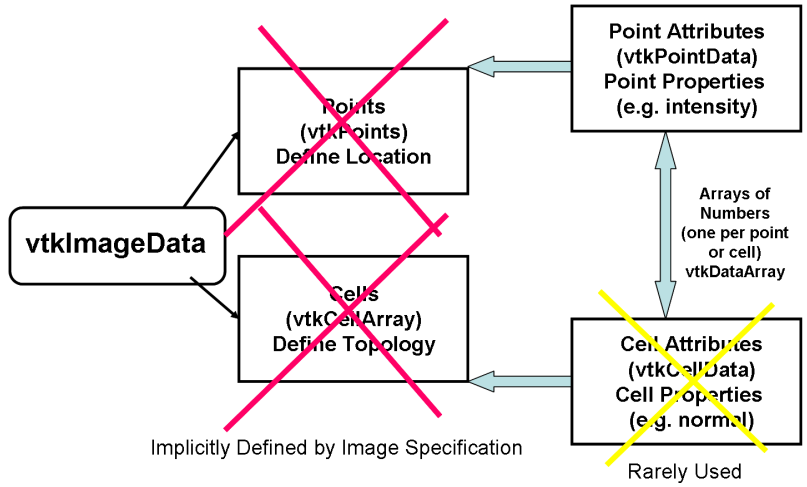


Figure 11.4: Because of the regularity of the underlying geometry/topology, the only aspect of the complex vtkDataObject data-structure that needs to be explicitly defined in vtkImageData is the PointData, where we store the image intensity. The geometry and topology can be specified simply by specifying the dimensions of the grid, the position of the corner, and the size of the voxels.

- `int Dimensions[3]` — the number of voxels in the x,y and z directions respectively. In the case of the 'image' shown in figure 11.3, these array takes values 14,4,1.
- `double Origin[3]` — the position in 3D space of the centroid of the first voxel (with index 0, as defined in figure 11.3.) Often this is specified with respect to the center of the scanner, although it is often simply set to 0,0,0.
- `double Spacing[3]` — the physical size of each voxel e.g. $1 \times 1 \times 1.2mm$. While sometimes images are isotropic, this is not always the case, so one needs to be careful.

Once these variables are set, they completely define the topology and the geometry of the image. Hence (as shown in Figure 11.4), in a vtkImageData we do not specify the positions of the points or the topology explicitly, these are simply implicitly defined by these three variables. The key element in the specification of a vtkImageData is a Data Array that is contained in the PointData attributes.

This Data Array has as many "tuples" as the image has voxels, and each tuple has a number of components corresponding to the components (in color images) or, frames in 4D images. The tuples are ordered in *raster-scan* order as illustrated in Figure 11.3.

11.3 Creating and Manipulating Images

11.3.1 Creating an Image

Creating an image involves the following:

1. Creating the instance of vtkImageData
2. Specifying the Dimensions
3. Optionally specifying the Spacing – if not (1.0,1.0,1.0).
4. Optionally specifying the Origin – if not (0.0,0.0,0.0).
5. Optionally specifying the number of components – if not 1.
6. Specifying the image type, e.g. short, float etc.
7. **Allocating Memory**

In Tcl (using the vtk interpreter!) this can be accomplished using the following commands:

```
package require newname
set img [ vtkImageData [ newname::vnewobj ]]
$img SetDimensions 10 8 4
$img SetSpacing 1.0 1.0 1.0
$img SetOrigin 0.0 0.0 0.0
$img SetNumberOfScalarComponents 1
$img SetScalarTypeToFloat # or Short,Int,Double,Char,UnsignedChar etc.
$img AllocateScalars
```

11.3.2 Manipulating Image Intensities

The Slow Method: The easiest method for accessing and modifying image data is to use the pair of methods:

```
double  GetScalarComponentAsDouble (int x, int y, int z, int component)
void    SetScalarComponentFromDouble (int x, int y, int z, int component,double v)
```

In the example above to get the intensity at voxel (5,3,1) and component 0 we can call:

```
set v [ $img GetScalarComponentAsDouble 5 3 1 0 ]
```

To set the value at voxel (4,6,2) and component 0 to 15.0 we can call:

```
$img SetScalarComponentFromDouble 4 6 2 0 15.0
```

Note: Obviously if the underlying image is short, setting an intensity value to, for example, 15.2 will result in the intensity being set to 15! While the intensity is specified using a double variable (i.e. one that stores numbers in double precision floating type) this is appropriately (or inappropriately) converted to the specific type of the image prior to it being set!

The Somewhat Faster Method: In this method, we first get a pointer to the underlying data array in which the data is stored and manipulate this array directly. This is accomplished by:

```
# The two-step method
set pdata [ $img GetPointData ]
set data  [ $pdata GetScalars ]

# The 'direct'-method
set data  [ [ $img GetPointData ] GetScalars ]
```

Once the pointer to the underlying array is obtained, we can manipulate intensities using the GetComponent/SetComponent methods of the data array, e.g.

```
set v [ $data GetComponent 115 0 ]
$data SetComponent 224 0 15.0
```

where 115 is raster-index of voxel (5,3,1) and 224 is the raster-index of voxel (4,6,2) from before.

Direct Access, C++ only: The fastest method involves realizing that things like two and three-dimensional arrays are simply fictions of our imagination (or really convenience devices) and that all array data is ultimately stored in one-dimensional arrays (since memory is inherently one dimensional). To operate on an image at maximum speed one needs to, in C++, access and manipulate the underlying data pointer.

11.4 Additional VTK Classes for Image Manipulation

11.4.1 Image Input and Output

VTK supports by default a number of standard image file formats for read/write:

- Binary – vtkImageReader
- JPEG – vtkJPEGReader, vtkJPEGWriter
- – vtReader, vtWriter (ppm,pgm)
- TIFF – vtkTIFFReader, vtkTIFFWriter
- BMP – vtkBMPReader, vtkBMPWriter
- DICOM – vtkDICOMImageReader (not complete)

There are BioImage Suite extensions for reading Analyze, Signa LX/SPR, Prism (SPECT) etc.

11.4.2 Image to Image Filters

VTK has a rich set of filters for manipulating images. These are derived from vtkImageToImageFilter. Some common examples are:

- Smoothing – vtkImageGaussianSmooth, vtkImageMedian3D
- Computing Gradiens/Laplacians – vtkImageGradient, vtkImageLaplacian
- Fourier Operations – vtkImageFFT,vtkImageRFT
- Resampling/Reslicing – vtkImageResample, vtkImageReslice (vtkImageReslice on its own is reason enough to learn VTK, it implements enough operations that would take more than a year to code from scratch!)
- Flipping, Permutting – vtkImageFlip,vtkImagePermute

11.4.3 A Simple Example

The following simple example (script11-1.tcl) generates an 20×20 image which is a smoothed "x". (Some lines are omitted for clarity.)

First we load the package newname and hide the default window since this is a command-line script:

```
lappend auto_path [ file dirname [ info script ]]; package require newname
wm withdraw .
```

We then create the image and fill it with zeros:

```
set img [ vtkImageData [ newname::vnewobj ]]
$img SetDimensions 20 20 1
$img SetNumberOfScalarComponents 1
$img SetScalarTypeToUnsignedChar
$img AllocateScalars
set data  [ [ $img GetPointData ] GetScalars ]
$data FillComponent 0 0
```

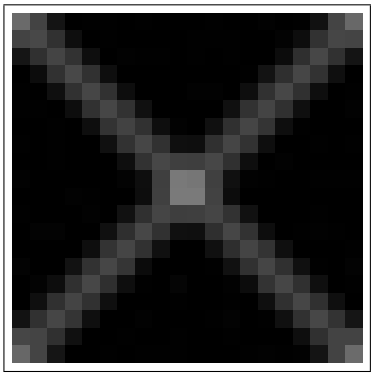


Figure 11.5: The Output JPEG File.

Next we draw two diagonal lines $(i == j)$ and $(i + j = 19)$:

```
for { set i 0 } { $i <= 19 } { incr i } {
    for { set j 0 } { $j <= 19 } { incr j } {
          if { $i == $j || [ expr $i + $j ] == 19 } {
            $img SetScalarComponentFromDouble $i $j 0 0 200
          }
        }
}
```

The smoothing filter is used to slightly blur the image:

```
set smooth [ vtkImageGaussianSmooth [  newname::vnewobj ] ]
$smooth SetInput $img
$smooth SetStandardDeviations 1.0 1.0 0.0
$smooth SetDimensionality 2
$smooth Update
```

The output is saved in a jpeg file:

```
set writer [ vtkJPEGWriter [ newname::vnewobj  ] ]
$writer SetFileDimensionality 2
$writer SetFileName "smoothedx.jpeg"
$writer SetInput [ $smooth GetOutput]
$writer Write
```

We clean up the objects and exit.

```
$writer Delete;$smooth Delete;$img Delete
exit
```

11.5 Visualizing Images

In displaying three-dimensional images, we face an additional complexity when we compare this task to the task of displaying polygonal surfaces. Surface display is predicated on the fact that surfaces are "thin" structures with no "inside". Images are, on the other hand, "thick" structures, where the "inside" is often more important than the bounding voxels. Hence direct display of a three-dimensional image is problematic, it will simply look like a cube.

The most common ways to render 3D images are:

- Orthogonal Slices – e.g. example in Figure 11.1 (right).
- Oblique Slices – slices obtained by resampling the image along a plane that is not aligned with one of the coordinate axes.
- Volume Rendering – a "3d-rendering" of the whole image using a combination of intensity mapping and transparency mapping tricks.

While image slices can be displayed as surfaces with the intensity of each voxel mapped to patches of a surface, they are most commonly displayed as textures mapped on rectangles of appropriate dimensions. As stated on the Wikipedia page (paraphrasing) texture mapping is a method of adding detail, surface texture, or color to a computer-generated graphic or 3D model. A texture map is applied (mapped) to the surface of a shape. This process is akin to applying gift wrapping paper to a plain white box. This allows the realization of detail that would take very many additional polygons to realize otherwise. This kind of coloration is the most common application of texture mapping.

As an aside, the underlying rendering engine used in VTK, OpenGL, renders all textures in dimensions that are powers of two. The images are interpolated before display, hence some (small) loss of sharpness takes place (only visible in small images) E.g. an 100×50 image will be resampled to 128×64 before display. In cases of large images (especially volume rendering) this is worth keeping in mind. A 257×257 image has the same effective dimensions as a 512×512 image as far as display-performance is concerned. In such cases, small amounts of judicious cropping can result in substantial improvements in rendering time.

We will discuss image rendering in more detail in the next Chapter.

Assignment

- Read the Handout!
- Read the vtkImageData Man page `http://noodle.med.yale.edu/vtk/classvtkImageData.html` to get fuller sense of the functionality in vtkImageData.
- Modify, in three steps, script11-1.tcl to :
  1. Change the code to generate a square, rather than an "x" shape.
  2. Replace the Smoothing filter with a GradientMagnitude filter (vtkImageGradientMagnitude).
  3. Replace the JPEG Writer with a PNM Writer.

A more complete image-manipulation assignment can be found at the end of the next Chapter.

Chapter 12

Displaying Images in VTK

Following a brief discussion on colormaps, we covers three different methods for displaying images in VTK, (i) using the dedicated vtkImageViewer/vtkTkImageViewerWidget classes, (ii) displaying slices using texture mapping on planes and (iii) volume rendering.

12.1 Introduction

There a number of ways to display images depending on the need to display additional elements in conjuction with the images. In general, images are best displayed in a 3D rendering environment which enables polygonal objects to be jointly displayed with them. This used to the "holy grail" of medical image visualization when the first machines with 3D rendering capabilities appeared about 10-15 years ago. This enables the visual inspection of, for example, segmentation results, by co-visualizing the underlying image and the surface of the segmented structure.

There are three basic ways of displaying images in VTK. The first, is to use the dedicated vtkImageViewer class which renders images using bitmaps and is only really suitable for 2D image display with little additional elements. The second method is to display selected slices through the 3D image as textures mapped on planes in 3D space. The final method is volume rendering which attempts to directly visualize the whole 3D volume using a set of transparency tricks.

I have learned, the hard way, that it is dangerous to modify the original image for display purposes, in any way. Visualization techniques, such as camera manipulation and zooming, can be used to flip and enlarge images. Appropriate color mapping schemes can be used to change the image appearance to enhance the contrast. Hence it is rarely the case that the original image needs to be modified.

My five rules of image display are as follows:

1. Never modify the original image
2. If the image needs to be displayed in a different orientation, move the camera – never modify the original image.
3. If the image needs to be contrast-adjusted, use a colormap, never modify the original image
4. If an image needs to be magnified, use the zoom control on the camera, never modify the original image.
5. If in doubt, see rule 1.

Figure 12.1: The Window/Level Colormap.

12.2 Colormaps

Colormaps (or lookup tables) are essentially functions that map image intensity to display color. The most common colormap (which is often implicitly used) simply maps the smallest value in the image to black and the highest value to white. For example, in the case of an image with range 0:255, 0 is mapped to black, 255 to white and everything in between to progressively lighter shades of gray. The most common medical image colomap is the so called Level/Window colormap. This colormap is defined by the variables, the level l and the window size w. The mapping $x \mapsto y$, where x is the input intensity and y the output color (from $black = 0$ to $white = 1$) is then specified as:

$$y = \begin{cases} 0 & if x \leq l - \frac{w}{2} \\ 1 & if x \geq l + \frac{w}{2} \\ \frac{x-(l-\frac{w}{2})}{w} & \text{otherwise} \end{cases}$$

Colormaps in VTK are explicitly defined using an instance of the vtkLookupTable class. This defines the colormap as a lookup table, consisting of N rows, each row having 4 elements (RGBA=Red,Green,Blue,Alpha). In addition, we define the table range $L : H$. The mapping is then performed in two steps: First given x we compute i the index into the lookup table. Then. the output array y is simply row i of the lookuptable.

$$i = \begin{cases} 0 & if x \leq L \\ N - 1 & if x \geq H \\ \frac{x-L}{H-L} \times (N-1) & \text{otherwise} \end{cases}$$

The lookup table setup can be used to construct very complex Colormaps. In code this takes the form:

```
set cmap vtkLookupTable [ newname::vnewobj ]]
$cmap SetNumberOfColors 256
$cmap SetTableRange 0 255
for { set i 0 } { $i  < 256 } { incr i } {
    set v [ expr $i /255.0 ]
    $cmap SetTableValue $i $v $v $v 1.0
}
```

See the man page of vtkLookupTable for more details.

12.3 An Aside – Using the Dedicated Image Viewer

When all one needs is the ability to display simple 2D Image Slices, then the vtkImageViewer class (embeded in the vtkTkImageViewerWidget) is ofen the most convenient option. This takes four inputs namely: (i) Input – an image of type vtkImageData, (ii) ZSlice – the slice to be displayed in the case of 3D Images, (iii) ColorLevel – the Level of the Colormap and (iv) ColorWindow – the Window of the Colormap.

The following simple script (script12-1.tcl) illustrates it's use. The first part is fairly standard:

```
lappend auto_path [ file dirname [ info script ]]
package require newname
wm geometry . 300x150
```

Next we create a simple GUI with an empty frame (top) for the viewer and a slider/button combination on the bottom for selecting the slice and exiting.

```
set top [ frame .top ]; set bot [ frame .bottom ]
pack $bot -side bottom -expand false -fill x -pady 2 -padx 20
pack $top -side top -expand true -fill both

set slice 0
label $bot.a -text "Smoothness:"
set slicescale [ scale $bot.b -variable slice -from 0 -to 10 -digits 3 \
                 -orient horizontal -resolution 1 ]
pack $bot.a $bot.b -side left -expand true -fill x
button $bot.exit -text "Exit!" -command { destroy . ; exit }
pack $bot.exit -side right
```

Next we load the image, which is stored, in this case, in the native VTK format. Upon loading, we print out some diagnostics and get the number of slices, which is used to configure the range of the slider

```
# Load the Image
set reader [ vtkStructuredPointsReader [ newname::vnewobj ]]
$reader SetFileName brain.vt; $reader Update

set img [ $reader GetOutput ]
puts stdout "dimensions [ $img GetDimensions ] spacing [ $img GetSpacing ]"
set numslices [ lindex [ $img GetDimensions ] 2 ]
$slicescale configure -to [ expr $numslices - 1 ]
```

Now we are ready to create the viewer widget. This is created and packed much like a usual Tk Widget. We call the GetImageViewer method to get the actual image viewer (a special type of renderer, of type vtkImageWindow, that is embedded in the viewer widget).

```
set v1 [ vtkTkImageViewerWidget $top.1 -height 170 -width 170 ]
pack $v1  -side top -expand true -fill both -pady 2
# Force GUI to be drawn
update
# Get the Image Viewers
set viewer [ $v1 GetImageViewer ]
```

Next we set 4 inputs for the viewer, as described previously:

```
$viewer SetInput $img
$viewer SetZSlice 0
$viewer SetColorLevel 128; $viewer SetColorWindow 255
$viewer Render
```

Finally a little bit of magic to set the slice from the GUI and to process "expose" events. GUI-elements need to re-draw themselves each time they are exposed, i.e. a window moves from being above them etc. VTK Render widgets, unlike say simple widgets like buttons, do not automatically do this – unless they are explicitly controlled by an interactor. They need be asked politely to re-draw themselves using their Render method. This is accomplished using the Tcl "bind" command which captures GUI-events and sends them to appropriate procedures (callbacks). A similar technique is used to detect when the value of the slider has changed.

```
eval "bind $v1 <Expose> { $viewer Render }"
bind $slicescale <ButtonRelease> { SetSlice }
```

Finally the SetSlice callback (which in the script is near the top). We use global variables here for simplicity.

```
proc SetSlice { } {
    global slice;
    global viewer
    $viewer SetZSlice [ expr int($slice) ];
    $viewer Render
}
```

The Image Viewer has a number of weaknesses. Images are displayed using simple bitmaps in actual-size i.e. an 100x100 images will use 100x100 pixels with no possibility for zooming etc. These limitations can be overcome by magnifying the image, but this requires modifying the actual data directly which should be avoided (see rule 1). Texture mapping and 3D rendering offer alternatives.

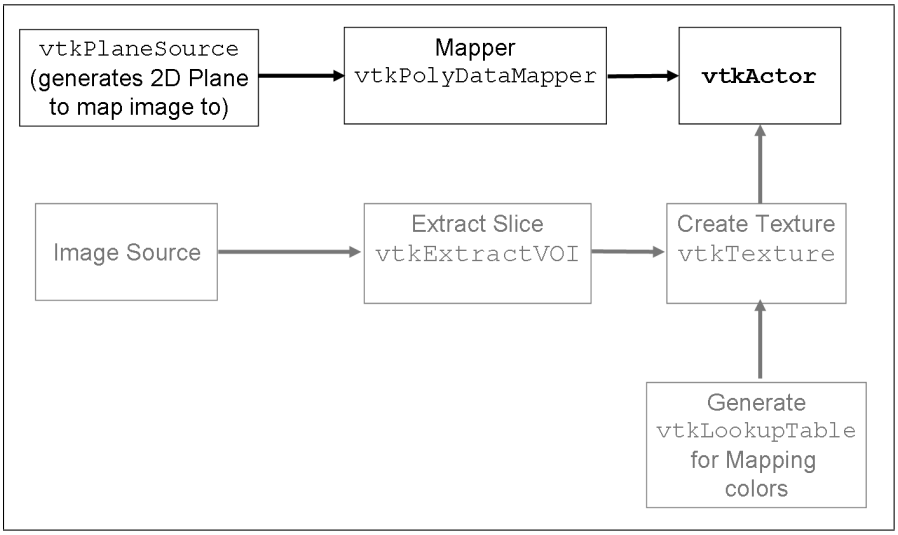


Figure 12.2: The pipeline for displaying an image slice as a texture mapped onto a rectangular plane. The top row is the geometry pipeline which simply displays a bounded plane. The middle row is the texture pipeline for generating the appearance of the rectangle. A colormap (bottom) is added to perform the mapping from image intensity to colors.

12.4 Displaying Texture Mapped Slices

The pipeline for displaying slices as textures mapped onto planes is shown in Figure 12.2. In general there are three parts: (i) We draw the 'geometry' by placing a rectangle at the appropriate position. (ii) We generate the source image by extracting the appropriate 2D slice from a potentially 3D image. (iii) We create the texture by combining this 2D image slice with a Colormap and apply it to the geometry.

The fist part of the script (script12-2.tcl) is fairly standard:

```
lappend auto_path [ file dirname [ info script ]]
package require newname; wm geometry . 200x200
```

Next we load the image using a StructuredPointsReader, which reads in images stored in the native vtk format. Following this we extract a single slice from the image using the vtkExtractVOI filter (VOI= Volume of Interest). Incidentally, if we wanted to extract a "coronal" slice we could specify `$voi SetVOI 0 41 25 25 0 55`, to extract a slice that has constant y-axis.

```
set tr [ vtkStructuredPointsReader [ newname::vnewobj ]]
$tr SetFileName brain.vt
$tr Update

set voi [ vtkExtractVOI [ newname::vnewobj ]]
$voi SetInput [ $tr GetOutput ]
$voi SetVOI 0 41 0 47 30 30
```

The next step is to generate a simple colormap that maps 0 to black and 1 to white. Then a texture (vtkTexture) is created which takes two inputs: (i) the image slice – from the output of vtkExtractVOI, and (ii) the colormap.

```
set colormap [ vtkLookupTable [ newname::vnewobj ]]
$colormap SetNumberOfColors 256
$colormap SetTableRange 0 255
for { set i 0 } { $i  < 256 } { incr i } {
    set v [ expr $i /255.0 ]
    $colormap SetTableValue $i $v $v $v 1.0
}

set texture [ vtkTexture [ newname::vnewobj ]]
$texture SetInput [ $voi GetOutput ]
$texture InterpolateOn
$texture SetLookupTable $colormap
```

Once the texture is ready we shift focus to the geometry. This consists of a plane (created using vtkPlaneSource). This specific image slice has dimensions 42×48 and voxel spacing (1.0,1.0,1.0), with the origin (the centroid of the "0"-th voxel) at (0.0,0.0,0.0). Hence the plane must extend from the bottom-left corner of the bottom-left voxel (-0.5,-0.5,30.0) to the top-right corner of the top-right voxel (40.5,46.5,30.0), as opposed to simply to voxel centroids. The plane source is defined by three points, the origin, and two points which define the two 'axis'-lines, Point1 and Point2. The output of the plane source is fed to the Mapper.

```
set imageplane [ vtkPlaneSource [ newname::vnewobj ]]
$imageplane SetXResolution 1
$imageplane SetYResolution 1
$imageplane SetOrigin -0.5 -0.5  30
$imageplane SetPoint1 40.5 -0.5  30
$imageplane SetPoint2 -0.5  46.5 30

set map [ vtkPolyDataMapper [ newname::vnewobj ]]
$map SetInput [ $imageplane GetOutput ]
```

Geometry and Texture meet at the Actor. This takes two inputs, the mapper which defines the geometry and the texture which defines the appearance:

```
set imactor [ vtkActor [ newname::vnewobj ]]
$imactor SetMapper $map
$imactor SetTexture $texture
```

Finally, the usual infrastructure for the display. We creatre a Renderer/RenderWindow and a RenderWindowInteractor.

```
set ren [ vtkRenderer [ newname::vnewobj ]]
$ren AddActor $imactor

set renwin [ vtkRenderWindow [ newname::vnewobj ] ]
$renwin AddRenderer $ren
```

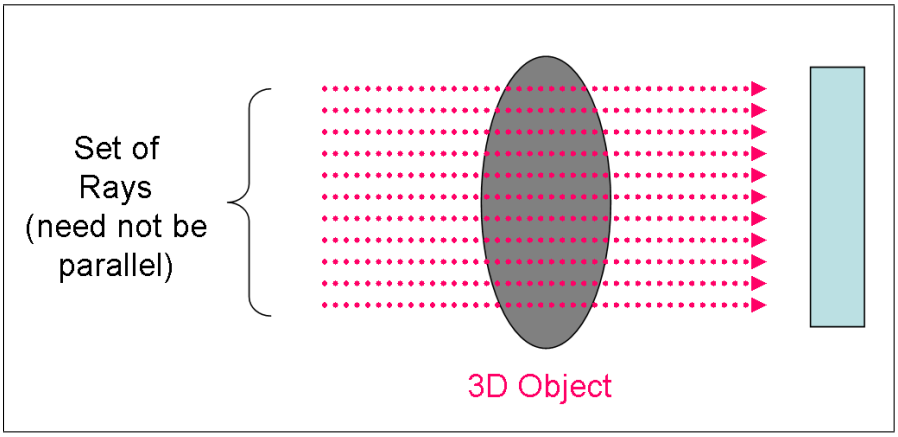


Figure 12.3: In volume rendering, a 2D view of the whole 3D image is generated by a process known as ray-casting. At each point in the camera plane a ray is projected through the 3D image where an integral of the form $\int_l F(\text{intensity}) \times G(\text{transparency})dl$ is computed.

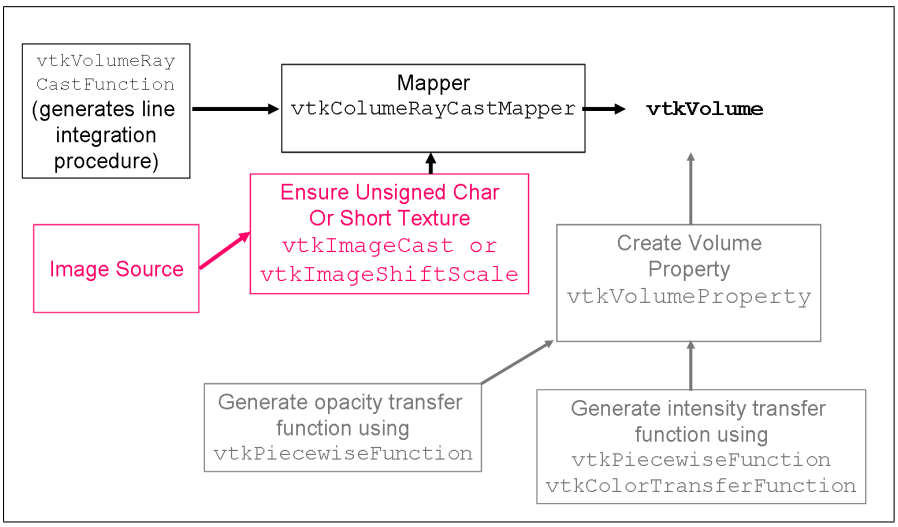


Figure 12.4: The pipeline for volume rendering. This is not as complicated as it first appears and is described in the text.

```
$renwin SetSize 300 300
$ren ResetCamera
$renwin Render

# Interactor
set iren [ vtkRenderWindowInteractor [ newname::vnewobj ] ]
$iren SetRenderWindow $renwin
$iren Initialize
$iren AddObserver SetExitMethod { exit }

wm withdraw .; vwait forever
```

12.5 Volume Rendering

Volume rendering is the art of generating a 2D view of an object that is, in some way, on integral of the whole 3D volume. The whole process involves, in most cases, integrating a function of the image along parallel rays, as illustrated in Figure 12.3, to generate a single 2D representation of the image. For example, in maximum-intensity projection which is often used in angiography, the output of the ray-casting is simply the maximum value encountered in the image along each ray. For more volumetric data, a more complex representation is used where the integral becomes a function of the intensity mapping assigned to each voxel intensity and the transparency

mapping assigned to each intensity. Defining these mappings such that some structures become transparent whereas others are highlighted is an art which, in some respects, resemples a semgnetation process using simple thresholds.

VTK provides a number of different volume rendering techniques, which are covered in detail in the VTK User's guide. We discuss two of them here, which both involve the same type of ray casting described in Figure 12.3. In the first method, the ray casting is done in software, whereas in the second method, we render a set of 2D texture plane images and let the graphics card simulate the ray-casting process in hardware. This second method uses hardware acceleration to give significantly better performance at the expense of a slight decrease in quality and flexibility.

The basic structure of both methods is similar, see Figure 12.4 for the pipeline. The pipelines involves: (i) Ensuring that the input image is either UnsignedChar or Short using the vtkImageCast or vtkImageShiftScale filters. (ii) Creating the two mapping functions, the opacity transfer function (G) and the intensity transfer function (F) and attaching them to vtkVolumeProperty object. (iii) Creating the ray casting structure using a combination of a vtkVolumeRayCastFunction which is then attached to a vtkVolumeRayCastMapper, to which the image generated in (i) is fed. Finally the Mapper and vtkVolumeProperty meet at an output class called vtkVolume which is to volume rendered data what vtkActor is to polygonal data.

We note that for the texture mapping approach, the only changes are in the top row, where there is no longer a vtkVolumeRayCastFunction and we replace the VolumeRayCastMapper with a VolumeTextureMapper.

Software Volume Rendering by Ray Casting: This is illustrated in the following script (script12-3.tcl). The first part of the script is the standard header:

```
lappend auto_path [ file dirname [ info script ]]
package require newname
wm geometry . 200x200
```

Next we read the image in, and cast it to Unsigned Char using an instance of vtkImageCast:

```
# Create the reader for the data
# First Load the Image
set reader [ vtkStructuredPointsReader [ newname::vnewobj ]]
$reader SetFileName brain.vt
$reader Update

set cast [ vtkImageCast [ newname::vnewobj ]]
$cast SetInput [ $reader GetOutput ]
$cast SetOutputScalarTypeToUnsignedChar
```

Next, we create the two transfer functions. For the opacity, this is a scalar function that maps the input intensity to a scalar range (0.0:1.0). This is a piecewise linear function (as implied by the class name vtkPieceswiseFunction) which generates a mapping based on a number of knot points. Functions of arbitrary complexity can be used. The basic principle of most volume rendering is that some dark structures (typically outside the part of the image we are interested in) are made completely transparent by mapping a low intensity value (e.g. 40.0 in this case) to an opacity of 0.0 (completely transparent). We often saturate the upper range as well.

The intensity transfer function is generated using an instance of vtkColorTransferFunction. This takes the intensity value and generates a vector output. This vector has three components corresponding to the red, green and blue output color. For gray-scale like output we set the three components to equal values. This particular example, is simply a linear ramp from black to white.

Finally the two transfer functions are attached to a an instance of vtkVolumeProperty.

```
# Create transfer mapping scalar value to opacity
set opacityTransferFunction [ vtkPiecewiseFunction [ newname::vnewobj ]]
$opacityTransferFunction AddPoint  40   0.0
$opacityTransferFunction AddPoint  255  1.0

# Create transfer mapping scalar value to color
set  colorTransferFunction [ vtkColorTransferFunction [ newname::vnewobj ]]
$colorTransferFunction AddRGBPoint      0.0 0.0 0.0 0.0
$colorTransferFunction AddRGBPoint    255.0 1.0 1.0 1.0

# The property describes how the data will look
set volumeProperty [ vtkVolumeProperty [ newname::vnewobj ]]
$volumeProperty SetColor $colorTransferFunction
$volumeProperty SetScalarOpacity $opacityTransferFunction
$volumeProperty SetInterpolationTypeToLinear
```

The next step is to generate the ray-casting itself (top-row of Figure 12.4). This consists of a vtkVolumeRayCast-CompositeFunction class (which can be used to specify the sampling rate of the ray-casting, we use the defaults here) and the Volume Mapper – which is the replacement for vtkPolyDataMapper for volume rendered data. The Mapper takes two inputs, the ray-cast function and an input image. The image must of of types unsigned char or short for this to work.

```
set compositeFunction [ vtkVolumeRayCastCompositeFunction [ newname::vnewobj ]]
set volumeMapper [ vtkVolumeRayCastMapper [ newname::vnewobj ]]
$volumeMapper SetVolumeRayCastFunction $compositeFunction
$volumeMapper SetInput [ $cast GetOutput]
```

Finally we create the Volume itself. This is of type vtkVolume and is the replacement for vtkActor for volume-rendered data. It takes as inputs, the VolumeMapper and the VolumeProperty created earlier.

```
# The volume holds the mapper and the property and
# can be used to position/orient the volume
set volume [ vtkVolume [ newname::vnewobj ]]
$volume SetMapper   $volumeMapper
$volume SetProperty $volumeProperty
```

The rest should be familiar by now. The volume is attached to a renderer in a similar fashion the way we attach vtkActor's. Then the usual RenderWindow and RenderWindowInteractor are created. The only additional element

is that we also set the desired update rate of the render window (SetDesiredUpdateRate) which defines the quality of the volume rendering – the lower the value the better. This is often valuable on slower computers (e.g. laptops):

```
set ren [ vtkRenderer [ newname::vnewobj ]]
$ren AddVolume $volume

set renwin [ vtkRenderWindow [ newname::vnewobj ] ]
$renwin AddRenderer $ren
$renwin SetSize 300 300
$ren ResetCamera
$renwin Render
$renwin SetDesiredUpdateRate 1.0

set iren [ vtkRenderWindowInteractor [ newname::vnewobj ] ]
$iren SetRenderWindow $renwin
$iren Initialize
$iren AddObserver SetExitMethod { exit }

wm withdraw .; vwait forever
```

Hardware Accelerated Volume Rendering by Texture Mapping: Hardware accelerated volume rendering can yield significant performance improvements over software-based ray-casting. This was a feature available ten years ago on only really expensive graphics boards (typical medium-high end SGI graphics boards run close to $10,000!). Now, the average sub $50 graphics board is fully capable of handling decent sized volumes! A major revolution happened, when NVIDIA released Open-GL accelerated graphics drivers for Linux, which enabled the transition from SGI-only setups to Linux about 5-6 years ago (2000 or so). This capability is now practically universal.

To leverage hardware accelerated texture mapping for volume rendering we replace (new script script12-4.tcl) the following lines from the previous script (script12-3.tcl):

```
set compositeFunction [ vtkVolumeRayCastCompositeFunction [ newname::vnewobj ]]
set volumeMapper [ vtkVolumeRayCastMapper [ newname::vnewobj ]]
$volumeMapper SetVolumeRayCastFunction $compositeFunction
$volumeMapper SetInput [ $cast GetOutput]
```

with:

```
# The mapper
# THIS IS THE ONLY CHANGE FROM script12-3.tcl !!
set volumeMapper [ vtkVolumeTextureMapper2D [ newname::vnewobj ]]
$volumeMapper SetInput [ $cast GetOutput]
# END OF CHANGE
```

This switches the mapper to texture mapping and eliminates the ray cast composite function.

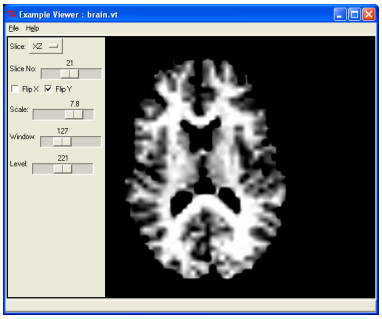


Figure 12.5: A Complete Viewer Application.

12.6 A Big Example

A complete image viewer application is presented in script12-5.tcl. This uses the [Incr Tcl] extensions to build a more-or-less functional 3D image viewer. The script consists of about 380 lines of code, and it is too long to reproduce here. Rather I will highlight some key features, and expect you to look at it yourselves.

The viewer application consists of (i) an [Incr] Tcl class MyViewer and (ii) a little bit of code at the end to initialize this. MyViewer has the following components:

Methods

- Constructor/Destructor: Create and Destroy the object.
- InitializeParameters: Set all member variables to default values
- PermuteImage: Since we are using the vtkImageViewer class, this is used to set to change the axis of the image around to permit different slices to be displayed.
- FlipScaleAndDisplayImage: As the names suggests, this performs left-right and top-bottom flips of the image and sends it to the display.
- ResetParametersAndGUI: Upon Image Load reset the ranges of the scales etc.
- LoadImage : Loads an Image
- BuildControlFrameGUI : Builds the Left frame with all the controls
- GenerateMenuUI : Create the menu
- CreateViewer: Creates the viewer
- GenerateUI : Master routine for building the UI

Member Variables: There a number of these for storing (i) key widgets, (ii) the two images, currentimage=the original load image and currentresults=the modified image for display and (iii) the params static array for storing variables bound to widgets.

Breaking the Pipeline A key technique illustrated in this example is "breaking the pipeline". Most VTK example code assumes that the visualization will be run once through and that the pipeline goes from data source to display. In real programs, we have a number of small pipelines that do piecemeal tasks that are more "manually connected". For example consider the PermuteImage method shown below:

```
itcl::body MyViewer::PermuteImage { } {
    set perm [ vtkImagePermute [ newname::vnewobj ]]
    $perm SetInput $currentimage
    switch $params($this,slice) {
      "XY"  { $perm SetFilteredAxes 0 1 2 }
```

```
      "YX"  { $perm SetFilteredAxes 1 0 2 }
      "XZ"  { $perm SetFilteredAxes 0 2 1 }
      "ZX"  { $perm SetFilteredAxes 2 0 1 }
      "YZ"  { $perm SetFilteredAxes 1 2 0 }
      "ZY"  { $perm SetFilteredAxes 2 1 0 }
    }
    $perm Update
    $currentresults ShallowCopy [ $perm GetOutput ]
    $perm Delete
}
```

Here the image input is specified in currentimage. The axes of the image are permuted using vtkImagePermute and the output stored in another instance of vtkImageData "currentresults" using the ShallowCopy command of vtkImageData. Then the vtkImagePermute object perm is deleted thus destroying the pipeline, yet maintaining the output for setting as input to the viewer at a later stage.

Assignment

- Read the Handout!
- Read the vtkImageData Man page `http://noodle.med.yale.edu/vtk/classvtkImageData.html` to get fuller sense of the functionality in vtkImageData.
- Modify, in three steps, script11-1.tcl to :
  1. Change the code to change the colorlevel and window to generate a binary looking image.
  2. Modify script12-2.tcl to show BOTH an XY and an XZ slice (i.e. two separate Actors with associate pipelines!)
  3. Modify script12-4.tcl to adjust the transparency/opacity to fade out all gray matter (the outer layer of the brain!)

Chapter 13

Transformations

Medical image analysis programming involves, for the most part, three types of objects: (i) images, (ii) surfaces and (iii) transformations. Transformations are often computed by a registration algorithm and can be used to transform a surface (vtkTransformPolyDataFilter) and to reslice an image (vtkImageReslice). VTK has support for both linear (i.e. transformations that are essentially 4×4 matrices) and non-linear warping transformations.

13.1 Introduction

The transformation class hierarchies in VTK are shown in Figures 13.1, 13.2 and 13.3. Transformations are maps which implement the following equation:

$$T : x \mapsto y, \text{ or } y = T(x) \tag{13.1}$$

where x is the input point and y is the output point. Transformations can be concatenated to yield complex maps. We can describe a compound transformation in which a point is transformed first by transformation T_1 and next by T_2 as:

$$T_1 : x \mapsto x', T_2 : x' \mapsto y, \text{ or } y = T_2(T_1(x)) \tag{13.2}$$

This style of concatenation is termed "post-multiply" in VTK – because of the order in which, in the case of linear transformations, the transformation matrices are multiplied. In the two classes which allow for concatenation of multiple transformations, namely vtkTransform and vtkGeneralTransform, the default order of concatenation is "pre-multiply". In my own work, I have never found this to be useful, and I switch these to post-multiply mode using their PostMultiply methods.

13.2 Homogeneous Linear Transformations

Representation: Linear transformations in VTK are represented internally as 4×4 matrices. This enables the use of a single operation to capture both a translation as well as a combination of rotation/shear/scale. Ordinarily, we would write such a transformation in two parts as:

$$y = Ax + b \tag{13.3}$$

where A is a 3×3 matrix that performs a combinations of rotation, scale and shear and b is a 3×1 vector specifying the translation. A more compact representation is to use homogeneous coordinates. To accomplish

this, we write each point as a 4-vector $(x_1, x_2, x_3, 1)$, and apply the transformation as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ 1 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & b_1 \\ A_{21} & A_{22} & A_{13} & b_2 \\ A_{31} & A_{32} & A_{13} & b_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} \tag{13.4}$$

This method can be used to transform all linear transformations into linear algebra operations on 4×4 matrices. This enables easy concatenation (matrix multiplication) and inversion (matrix inversion). Note also that a linear transformation can have at most 12-free parameters. There are 3 general types of linear transformations as follows:

1. Rigid – these have six parameters (3 rotations and 3 translations)
2. Similarity – these have seven parameters, rigid + overall scale factor
3. Affine – this is the general linear transformation group and has 12 parameters.

The vtkMatrix4x4 Class: The class hierarchy for all linear transformations is shown in Figure 13.2. A key helper class is the vtkMatrix4x4 class which is used to store the 4×4 matrices. This class has a number of methods, the most important of which are:

1. void SetElement (int i, int j, double value)
2. double GetElement (int i, int j) const
3. void Zero ()
4. void Identity ()
5. void DeepCopy (vtkMatrix4x4 *source)
6. double Determinant ()
7. void MultiplyPoint (const float in[4], float out[4])

An example of directly using vtkMatrix4x4 is presented in the script below (script13-1.tcl)

```
lappend auto_path [ file dirname [ info script ]]
wm withdraw .
package require newname

set mat [ vtkMatrix4x4 [ newname::vnewobj ]]

$mat Identity
$mat SetElement 0 0 0
$mat SetElement 0 1 1.0
$mat SetElement 1 1 0.0
$mat SetElement 1 0 1.0

for { set i 0 } { $i <= 3 } { incr i } {
  puts -nonewline stdout "\["
  for { set j 0 } { $j <= 3 } { incr j } {
    puts -nonewline stdout "[ $mat GetElement $i $j] "
  }
  puts stdout "\]"
}
```

This script creates the matrix, modifies a small part and prints its contents.

The vtkTransform Class: This complex class has a variety of functionality for implementing linear transformations. It has a vtkMatrix4x4 member which stores the current matrix, and allows for the concatenation of various operations, either in pre-multiply or post-multiply order. The most important of its methods are:

- void Identity ()
- void PostMultiply ()
- void Inverse ()
- vtkLinearTransform * GetLinearInverse ()
- void SetMatrix (vtkMatrix4x4 *matrix)
- void Concatenate (vtkMatrix4x4 *matrix)
- void Concatenate (vtkLinearTransform *transform)
- void Translate (double x, double y, double z)
- void RotateWXYZ (double angle, double x, double y, double z)
- void RotateX (double angle)
- void RotateY (double angle)
- void RotateZ (double angle)
- void Scale (double x, double y, double z)

Typically, when using vtkTransform, we first set it to Identity and then invoke the PostMultiply method. Then a series of transformations can be concatenated to yield a single compound linear transformations. This concatenations can be either explicit (using the Concatenate methods) or implicit (using the Rotate, Translate and Scale methods). An example is shown in the script below (script13-2.tcl):

```
set tr [ vtkTransform [ newname::vnewobj ]]
$tr Identity
$tr PostMultiply
$tr Scale 2.0 2.0 1.0
$tr Translate 1.0 0.0 -3
set out [ $tr TransformPoint 4 2 1 ]
puts stdout "(4 2 1) --> ($out)"
```

The TransformPoint method is the most basic method of all transformation classes. It is originally defined in vtkAbstactTransform and overridden as needed by derived classes.

The vtkLandmarkTransform class: This extremely useful class contains functionality for the least squares estimation of a linear transformation from two sets of corresponding points. Given two point sets, each having n points: $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ the operation implemented by this class can be written mathematically as:

$$\hat{T} = \underset{T}{\arg\min} \sum_{i=1}^{n} |x_i - y_i|^2 \tag{13.5}$$

where T is a linear transformation. The key methods of vtkLandmarkTransform are:

- void SetSourceLandmarks (vtkPoints *points)
- void SetTargetLandmarks (vtkPoints *points)
- void SetModeToRigidBody ()
- void SetModeToSimilarity ()
- void SetModeToAffine ()

The last three specify the exact form of T which can be restricted to be rigid, similarity or the full affine transformation. Its usage is illustrated in the following script (script13-3.tcl). First we define the two sets of points:

```
set pts1 [ vtkPoints [ newname::vnewobj ]]
$pts1 SetNumberOfPoints 5
$pts1 SetPoint 0 0.0 0.0 0.0; $pts1 SetPoint 1 1.0 0.0 0.0
$pts1 SetPoint 2 0.0 1.0 0.0; $pts1 SetPoint 3 0.0 0.0 1.0
$pts1 SetPoint 4 1.0 1.0 1.0

set pts2 [ vtkPoints [ newname::vnewobj ]]
$pts2 SetNumberOfPoints 5
$pts2 SetPoint 0 0.0 0.1 0.0; $pts2 SetPoint 1 2.0 0.1 0.0
$pts2 SetPoint 2 0.0 1.1 0.0; $pts2 SetPoint 3 0.0 0.1 3.0
$pts2 SetPoint 4 2.0 1.1 3.0
```

Next we create a vtkLandmarkTransform object, set the two point sets and the transformation type and invoke the Update method to compute the transformation:

```
set land [ vtkLandmarkTransform [ newname::vnewobj ]]
$land SetSourceLandmarks $pts1
$land SetTargetLandmarks $pts2
$land SetModeToAffine
$land Update
```

Once this is done, we print the output, as before:

```
set mat [ $land GetMatrix ]
puts stderr "Fitting Output is:"
for { set i 0 } { $i <= 3 } { incr i } {
  puts -nonewline stdout "\["
  for { set j 0 } { $j <= 3 } { incr j } {
        puts -nonewline stdout "[ format "%+5.2f" [ $mat GetElement $i $j]] "
    }
  puts stdout "\]"
}
```

13.3 Non-Linear Transformations

Non-linear transformations are loosely defined as those transformations which can not be expressed as a 4×4 matrix. The family of nonlinear transformations defined in VTK is shown in Figure 13.3. The most useful ones are vtkGridTransform and vtkThinPlateSplineTransform.

vtkThinPlateSplineTransform: Kernel-based transformations, such as the thin-plate spline transform, are based on (i) two sets of corresponding points (landmarks) which the transformation maps exactly, and (ii) an interpolation rule (kernel) which is used to 'sensibly' interpolate between these landmark points. The thin-plate spline transform, popularized by Bookstein in the early 90's, is one of the most common forms of this type of transformation. Its most useful methods are:

- void SetBasisToR ()
- void SetBasisToR2LogR ()
- void SetSourceLandmarks (vtkPoints *source)
- void SetTargetLandmarks (vtkPoints *target)

The general usage – the syntax is very similar to vtkLandmarkTransform – takes the form:

```
set tps [ vtkThinPlateSplineTransform [ newname::vnewobj ]]
$tps SetSourceLandmarks $pts1
$tps SetTargetLandmarks $pts2
$tps SetBasisToR
$tps Update
```

For 2D transformations use the SetBasisToR2LogR method instead, to select the appropriate basis function (kernel).

vtkGridTransform: The most general type of non-linear transformation is one explicitly specified by a dense displacement field. VTK provides the vtkGridTransform class for this purpose. Its key methods are:

- virtual void SetDisplacementGrid (vtkImageData *)
- virtual void SetDisplacementScale (double)
- virtual void SetDisplacementShift (double)
- void SetInterpolationModeToNearestNeighbor ()
- void SetInterpolationModeToLinear ()
- void SetInterpolationModeToCubic ()

The displacements are specified on a grid stored in a vtkImageData structure – this must have three components (frames) which store the displacements in x, y, and z respectively. These displacements can be, optionally, scaled by a scale factor (specified using SetDisplacementScale) and shifted by a translation term (SetDisplacementShift). The only additional parameter is the interpolation mode which specifies how this displacement field is to be interpreted. (An internal extension also provides for a B-Spline tensor grid interpolation method). The following snippet describes the basic usage:

```
set img [ vtkImageData [ newname::vnewobj ]]
$img SetDimensions 10 10 10
$img SetOrigin 0.0 0.0 0.0
$img SetSpacing  10.0 10.0 10.0
$img SetNumberOfScalarComponents 3
$img SetScalarTypeToDouble
$img AllocateScalars

set data [ [ $img GetPointData ] GetScalars ]
$data FillComponent 0 0.0
$data FillComponent 1 0.0
$data FillComponent 2 0.0

# Set some displacements much like image intensities

# Create Grid
set grid [ vtkGridTransform [ newname::vnewobj ]]
```

```
$grid SetInterpolationModeToLinear
$grid SetDisplacementGrid $img
$img Delete
```

The image is created and manipulated as usual. Each voxel contains three components (its x,y and z displacement). Next, the grid transform is created and the displacement field image is attached to it.

13.4 The vtkGeneralTransform

The vtkGeneralTransform class allows for the concatenation of a number of transformations into a single transformation. These transformations (unlike similar functionality in vtkTransform) can be both linear and/or *non-linear*. The syntax is very simple as illustrated in the following code snippet:

```
set gen [ vtkGeneralTransform [ newname::vnewobj ]]
$gen PostMultiply
$gen Concatenate $t1
$gen Concatenate $t2
$gen Concatenate $t3
```

where $t1,$t2 and $t3 are existing transformations.

13.5 Transforming Surfaces

Surfaces are most easily transformed using the vtkPolyDataFilter. A short code snippet illustrates its use:

```
set tr [ vtkTransform [ newname::vnewobj ]]
$tr Translate 10 5 0

set tf vtkTransformPolyDataFilter [ newname::vnewobj ]]
$tf SetInput $poly
$tf SetTransform $tr
$tf Update

set output [ vtkPolyData [ newname::vnewobj ]]
$output ShallowCopy [ $tf GetOutput ]
$tf Delete
```

The vtkTransformPolyDataFilter class takes two inputs: (i) an input surface and (ii) a transformation. In this case, the script assumes an input surface stored in the variable $poly. While a linear transformation is created, a non-linear transformation can also be specified. The output is also of type vtkPolyData. The filter essentially transforms the points of the input surface one-by-one and stores the result in the output surface.

13.6 Reslicing Images

Reslicing images is at the heart of most image registration procedures. While transforming surfaces is intuitive, and can be summarized in the three steps (i) take point, (ii) transform point and (iii) store point, image reslicing is somewhat counter-intuitive.

We will explain the process with reference to figure 13.4. In a typical registration process we have a Reference image and a Target image. The registration estimates the transformation *FROM* the Reference image *TO* the target image. This transformation can then be used in an image-reslicing operation to warp the Target image *BACK* to the Reference image, i.e. make the target look like the reference. In this way, while the transformation is "forward" the image moves **BACKWARDS**.

The process can be described by the following recipe, once the transformation exists:

- Create an empty image (often having the same geometry as a Reference Image).
- For each point (voxel) r in the empty image:
  1. Compute its corresponding point in the Target image r', $T : r \mapsto r'$.
  2. Interpolate the target image to find the image intensity I at position r' – which rarely corresponds to an exact voxel.
  3. Set the voxel r in the empty reference image to have intensity I.
- Repeat for all voxels.

VTK has the very powerful vtkImageReslice class for performing various image-reslicing operations. An example is shown in the script (script13-4.tcl) below: First we load an image:

```
set reader [ vtkTIFFReader [ newname::vnewobj ]]
$reader SetFileName brain.tif
$reader Update
```

Next we specify a rotation. All rotations are by default centered at the origin. In order to get a rotation centered at the center of the image (in this case 62.0,80.0,0.0) we first translate the center of the image to the origin, apply the rotation, and translate back:

```
set xform [ vtkTransform [ newname::vnewobj ]]
$xform PostMultiply
$xform Translate -62.0 -80.0 0
$xform RotateWXYZ 20 0 0 1
$xform Translate 62.0 80.0 0.0
```

Next the heart of the script, the reslicing. vtkImageReslice has a ton of options; we present here one of the two most common ways of using it. We specify the "size" of the empty image using the three statements SetOutputExtent, SetOutputOrigin and SetOutputSpacing. The last two should be obvious: they specify the position of the '0'-th voxel and the distance between voxel centroids. The SetOutputExtent method specifies the size of the empty image. The most common usage has the form:

SetOutputExtent 0 width-1 0 height-1 0 depth-1

Next we specify the input image (the moving image, the image to be resliced), the transformation and the interpolation mode:

```
set resl [ vtkImageReslice [ newname::vnewobj ]]
$resl SetOutputExtent 0 199 0 199 0 0
$resl SetOutputOrigin -20.0 -20.0 0.0
$resl SetOutputSpacing 1.0 1.0 1.0
$resl SetInput [ $reader GetOutput ]
$resl SetResliceTransform $xform
$resl SetInterpolationModeToLinear
$resl Update
```

Finally, we save the output to a jpeg file:

```
set writer [ vtkJPEGWriter [ newname::vnewobj ]]
$writer SetInput [ $resl GetOutput ]
$writer SetFileName rotated.jpg
$writer Write
```

An alternative way to specify the size of the empty image is to "clone" an existing image. This uses the SetInformationInput method of vtkImageReslice – for an example see script13-5.tcl. The heart of the script is:

```
set resl [ vtkImageReslice [ newname::vnewobj ]]
$resl SetInformationInput [ $reader GetOutput ]
$resl SetInput [ $reader GetOutput ]
$resl SetResliceTransform $xform
$resl SetInterpolationModeToLinear
$resl Update
```

where in this case the output image is set to have the same dimensions, origin and spacing as the input image, for convenience. In a typical registration application, the reference image is used as the InformationInput and the target image as the Input, with the registration transformation being used to set the ResliceTransform.

Assignment

- Read the Handout!
- Create a sphere surface (see script14-2.tcl for an example). Transform it using a linear transform (e.g. scale + translate). Display both the original and the transformed surfaces as two separate actors in the same scene. For good measure set the color of the transformed actor to red.
  Hint: To set the color of an actor to red: do [$act GetProperty] SetColor 1.0 0.0 0.0
- Create a ThinPlateSplineTransform. Use as source landmarks the 6 corners of a cube with corners (0,0,0) and (100,100,100). Use the same points as targets but move one corner (e.g. 100,100,100) to (75,75,75). Replace the linear transform in the script above with this non-linear transform and display both the original and transformed surfaces.
- Using script13-4.tcl as a guide, try to add a scaling to the transformation (hint do it before the rotation) and save the output image as a pnm file (use the vtkPNMWriter).

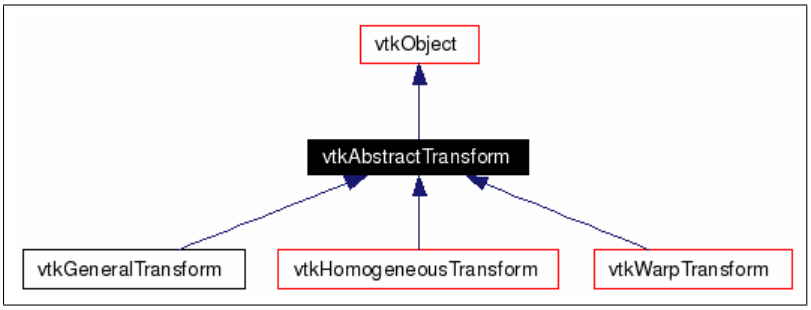


Figure 13.1: The top-level transformation hierarchy in VTK 4.4. The basic interface to all transformations is captured in the abstract superclass vtkAbstractTransform. The transformations are divided into three branches. The first group is the linear transformations which are derived from vtkHomogeneousTransform. The second group is the non-linear transformations derived from vtkWarpTransform. The final group is the vtkGeneralTransform.

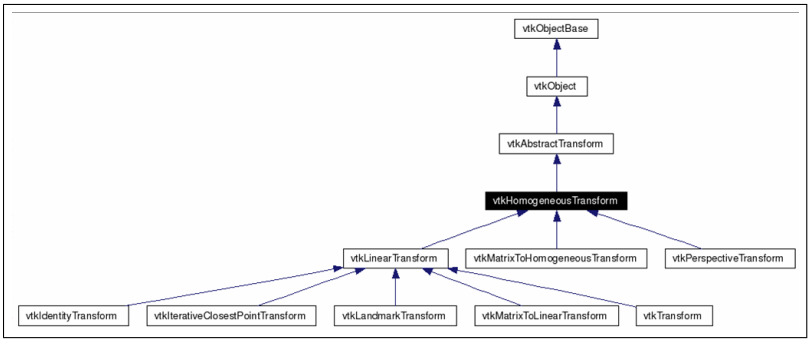


Figure 13.2: The linear transformation hierarchy derived from vtkHomogeneousTransform. The useful ones are on the bottom row. vtkTransform can be used to manually specify a combination of translations/rotations and scales. vtkMatrixToLinearTransform is used to create a linear transformation from a vtkMatrix4x4 object. vtkLandmarkTransform creates a transformation by computing a least squares fit between two sets of corresponding points. Finally, the vtkIterativeClosestPointTransform is an implementation of ICP transform.

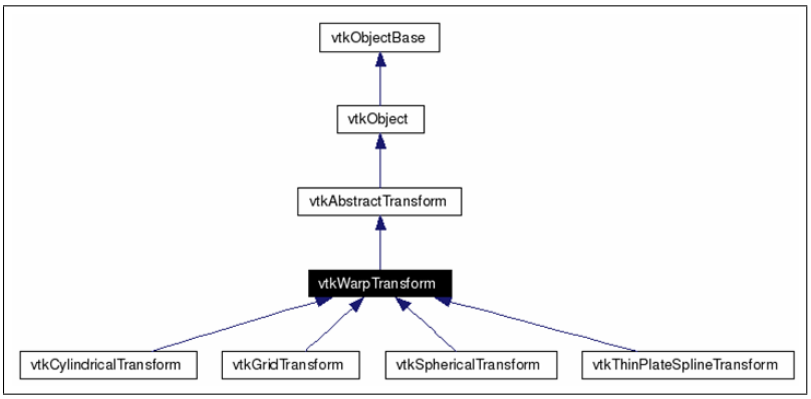


Figure 13.3: The non-linear transformation hierarchy derived from vtkWarpTransform. vtkGridTransform is a transformation defined as a displacement field, whereas vtkThinPlateSpline transform implements a transformation defined by two sets of corresponding points and an interpolation function.

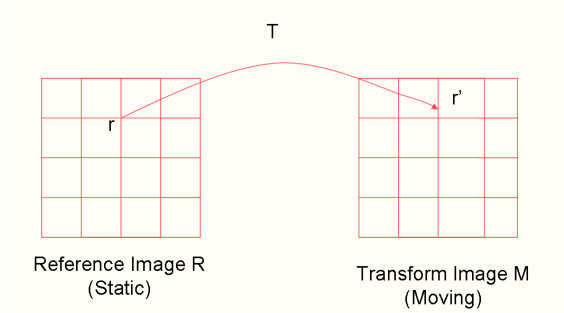


Figure 13.4: The Image Reslicing Process as implemented in vtkImageReslice

Chapter 14

Some Additional VTK Classes

14.1 Introduction

The Visualization Toolkit is a large object-oriented library that has a touch under 1000 classes. Discovering what functionality is provided by these classes, and how to best take advantage of it, is frequently overwhelming to the beginner. The goal of this Chapter is to (i) highlight some classes that are particularly useful and do not fit into the previous Chapters and (ii) to provide a brief reference guide/roadmap .

14.2 Some Additional Useful Classes

The vtkProbeFilter class: An often overlooked class in VTK is the vtkProbeFilter class. This *extremely* powerful class allows us to sample a data object (e.g. an image) at arbitrary locations and return the values (e.g. intensities) at these locations.

The ProbeFilter class takes two inputs: (i) the Source – this is the dataset to be sampled and (ii) the Input – this is the dataset whose points define the sampling array. The output is of the same type as the input. The sampled values are stored in the scalar array of the output's pointdata as illustrated in the following script (script14-1.tcl).

First we load an image, and create a surface consisting of the sampling points (four in this case). Note that no topology is specified for the surface, in this case, as it is not needed for the purposes of the ProbeFilter:

```
set reader [ vtkTIFFReader [ newname::vnewobj ]]
$reader SetFileName brain.tif
$reader Update

set pts [ vtkPoints [ newname::vnewobj ]]
$pts SetNumberOfPoints 4
$pts SetPoint 0 45 25 0
$pts SetPoint 1 61.1 98.3 0
$pts SetPoint 2 82.2 36.1 0
$pts SetPoint 3 4.2 11.6 0

set polydata [ vtkPolyData [ newname::vnewobj ]]
$polydata SetPoints $pts
$pts Delete
```

Next we apply the probe filter to sample the image at the 4 locations:

```
set probe [ vtkProbeFilter [ newname::vnewobj ]]
$probe SetInput $polydata
$probe SetSource [ $reader GetOutput ]
$probe Update
```

Next we print the output intensities:

```
set out [ $probe GetOutput ]
puts stderr "Out is a [ $out GetClassName ], points = [ $out GetNumberOfPoints ]"

set data [ [ $out GetPointData ] GetScalars ]

for { set i 0 } { $i < 4 } { incr i } {
    set pt [ $out GetPoint $i ]
    set val [ $data GetComponent $i 0 ]
    puts stderr "Point index = $i\t Location $pt \t\t Value=$val"
}
```

This filter (vtkProbeFilter) can be extremely useful in computing deformable surface segmentations, for example, where we need to obtain the value of the image (or it's gradient) at a fixed set of locations.

The vtkPointLocator class: The point locator class can be used to quickly locate points in 3D. It divides the space into a regular array of hexahedral buckets and it keeps a list of points that lie in each bucket. The most common operation involves giving a position in 3D and finding the closest point to it, which is illustrated in the following script (script14-2.tcl).

First we create a set of points – in this case using the vtkSphereSource object:

```
}
set sphere [ vtkSphereSource [ newname::vnewobj ]]
$sphere SetCenter 50.0 50.0 50.0
$sphere SetRadius 20.0
$sphere SetPhiResolution 24
$sphere SetThetaResolution 24
$sphere Update
```

Next we create the point locator object itself:

```
}
# Create Locator
set locator [ vtkPointLocator [ newname::vnewobj ]]
```

```
$locator SetDataSet [ $sphere GetOutput ]
$locator BuildLocator
```

Finally, we use the locator to find the nearest points in the sphere to the testpoints defined in the list 'sometestpoints':

```
}
# Define Some Test Points and search:
set sometestpoints { { 42 23 48 } { 81 75 32 } { 50 62 73 } }
set len [ llength $sometestpoints ]
for { set i 0 } { $i < $len } { incr i } {
    set plist [ lindex $sometestpoints $i ]
    set x [ lindex $plist 0 ]
    set y [ lindex $plist 1 ]
    set z [ lindex $plist 2 ]
    set index [ $locator FindClosestPoint $x $y $z  ]
    set outpt [ [ $locator GetDataSet ] GetPoint $index ]

    puts stderr "Input Point = ($x,$y,$z) -->
                 nearest point on sphere index =$index, location = $outpt"
}
```

A more advanced use of the class is to find the closest N points to a search point. This is illustrated below:

```
set idlist [ vtkIdList [ newname::vnewobj ]]
$locator FindClosestNPoints  4 70 70 70 $idlist

set numids [ $idlist GetNumberOfIds ]
for { set i 0 } { $i < $numids } { incr i } {
    set index [ $idlist GetId $i ]
    set outpt [ [ $locator GetDataSet ] GetPoint $index ]
    puts stderr "Closest Point $i, index =$index, location = $outpt"
}
```

The indices of the closest N points ($N = 4$ in this case) are stored in a vtkIdList object. vtkIdList is used to represent and pass data id's between objects. vtkIdList may represent any type of integer id, but usually represents point and cell ids. It has a very simple interface that is similar 'in-spirit' to vtkDataArray. The one additional feature is the InsertUniqueId method that will only insert an integer, if this does not already exist in the list.

The vtkCollection class: The vtkCollection class can be used to create and manipulate unsorted lists of objects. The lists allow duplicate entries. vtkCollection also serves as a base class for lists of specific types of objects. It's children classes are: vtkActorCollection vtkAssemblyPaths vtkDataSetCollection vtkImplicitFunctionCollection vtkLightCollection vtkPolyDataCollection vtkRenderWindowCollection vtkRendererCollection vtkStructuredPointsCollection vtkTransformCollection vtkVolumeCollection.

The basic interface to all vtkCollection-derived classes is the same (the beauty of OOP) and is, naturally, defined

in vtkCollection itself. The objects stored in a Collection are called "items". The key methods are:

- int GetNumberOfItems () – get the total number of items
- int IsItemPresent (vtkObject *object) – check for the presence of item object.
- void RemoveAllItems () – empty the collection.
- void AddItem (vtkObject *object) – adds a new object to the collection. Derived classes specialize this method to the specific type. For example in vtkPolyDataCollection this takes the form: AddItem (vtkPolyData *pd)
- void ReplaceItem (int i, vtkObject *object) – replace the item at position i with a new object.
- void RemoveItem (int i) – remove item at position i.
- void RemoveItem (vtkObject *object) – remove the object "object" if it exists.
- vtkObject * GetItemAsObject (int i) – get the object at position i (slower).
- void InitTraversal () – initialize the list for quick traversal.
- vtkObject * GetNextItemAsObject () – get the next object and increment the pointer.

An example using a vtkCollection object to store three different Data Arrays is shown below (script14-3.tcl). First we create the three arrays (of different types and sizes!):

```
# Create three arrays
set data1 [ vtkFloatArray [ newname::vnewobj ]]
$data1 SetNumberOfTuples 2
$data1 SetComponent 0 0 10.0; $data1 SetComponent 1 0 20.0

set data2 [ vtkShortArray [ newname::vnewobj ]]
$data2 SetNumberOfTuples 3
$data2 SetComponent 0 0 1; $data2 SetComponent 1 0 2; $data2 SetComponent 2 0 3

set data3 [ vtkDoubleArray [ newname::vnewobj ]]
$data3 SetNumberOfTuples 10
for { set i 0 } { $i < 10.0 } { incr i } {
    $data3 SetComponent $i 0 [ expr $i*$i ]
}
```

Next we create the collection and add the arrays to it:

```
set col [ vtkCollection [ newname::vnewobj ]]
$col AddItem $data1
$col AddItem $data2
$col AddItem $data3
```

Finally we traverse the collection by first reseting the iterator pointer to the start (InitTraversal). Then the GetNextItemAsObject method is called succesively to obtain each object in turn:

```
puts stdout "Traversing the collection"
set nitems [ $col GetNumberOfItems ]
$col InitTraversal
for { set i 0 } { $i < $nitems } { incr i } {
```

```
    set arr [ $col GetNextItemAsObject ]
    puts stderr "Array $i type [ $arr GetClassName ], num tuples [ $arr GetNumberOfTuples ]"
}
```

14.3 100 Useful Classes

The VTK Class List provides an annotated list of all VTK classes. This can be accessed at: `http://noodle.med.yale.edu/vtk/annotated.html`. In this section, I highlight some of these classes:

1. **vtkActorCollection:** List of actors. This can be useful for storing groups of actors to show/hide together.
2. **vtkAppendPolyData:** Appends one or more polygonal datasets together. This is extremely useful for forming complex objects consisting of many polygonal pieces. It is often preferable to combine the objects at this level as opposed to have multiple actors.
3. **vtkApproximatingSubdivisionFilter:** Generate a subdivision surface using an Approximating Scheme.
4. **vtkAssembly:** Create hierarchies of vtkProp3Ds (transformable props). This can be used to combine multiple actors into a single one.
5. **vtkCallbackCommand:** Supports function callbacks. This is exceedingly useful in large programs.
6. **vtkCamera:** Virtual camera for 3D rendering. This is used by all vtkRenderer classes.
7. **vtkCell:** Abstract class to specify cell behavior – this is worth looking at if you are interested in finite element meshes and arbitrary topologies.
8. **vtkCellArray:** Object to represent cell connectivity
9. **vtkCleanPolyData:** Merge duplicate points, and/or remove unused points and/or remove degenerate cells. This is very useful for downsampling a surface to generate a set of points to use as an input to a point-based registration method. It does "destroy" the surface structure.
10. **vtkCollection:** Create and manipulate unsorted lists of objects. This is very useful in large programs, for grouping a set of objects together.
11. **vtkConnectivityFilter:** Extract data based on geometric connectivity.
12. **vtkContourFilter:** Generate isosurfaces/isolines from scalar values. This can be used to extract, for example, zero-levelsets from a levelset propagation algorithm.
13. **vtkCurvatures:** Compute curvatures (Gauss and mean) of a Polydata object.
14. **vtkDataArray:** Abstract superclass for arrays. This is the parent class for all data arrays and it is worth being familar with it's structure.
15. **vtkDataObject:** General representation of visualization data
16. **vtkDataSet:** Abstract class to specify dataset behavior
17. **vtkDecimate:** Reduce the number of triangles in a mesh
18. **vtkDICOMImageReader:** Reads DICOM images. This is not a complete implementation but useful nonetheless.
19. **vtkExtractVOI:** Select piece (e.g., volume of interest) and/or subsample structured points dataset
20. **vtkGaussianSplatter:** Splat points into a volume with an elliptical, Gaussian distribution
21. **vtkGeneralTransform:** Allows operations on any transforms. This is very useful for concatenating transformations of different types into a single transformation. Note that for proper operationg the "PostMultiply" flag needs to be set.
22. **vtkGeometryFilter:** Extract geometry from data (or convert data to polygonal type)
23. **vtkGridTransform:** Nonlinear warp transformation. This stores a transformation as a displacement field (with either linear or cubic interpolation).
24. **vtkHeap:** Replacement for malloc/free and new/delete
25. **vtkHull:** Produce an n-sided convex hull
26. **vtkIdentityTransform:** Transform that doesn't do anything. This may sound useless, but if you have to have a transformation somewhere that is identity, this is the fastest way to do it!
27. **vtkIdList:** List of point or cell ids
28. **vtkIdListCollection:** Maintain an unordered list of dataarray objects. This can be useful for getting a list of unique indices for example.

29. **vtkImageAccumulate:** Generalized histograms up to 4 dimensions.
30. **vtkImageAnisotropicDiffusion3D:** Edge preserving smoothing. This is worth looking at.
31. **vtkImageAppend:** Collects data from multiple inputs into one image. The AppendAxis is used to define the direction of "stiching".
32. **vtkImageAppendComponents:** Collects components from two inputs into one output.
33. **vtkImageBlend:** Blend images together using alpha or opacity.
34. **vtkImageCast:** Image Data type Casting Filter
35. **vtkImageConvolve:** Convolution of an image with a kernel
36. **vtkImageCorrelation:** Correlation imageof the two inputs
37. **vtkImageData:** Topologically and geometrically regular array of data
38. **vtkImageExport:** Export VTK images to third-party systems. This basically gets you a raw pointer that you can use to access the data.
39. **vtkImageExtractComponents:** Outputs a single component
40. **vtkImageFFT:** Fast Fourier Transform
41. **vtkImageFlip:** This flips an axis of an image. Right becomes left ..
42. **vtkImageGaussianSmooth:** Performs a gaussian convolution
43. **vtkImageGradient:** Computes the gradient vector
44. **vtkImageGradientMagnitude:** Computes magnitude of the gradient
45. **vtkImageImport:** Import data from a C array. This is useful for integrating with legacy code.
46. **vtkImageLaplacian:** Computes divergence of gradient
47. **vtkImageMagnitude:** Colapses components with magnitude function.
48. **vtkImageMarchingCubes:** Generate isosurface(s) from volume/images
49. **vtkImageMask:** Combines a mask and an image
50. **vtkImageMathematics:** Add, subtract, multiply, divide, invert, sin, cos, exp, log
51. **vtkImageMedian3D:** Median Filter
52. **vtkImageNonMaximumSuppression:** Performs non-maximum suppression. This is part of the implementation of the Canny edge detection filter.
53. **vtkImageRFFT:** Reverse Fast Fourier Transform
54. **vtkImageSeedConnectivity:** SeedConnectivity with user defined seeds
55. **vtkImageSeparableConvolution:** 3 1D convolutions on an image
56. **vtkImageShiftScale:** Shift and scale an input image. This also allows for changing the image type from e.g. short to float.
57. **vtkLabeledDataMapper:** Draw text labels at dataset points. This is useful for automatically numbering points.
58. **vtkLandmarkTransform:** Linear transform specified by two corresponding point sets
59. **vtkLookupTable:** Map scalar values into colors via a lookup table
60. **vtkMarchingContourFilter:** Generate isosurfaces/isolines from scalar values
61. **vtkMarchingCubes:** Generate isosurface(s) from volume
62. **vtkMath:** Performs common math operations. This is a good example of integrating procedural code into VTK. All member methods of this class are static. Pi is also defined here in a cross-platform manner.
63. **vtkMatrix4x4:** Represent and manipulate 4x4 transformation matrices
64. **vtkOutlineFilter:** Create wireframe outline for arbitrary data set
65. **vtkPCAAnalysisFilter:** Performs principal component analysis of a set of aligned pointsets
66. **vtkPlane:** Perform various plane computations
67. **vtkPlaneSource:** Create an array of quadrilaterals located in a plane
68. **vtkPNMReader:** Read pnm (i.e., portable anymap) files
69. **vtkPNMWriter:** Writes PNM (portable any map) files
70. **vtkPoints:** Represent and manipulate 3D points
71. **vtkPolyData:** Concrete dataset represents vertices, lines, polygons, and triangle strips
72. **vtkPolyDataConnectivityFilter:** Extract polygonal data based on geometric connectivity
73. **vtkPolyDataMapper:** Map vtkPolyData to graphics primitives
74. **vtkPolyDataNormals:** Compute normals for polygonal mesh
75. **vtkPolyDataReader:** Read vtk polygonal data file
76. **vtkPolyDataWriter:** Write vtk polygonal data

77. **vtkPostScriptWriter:** Writes an image as a PostScript file
78. **vtkPriorityQueue:** List of ids arranged in priority order
79. **vtkProbeFilter:** Sample data values at specified point locations. This is *extremely useful* for sampling images at arbitrary locations, e.g. computing line integrals using surfaces and or curves for the implementation of deformable model segmentation.
80. **vtkProcrustesAlignmentFilter:** Aligns a set of pointsets together
81. **vtkRenderer:** Abstract specification for renderers
82. **vtkRenderWindow:** Create a window for renderers to draw into
83. **vtkScalarBarActor:** Create a scalar bar with labels
84. **vtkTextActor:** An actor that displays text. Scaled or unscaled
85. **vtkTextMapper:** 2D text annotation
86. **vtkTexture:** Handles properties associated with a texture map
87. **vtkThinPlateSplineTransform:** Nonlinear warp transformation. This is used in many non-rigid registration applications.
88. **vtkTkImageViewerWidget:** Tk Widget for viewing vtk images
89. **vtkTkRenderWidget:** Tk Widget for vtk renderering
90. **vtkTransform:** Describes linear transformations via a 4x4 matrix
91. **vtkTransformFilter:** Transform points and associated normals and vectors
92. **vtkTransformPolyDataFilter:** Transform points and associated normals and vectors for polygonal dataset
93. **vtkTriangleFilter:** Create triangle polygons from input polygons and triangle strips
94. **vtkUnstructuredGrid:** Dataset represents arbitrary combinations of all possible cell types. This is useful for representing meshes.
95. **vtkVolume:** Volume (data & properties) in a rendered scene
96. **vtkVolumeRayCastCompositeFunction:** Ray function for compositing
97. **vtkVolumeRayCastMapper:** A slow but accurate mapper for rendering volumes
98. **vtkVolumeTextureMapper2D:** Abstract class for a volume mapper
99. **vtkVRMLExporter:** Export a scene into VRML 2.0 format
100. **vtkWindowLevelLookupTable:** Map scalar values into colors or colors to scalars; generate color table

Part IV

Interfacing To BioImage Suite using Tcl

Chapter 15

Leveraging BioImage Suite Components

In this Chapter and the one following we describe how to leverage code already in the Yale BioImage Suite package for your own projects. In this Chapter, we focus on individual components, whereas in the next Chapter we will talk about how to use the BioImage Suite application framework to write your own "BioImage Suite application". More information about BioImage Suite can be found at its web-page `www.bioimagesuite.org`.

15.1 Introduction

BioImage Suite is our home grown Medical Image Analysis Utility. It uses a combination of Tcl/[Incr Tcl] and C++ and leverages both VTK and ITK fairly substantially. For more information see `www.bioimagesuite.org`. BioImage Suite has around 300,000 lines of code and in it are a number of key classes and components which can be usefully leveraged to simplify the tasks of implementing your own software. This can be accomplished by loading the core BioImage Suite libraries as extensions into the vtk interpreter and using classes defined in the BioImage Suite source tree directly from Tcl.

The package `loadbioimagesuite.tcl` loads the BioImage Suite extensions into your current vtk shell. The package assumes that BioImage Suite is installed in the default locations which are: `/usr/local/bioimagesuite` – on Linux/Mac OS X and `c:/yale/bioimagesuite` on Windows. These can be overridden by either directly editing the loadbioimagesuite.tcl file or by setting the environment variable BIOIMAGESUITE.

Note: BioImage Suite defines the function `pxvtable::vnewobj` which is mostly equivalent to the `newname::vnewobj` convention we have been previously using. Both can be used inter-changeably. For examples leveraging BioImage Suite code we will, for the most part, stick to using `pxvtable::vnewobj` to generate unique object names.

15.2 Loading and Saving Analyze Formatted Images

The first example demonstrates the use of the class vtkpxAnalyzeImageSource and vtkpxAnalyzeImageWriter which can be used to read and write analyze images into the "old" Analyze image format that is in common use. The following script (script15-1.tcl) demonstrates this:

```
lappend auto_path [ file dirname [ info script ]]
wm withdraw  .
package require loadbioimagesuite 1.0
```

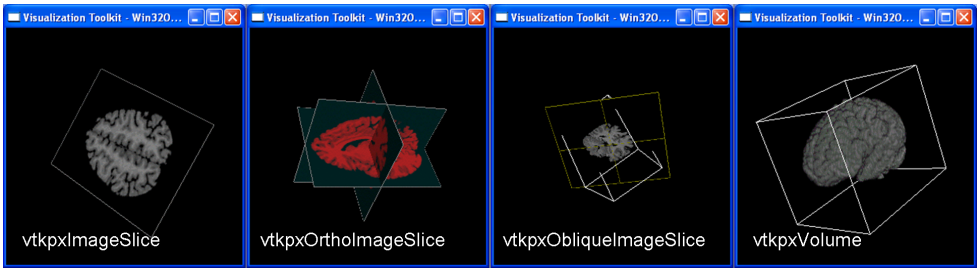


Figure 15.1: . The four specialized display objects in BioImage Suite.

```
set ana [ vtkpxAnalyzeImageSource [ pxvtable::vnewobj ]]
$ana Load axial.hdr
set img [ $ana GetOutput ]
set orient [ $ana GetOrientation ]

set shift [ vtkImageShiftScale [ pxvtable::vnewobj ]]
$shift SetInput $img
$shift SetScale -1; $shift SetShift 0
$shift Update

set anaw [ vtkpxAnalyzeImageWriter [ pxvtable::vnewobj ]]
$anaw SetInput [ $shift GetOutput ]
$anaw SetOrientation $orient
$anaw Save inverted.hdr
```

The image is first loaded in using the vtkpxAnalyzeImageSource object. The interface to this is very simple as shown in the script. It has two outputs (i) the usual Output (GetOutput) which is the image and an integer (0:3) which is the orientation of the image (0=Axial,1=Coronal,2=Sagittal,3=Other).

Next this is contrast inverted using an instance of vtkImageShiftScale. The inverted image is saved using an instance of vtkpxAnalyzeImageWriter. This last class takes three inputs: the input image, the input orientation and the filename which is specified as an argument to the Save method.

15.3 Some Specialized Display Objects

BioImage Suite defines the following classes for specialized and streamlined image display:

- vtkpxImageSlice – this takes as inputs, an image, an orientation, a slice number and a frame and displays a texture-mapped slice on a plane.
- vtkpxOrthoImageSlice – this is a combination of three vtkpxImageSlice objecs that display the three orthogonal slices simultaneously.
- vtkpxObliqueSlice – this class displays an oblique slice through an image
- vtkpxVolume – this is a derived class from vtkVolume and it streamlines the volume rendering pipeline.

These are shown in Figure 15.1.

The vtkpxImageSlice class: This displays a single image slice appropriately oriented in space. It's use is illustrated in the following script (script15-4.tcl). Compare this to script12-2.tcl to appreciate the encapsulation of the whole image slice pipeline into a single class.

We first load the image.

```
set tr [ vtkpxAnalyzeImageSource [ pxvtable::vnewobj ]]
$tr Load axial.hdr
```

Then we create the imageslice object and specify: (i) the input image, (ii) the frame – in the case of 4D images, (iii) the current plane (2=XY,1=XZ,0=YZ), (iv) the level which is the slice number in the orientation selected by the plane and (v) the display mode (0=nothing,1=image only,2 = rectangle only, 3= image +rectangle). set imageslice [vtkpxImageSlice [pxvtable::vnewobj]]

```
$imageslice SetInput [ $tr GetOutput ]
$imageslice SetFrame 0
$imageslice SetCurrentPlane 2
$imageslice SetLevel 100
$imageslice SetDisplayMode 3
```

The rest is pretty boring stuff, by now, and included in outline form. Note that vtkpxImageSlice functions like an actor and can be added to a renderer directly.

```
set ren [ vtkRenderer [ pxvtable::vnewobj ]]
$ren AddActor $imageslice

set renwin [ vtkRenderWindow [ pxvtable::vnewobj ] ]
...
set iren [ vtkRenderWindowInteractor [ pxvtable::vnewobj ] ]
....
```

The output of this script is shown in Figure 15.1(left). In addition the lookup table can be customized using the SetLookupTable method which takes a vtkLookupTable as it's argument.

The vtkpxOrthoImageSlice class: This displays three orthogonal slices in space. It consists of 3 vtkpxImageSlice objects and it's use is illustrated by script15-5.tcl. The key code is:

```
set dim [ [ $tr GetOutput ] GetDimensions ]
set x [ expr round([ lindex $dim 0 ] /2)]
set y [ expr round([ lindex $dim 1 ] /2)]
set z [ expr round([ lindex $dim 2 ] /2)]

set colormap [ vtkLookupTable [ pxvtable::vnewobj ]]
$colormap SetNumberOfColors 256
$colormap SetTableRange 0 255
for { set i 0 } { $i  < 256 } { incr i } {
    $colormap SetTableValue $i [ expr $i/255.0] 0.2 0.2 1.0
}

set orthoslice [ vtkpxOrthoImageSlice [ pxvtable::vnewobj ]]
```

```
$orthoslice SetInput [ $tr GetOutput ]
$orthoslice SetFrame 0
$orthoslice SetLevels $x $y $z
$orthoslice SetDisplayMode 3
$orthoslice SetLookupTable $colormap

set ren [ vtkRenderer [ pxvtable::vnewobj ]]
$ren AddActor $orthoslice
```

Note that, in this case, we specify 3 levels on each for the YZ, XZ and XY planes respectively. In addition, we set a custom red/black lookup. The output of this script is shown in Figure 15.1(left middle).

The vtkpxObliqueImageSlice class: This displays an oblique cut through a 3D image. It's use is illustrated by script15-6.tcl. In addition to this slice, we also display a box object that consists of the outline bounds of the image to improve the visualization. The key code is:

```
# First the Oblique Slice
set obliqueslice [ vtkpxObliqueImageSlice [ pxvtable::vnewobj ]]
$obliqueslice SetInput [ $tr GetOutput ]
$obliqueslice SetFrame 0
$obliqueslice SetDisplayMode 3
$obliqueslice UpdateImagePlane 20.0 0.2 0.4 0.1

# Now the Outline
set outline [ vtkOutlineFilter [ pxvtable::vnewobj ]]
$outline SetInput [ $tr GetOutput ]
set map [ vtkPolyDataMapper [ pxvtable::vnewobj ]]
$map SetInput [ $outline GetOutput ]
set actor [ vtkActor [ pxvtable::vnewobj ]]
$actor SetMapper $map

set ren [ vtkRenderer [ pxvtable::vnewobj ]]
$ren AddActor $obliqueslice
$ren AddActor $actor
```

The position and orientation of the oblique slice is specified using the UpdateImagePlane command which takes four inputs: (i) the offset from the image center and (ii-iv) the x,y and z components of the normal to the image plane. The output from this script is shown in Figure 15.1(right middle). The lookup table can also be customized, as in vtkpxImageSlice.

The vtkpxVolume class: This class encapsulates the volume rendering pipeline in VTK. It takes as input an image, the volume resolution, and volume rendering mode and displays a volume rendered image. It's use is illustrated by script15-7.tcl. In addition to this slice, we also display a box object that consists of the outline bounds of the image to improve the visualization. The key code is:

```
set vol [ vtkpxVolume [ pxvtable::vnewobj ]]
```

```
$vol SetInput [ $tr GetOutput ]
$vol SetFrame 0
$vol SetResliceModeToHalfOriginal
$vol SetTextureMode 1

set ren [ vtkRenderer [ pxvtable::vnewobj ]]
$ren AddVolume $vol
```

The key settings are:

- ResliceMode – (SetResliceModeToOriginal, SetResliceModeToHalfOriginal, SetResliceModeToQuarterOriginal) which determines the size of the displayed volume and sets the tradeoff between rendering performance and quality.
- TextureMode – this selects the type of volume rendering: -1 = Maximum Intensity Projection, 0 = Normal ray-casting, 1=Texture Mapped Volume Rendering

In addition the intensity/transparency set up can be specified using a standard vtkLookupTable which can be set using the SetFromLookupTable method. The output of this script is shown in Figure 15.1(right).

15.4 Some Useful Filters

BioImage Suite has a number of useful filters. We discuss a small selection in this section. In particular we will describe:

- vtkpxImageExtract – extracts a single slice from a 4D image.
- vtkpxMatrix – a simple class to make matrix operations easier. It is a wrapper around CLapack.
- vtkpxAverageImages – a class for computing the average of many images.

The vtkpxImageExtract class: This is an image-to-image filter which can be used to a extract a single 2D slice from a 4D image. It is used extensively by vtkpxImageSlice and vtkpxOrthoImageSlice. The most common methods are:

- SetCurrentPlane – sets the orientation of the slice: (0=YZ,1=XZ,2=XY).
- GetCurrentPlane – returns the orientation of the slice
- SetSliceNo – sets the slice number (beginning at 0) which corresponds to the X, Y or Z coordinates if the CurrentPlane is 0,1 or 2 respectively.
- GetSliceNo – returns the current slice number
- SetFrame – sets the current frame (beginning at 0) for 4D images.
- GetFrame – returns the current plane number.

The following code snippet illustrates its use:

```
set extr [ vtkpxImageExtract [ pxvtable::vnewobj ]]
$extr SetInput $img
$extr SetCurrentPlane 2
$extr SetSliceNo 25; $extr SetFrame 0
$extr Update
```

The vtkpxMatrix class: This class is meant as a helper class for converting MATLAB code to C++/VTK, and as such it is more usefully employed at the C++ level. However, it can also be used with Tcl. It has a number of methods, most of whose names are meant to be similar to MATLAB commands. The most common methods are:

- Methods for Allocating space and filling with default values:
  - Allocate with 2 args – sets the size of the matrix
  - Zeros with 2 args – sets the size of the matrix and fills it with zeros.
  - Ones with 2 args – sets the size of the matrix and fills it with ones.
  - Eye with 1 arg – creates a square identity matrix
  - Identity with no arguments – sets the current matrix to the identity (only if it is square!)
  - Zero with no arguments – fills the current matrix with zeros.
  - Fill with 1 arg – fills the current matrix with the value provide.d
  - Copy with 1 arg – copies another matrix
  - Copy with 5 args (other,row1,row2,col1,col2) – copies a part of another matrix to this one.
  - Transpose – transpose the matrix.
  - Scale with 1 arg – scales the values with a constant.
  - ScaleAdd with 2 args (scale shift) – scales the values and adds a shift
- Element Manipulation
  - GetElement with 2 args – returns the value of the element arg1 arg2
  - SetElement with 3 args (row,col,val) – sets the value of the element(row,col)=val.
  - AddToElement with 3 args (row,col,val) – adds val to the value of the element (row,col)
- Methods for Printing/Loading/Saving:
  - Print with no arguments – prints the matrix.
  - Print with 1 arg – prints the matrix with a name i.e. a=[ ].
  - PrintRange with 4 args (row1,row2,col1,col2)– prints a portion of the matrix row1:row2, col1:col2. PrintRange with 5 args (name,row1,row2,col1,col2) – as above but with a name.
  - Print with 2 args (name,format) – prints the matrix with name and format (e.g. "%.2f")
  - Load with 1 arg – loads the matrix from a file (custom file format)
  - Save with 1 arg – saves the matrix from a file.
- Information About the Matrix:
  - GetSize – returns a list of the size of the matrix
  - Max – returns the maximum value in the matrix
  - Median – returns the median value of the matrix entries.
  - Sum – returns the sum of all the values in the matrix
  - SumSquares – returns the sum of the squares of all the entries in the matrix.
  - SumMagnitude – returns the some of the absolute value of all the entries in the matrix.
  - MaxColumn with 1 arg – returns the maximum value of column arg.
  - MaxRow with 1 arg – returns the maximum value of row arg
  - RowSums – returns a vtkpxMatrix object with the sums of all the rows
  - ColumnSums – returns a vtkpxMatrix object with the sums of all the columns.
- Simple Matrix Operations – these are static methods, so the current value of the invoking object is not used!
  - Add with 3 args (A , B , C), where A,B,C are vtkpxMatrix objects – results in C=A+B;
  - Add with 5 args (wa, A, wb, B,C), where wa,wb are constants – results in C = wa*A + wb*B
  - Multiply with 3 args (A,B,C) – results in C=A*B
  - Multiply3 with 4 args (A,B,C,D) – results in D=A*B*C
  - MultiplyTripleProduct with 3 args (A,B,C) – results in C=A'*B*A
- Linear Algebra Operations – these take the current object as input:
  - Diagonalize with 2 args (D,U) – returns the eigenvalues in a row matrix D, and the eigenvectors in U. Input must be square!
  - Eigenvalues with 1 arg (D) – returns the eigenvalues in a row matrix D.
  - QRDecomposition with 2 args (Q,R) – performs QR decomposition with outputs in Q and R
  - Invert – inverts the current matrix

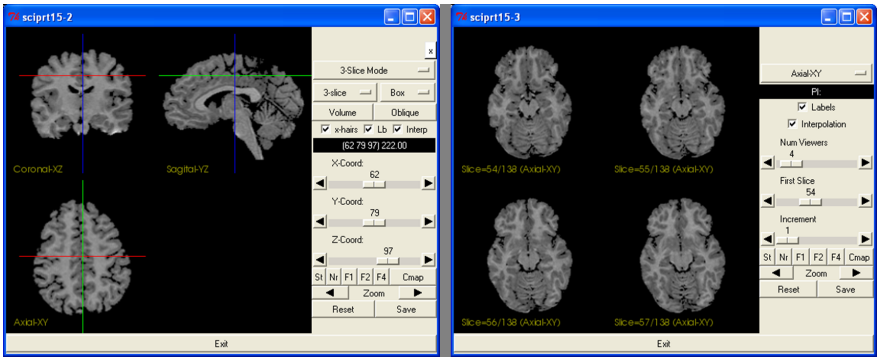


Figure 15.2: . The vtkpxGUIOrthogonalViewer (left, script15-2.tcl) and the vtkpxGUIMoisaicViewer (right, script15-3.tcl).

- SolveLeastSquares with 2 args (B,X) – Solves this*X=B , output in X (this=current matrix)
- SolveWeightedLeastSquares with 3 args (W,B,X) – weighted version of above.

Some of these operations are illustrated in script15-8.tcl.

The vtkpxAverageImages Class: This class takes a number of inputs (vtkImageData objects) of the SAME size and produces, either the mean, the median, the sum, and optionally the standard deviation. It's most common methods are:

- AddInput with 1 arg – adds an image to the input list
- SetMedian with 1 arg – if 1 then we compute the median if 0 the mean or the sum
- SetSumOnly with 1 arg – if 1 then we compute the sum (not the mean)
- SetComputeStandardDeviation with 1 arg – if 1, in addition to the mean compute the standard deviation
- GetOutputStandardDeviation – returns a vtkImageData object with the standard deviation (if computed)
- GetOutput – returns the sum,mean or median depneding on the options.

15.5 The BioImage Suite Viewers

BioImage Suite has four different viewers, all of which have a similar interface. These are:

1. vtkpxGUIMosaicViewer – can display multiple parallel slices of the same orientation.
2. vtkpxGUIOrthogonalViewer – can display orthogonal slices with linked cursors, as well volume renderings and oblique slices.
3. vtkpxGUIObjectmapOrthogonalViewer – an extension to the OrthogonalViewer which can transparently overlay a second image (commonly the output of a segmentation, hence the designation objectmap) over the standard image.
4. vtkpxGUI4DOrthogonalViewer – a 4D Version of the Orthogonal viewer which can be used to play movies of time-varying data.

These viewers are writen in C++ and make use of Tk widgets through a custom C++ wrapper around Tcl/Tk. The basic interface of all viewers is the same, as illustrated by the following scripts (script15-2.tcl,script15-3.tcl), see also Figure 15.2. These have the same first few lines, in which we load an image and create and pack an exit button, as follows. Note that the callback of the exit button has a second command to ensure proper exiting under windows.

```
lappend auto_path [ file dirname [ info script ]]
package require loadbioimagesuite 1.0

set ana [ vtkpxAnalyzeImageSource [ pxvtable::vnewobj ]]
```

```
$ana Load axial.hdr

set img [ $ana GetOutput ]
set orient [ $ana GetOrientation ]
puts stderr "Image Dimensions = [ $img GetDimensions ], Orientation = $orient"

button .bot -text "Exit" -command { destroy .; exit}
pack .bot -side bottom -expand false -fill x -pady 2
```

To create a viewer, we follow the following three steps:

- Create it, much like any VTK-object
- Optionally set any other flags to customize it's functionality.
- Initialize it given a parent widget and an "inside" flag. If the inside flag is set to zero the viewer is created inside a new toplevel widget, whose value is returned by the Initialize method.

In the case of the Orthogonal Viewer this takes, for example, the form:

```
set ortho [ vtkpxGUIOrthogonalViewer [ pxvtable::vnewobj ]]
$ortho  DisableTalairachButtons
$ortho Initialize . 1
```

In the case of the MosaicViewer, this code, similarly, has the form:

```
set mos [ vtkpxGUIMosaicViewer [ pxvtable::vnewobj ]]
$mos Initialize . 1
```

. Finally an image can be set as the input to the viewers with the command:

```
$ortho SetImage $img $orient
```

The syntax is identical for the Mosaic Viewer. SetImage can also take an optional third argument specifying the colormap (vtkLookupTable). If this is omitted, a default is used.

Getting access to the raw vtkRenderers: To add additional actors/volumes to the viewers one needs to get access to the raw vtkRenderer objects contained in them. In the case of the MosaicViewer, this is accomplished using the GetRenderer method, i.e. `set ren [ $mos GetRenderer 0 ]`. One *must be careful* not to ask for a renderer that does not yet exist!

In the case of vtkpxGUIOrthogonalViewer, there are 4 renderers. These correspond to the XY-slice, the XZ-slice, the YZ-slice and the 3D view respectively. They are accessed by a slightly more complex setup as follows:

```
set ren3d [ [ $ortho GetRenderer 3 ] GetRenderer ]
set renyz [ [ $ortho GetRenderer 0 ] GetRenderer ]
set renxz [ [ $ortho GetRenderer 1 ] GetRenderer ]
set renxy [ [ $ortho GetRenderer 2 ] GetRenderer ]
```

BioImage Suite encapsulates vtkRenderer inside a new class vtkpxGUIRenderer which adds some additional functionality. The first GetRenderer statement returns an instance of vtkpxGUIRenderer, which in turn when asked nicely (using GetRenderer) returns the raw underlying vtkRenderer object.

vtkpxGUIOrthogonalViewer: Additional Methods

- GetLastClickedPoint/ GetLastClickedPointScaled. These return the last point clicked in the viewer in voxels (GetLastClickedPoint) or mm (GetLastClickedPointScaled). An example is shown below:
  ```
  set lv [ $vtk_viewer GetLastClickedPointScaled ]
  set px [ lindex $lv 0 ]
  set py [ lindex $lv 1 ]
  set pz [ lindex $lv 2 ]
  ```
- SetCoordinates/SetScaledCoordinates $px $py $pz – The cross hairs can be set using the SetCoordinates (in voxels) or the SetScaledCoordinates (mm) methods.
- SetDisplayMode3D – switches to 3D only view
- SetDisplayMode2D – switches to the 3-slice linked cursor mode

vtkpxGUIOrthogonalObjectmapViewer: This viewer allows for the display of a second image that is transparently overlaid onto the main image. The level of transparency is set by the Mask control (0=completely transparent, 100=completely opaque).

The following two methods are used to set the object map image, and its associated color map (vtkLookuptable).

- int SetObjectMapImage(vtkImageData* img);
- int SetObjectLookupTable(vtkLookupTable* lkp);

The Objectmap Image must have the same dimensions as the primary image of the viewer (the one set using the SetImage method).

vtkpxGUIOrthogonal4DViewer: This class expands on vtkpxGUIOrthogonalViewer (is derived from it) and has 4 renderers for each frame, which are switched in and out as the frame is changed. To obtain access to these renderers one needs to modify the example, above, for vtkpxGUIOrthogonalViewer, in the following fashion:

```
set ren3d  [ [ $ortho GetMultiRenderer 3 ] GetRendererForFrame $frame ]
set renyz  [ [ $ortho GetMultiRenderer 0 ] GetRendererForFrame $frame ]
set renxz  [ [ $ortho GetMultiRenderer 1 ] GetRendererForFrame $frame ]
set renxy  [ [ $ortho GetMultiRenderer 2 ] GetRendererForFrame $frame ]
```

where frame is the desired frame (beginning at 0).

A Final Comment: At this point one can begin to think in terms of writing a complete application around one of these viewers. In practice, however, it is better to leverage higher-level Bioimage Suite functionality as desribed in the next Chapter.

Assignment

- (Perhaps using script15-8.tcl) write code that uses vtkpxMatrix to get the eigenvalues of a 4x4 symmetric positive definite matrix of your choice.
- Using script15-4.tcl as a base, modify it to add a scale widget that automatically adjusts the displayed slice of $imageslice along the z-axis.
- Using script15-2.tcl as a base, modify this to add a large sphere (centered at 80,80,62, radius 50) to the 3D renderer of OrthogonalViewer.
  Hint: You can get this renderer using: `set ren [ [ $ortho GetRenderer 3 ] GetRenderer ]`
  Hint2: You may want to use the SetDisplayMode3D command to switch the rendering to 3D mode.

Chapter 16

Writing your own BioImage Suite Application

In this Chapter we describe how one goes about writing a BioImage Suite application. This enables the use of many of the standard BioImage Suite components (e.g. collections of classes) for a new application. We first describe some of the [Incr] Tcl classes used by BioImage Suite to enhance the functionality of bare VTK classes. Then we desribe how to use the BioImage Suite application framework to write your own "BioImage Suite application". More information about BioImage Suite can be found at its web-page `www.bioimagesuite.org`.

16.1 Introduction

BioImage Suite is our home grown Medical Image Analysis Utility. It uses a combination of Tcl/[Incr Tcl] and C++ and leverages both VTK and ITK fairly substantially. For more information see `www.bioimagesuite.org`. The driving philosophy behind BioImage Suite is that it is a collection of customized applications sharing the same components and look and feel. As such it is ideally suited (not by accident) for use by students/researchers to create their own applications, by adding their own modules to the considerable functionality already in place.

16.2 Some Key [Incr] Tcl Classes

In this section we describe some key classes defined and used in BioImage Suite. A basic understanding of these classes is important in interfacing to this software.

The BioImage Suite Wrapper Classes

Most objects such as images, transformations and surfaces often require the storage of additional information that is allowed for in the standard vtk data structures. An example is the storage of filenames. While such information can be stored in the FieldData of a vtkDataObject, this is inconvenient at best.

The solution adopted in BioImage Suite is to create wrapper [Incr] Tcl objects around the basic vtk data structures to captures this additional functionality. The most useful of these objects are:

- pxitclobject – the basic parent class (bioimagesuite/main/pxitclobject.tcl), with the following derived classes:
  1. pxitclimage – a wrapper around vtkImageData

2. pxticlsurface – a wrapper around vtkPolyData
3. pxitcltransform – a wrapper around any transformation derived from vtkAbstractTransform

These type of wrapper objects are what is typically passed around BioImage Suite modules, so it is important to understand what they do.

The pxitclobject class: This does very little. It has one member variable (filename) and it defines a number of methods that derived classes are expected to override.

```
itcl::class pxitclobject {
    # --- begin variable def -----
    public variable filename  ""
    # --- end of variable def -----
    constructor { } {  }
    destructor  { }
    # --- begin method def -----
    public method GetThisPointer { } { return $this }
    # Things that must be over-ridden by derived classes
    # ------------------------------------------------
    public method GetObject { } { return 0 }
    public method GetType { } { return "vtkObject" }
    public method GetDescription { } { return [ $this GetType ]}
    public method UpdateStatus { }  { }
    public method Copy    { obj } { }
    public method Clear   { } { }
    # Procedures to Load/Save
    public method Load { args } { }
    public method Save { args } { }
}
```

The GetObject method provides a generic method for getting the VTK Data object that is wrapped inside this.

The pxitclimage class: This is the first concrete implementation. It adds four new member variables namely: (i) orientation, (ii) the vtkImageData object img, (iii) a lookuptable for storing an associated colormap and (iv) a status variable.

The most useful routine, in terms of interfacing to standard VTK code is the GetImage method which returns the underlying vtkImageData data structure. The CopyImage method can be used to copy the contents of an existing vtkImageData object into the wrapped object inside pxitclimage. This takes two arguments i.e. ShallowCopy obj mode, where obj is the incoming vtkImageData object and mode = 0 (shallow copy i.e. simply link the pointer) or 1 (deep copy, actually copying the data.)

See bioimagesuite/main/pxitclimage.tcl more more details.

The pxitcltransform class: This provides a wrapper around any vtkAbstractTransform object. It defines one new data member – transformation – which stores the VTK transformation. It also provides, similarly to pxitclimage, he methods GetTransformation and CopyTransformation.

See bioimagesuite/main/pxitcltransform.tcl more more details.

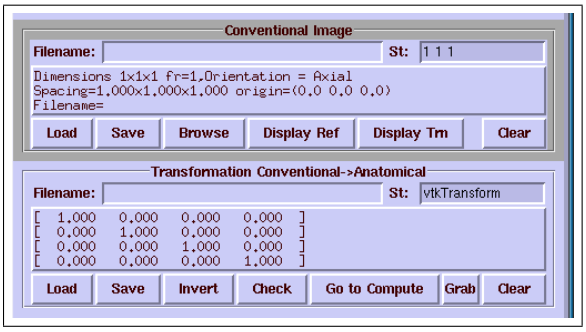


Figure 16.1: An example of two of the GUI controls: **Top:** The pxitclimageGUI control for manipulating an image. The filename is at the top (empty) with the dimensions (1 1 1) at the top right. The textbox below gives a longer description of the image. Common functionality (Load, Save, Browse, Clear) is available from the buttonbar below. Note that the two buttons "Display Ref" and "Display Trn" represent additional application specific functionality. **Bottom:** A transformation control for manipulating transformation objects (both linear and non-linear). The buttons "Check", "Go to Compute" and "Grab" represent additional application specific functionality.

The pxitclsurface class: This provides a wrapper around any vtkPolyData object. It defines two new data members, (i) sur – the vtkPolyData that is being wrapped and (ii) a status variable. It also provides, similarly to pxitclimage, he methods GetSurface and CopySurface.

See bioimagesuite/main/pxitclsurface.tcl more more details.

The BioImage Suite Simple GUI Classes

Often when we have an object we need a small set of controls to quickly perform common operations (think of right-clicking on an object in MS-WORD). For example, in the case of an image (pxitclimage), we need to be able to Load/ Save/ Get Information/ Display the image etc.

BioImage Suite provides a set of "ugly" controls for this purpose. These are derived from a parent class pxitclobjectGUI (which is in the file as bioimagesuite/main/pxitclobject.tcl), whose header (somewhat abbreviated) is described below. The first part consists of the member variables:

```
itcl::class pxitclobjectGUI {

    public variable  itclobject 0
    public variable  callback 0
    public variable  buttonbar 0
    public variable  loadbutton 0
    public variable  savebutton 0
    public variable  clearbutton 0
```

The itclobject member contains a pointer to the object that is being managed by the GUI (a derived class of pxitclobject). The callback variable contains the name of the command to call when the GUI is updated. The buttonbar, loadbutton, savebutton and clearbutton variables store the values of key widgets in the GUI, so that the user can manipulate it's appearance, e.g. `pack forget $savebutton` will remove the Save button from the GUI.

Next come the constructor and the destructor as well as some simple information methods. The GUI can either own the underlying pxitclobject or simply manage it. If it owns it, then this must be deleted when this object is deleted. The GetThisPointer method is a trick for returning a pointer to the object in global scope. The constructor only returns the object variable in local scope, by default.

```
    constructor { } { ... }
```

```
    destructor { if { $owns_object == 1} {
         # catch { } prevents error if the operation fails
         catch { itcl::delete object $itclobject }
      }
    }
    public    method GetThisPointer { } { return $this }
    public    method GetObject { } { return $itclobject  }
    public    method Reset { }
```

Next up, methods for creating the actual GUI and updating it. The GUI can be initialized either using the Initialize method (where basewidg) is the name of the core widget (which must not exist!), or the InitializeLite method which often creates a more 'compact look'. The args variable can be omitted in this case.

```
    public    method Initialize { basewidg }
    public    method InitializeLite { basewidg args }
    public    method Update { }
```

These are the callback functions for the standard buttons (Load, Save, Clear). They are overridden in derived classes.

```
    public    method Info { }
    public    method LoadObject { }
    public    method SaveObject { }
    public    method ClearObject { }
```

The add function command adds a new button to the gui whose name is specified by the name variable. The callback function for the button takes the form: `$command [ $this GetObject ] $post`.

```
    public    method AddFunction { command name post }
```

Finally a couple of functions for creating a new object and setting it. These must be redefined by derived classes (e.g. pxitclimageGUI).

```
    # Function that must be overriden
    protected method CreateNewObject { } { puts stdout "Error!"}
    public    method SetObject { tr } { puts stdout "Error!" }
```

The most useful derived classes of pxitclobjectGUI are pxitclimageGUI and pxitcltransformGUI which are (probably a bad idea) defined in the same files as pxitclimage and pxitcltransform respectively.

An extra layer of functionality is provided by the pxitclmultiObjectGUI (and derived classes). This can be used to manage a list of objects. It will not be described here in any detail.

The BioImage Suite Control Classes

Large GUI controls in BioImage Suite are derived from pxitclbasecontrol. The key chain in the hierarchy is:

```
pxitclbasecontrol --> pxitclbaseimagecontrol --> pxitclbaseimageviewer
```

In creating a your own BioImage Suite application, you will have to:

- Instantiate a new pxitclbaseimageviewer object to manage the core viewer.
- Construct your additional controls as derived classes from pxitclbaseimagecontrol and attach them to the viewer.

We describe these classes briefly next.

pxitclbasecontrol: This class provides the core functionality for a BioImage Suite control. It provides some useful methods which encapsulate complex functionality. The following is an abbreviated list:

- constructor parent – the parent refers to another pxitclbasecontrol (or derived class) which acts as a master for this control. Key events in this (slave) control are also passed to the parent (e.g. see WatchOn below).
- ShowWindow – shows the main control window.
- HideWindow – hides the main control window.
- SetTitle title – sets the title of the control window to title
- EnableUI widgetlist – enables all widgets (and their children) in the list widget list.
- DisableUI widgetlist – as above .
- WatchOn – puts a watch cursor on the window to indicate that something is going on. Also passes this to the parent (via $parent WatchOn).
- WatchOff – removes the watch cursor.
- AddLogoLabel w – Adds the BioImage Suite in a new label created inside frame $w. The new label is returned by this method.
- CreateProgressBar – Creates a progressbar inside an existing frame $w. This can be used to monitor progress of an algorithm.
- SetFilterCallbacks filter comment – Takes a filter (a vtkObject e.g. vtkImageShiftScale) and attaches it to the progressbar so that it's progress can be monitored.
- AboutCommand – brings up a dialog box – whose contents are controlled by the member variables appname, version and aboutstring to provide basic information about the current application.

pxitclbaseimagecontrol: This provides additional functionality in the form of three pxitclimage objects and methods to manipulate them. In general, although this need not be followed in all cases. the assumption is that the imagecontrol performs some processing on the currentimage and puts the output in currentresults. The currentimage variable will often be set by supervising imageviewer classes to which the control is attached as the image in the viewer is changed. Derived classes are naturally free to add more images if there is a need.

The most commonly used methods in this class are:

- SetImage img – sets the currentimage (img is a pxitclimage object)
- SetResults img – sets the currentresults object.

- SetImageFromObject img obj – similar to SetImage, obj is the source object for the change. This is often used to notify parent objects of a new image.
- SetResultsFromObject img obj – as above.
- AddPrefix oldname prefix – takes a filename and adds a prefix to it to reflect some processing.
- SendResultsToParent – passes the currentresults image to the parent object.
- SendImageToParent – passes the currentimage image to the parent object.
- CopyResultsToImage – make the results of the processing permanent, i.e. undoimage=currentimage, currentimage = currentresults.
- RevertToPreviousImage – undo the last CopyResultsToImage call

pxitclbaseimageviewer: This is a fairly complex (perhaps too complex) class for creating a viewer (one of the four viewers described in the last Chapter), and attaching some default functionality to it, see Figure 16.2 for an example. In the next Section we describe how to interface to this. A detailed description of this class is beyond the scope of the current class.

16.3 The Basic Application

Most BioImage Suite applications either derive from or have an instance of the pxitclbaseimageviewer.tcl class. This is an [Incr] Tcl class and can be found under bioimagesuite/main.

This basic application framework – see also `http://bioimagesuite.org/public/AppStructure.html` – is shown in Figure 16.2.

The main application first initializes the framework and then adds any custom controls. This is best illustrated by means of an example (mytool.tcl).

The first few lines are straightforward. MyUtility is the package containing the new control to be added to the core framework.

```
lappend auto_path [ file dirname [ info script ]]
package require loadbioimagesuite 1.0
package require pxitclbaseimageviewer 1.0
package require myutility 1.0
# Eliminate the default tk window
wm withdraw .
```

The next step is to initialize the core application, by selecting which of the common components should appear in the main menu.

These are the defaults and should serve most people well:

```
set baseviewer [ pxitclbaseimageviewer \#auto 0 ]
$baseviewer configure -appname "BioImage Suite::My Tool"
$baseviewer configure -show_standard_images 1
$baseviewer configure -enable_helpmenu 0
$baseviewer configure -enable_multisubjectcontrol 0
$baseviewer configure -enable_overlaytool 0
$baseviewer configure -enable_vvlinkgadget  0
$baseviewer configure -enable_talaraichoption 0
```

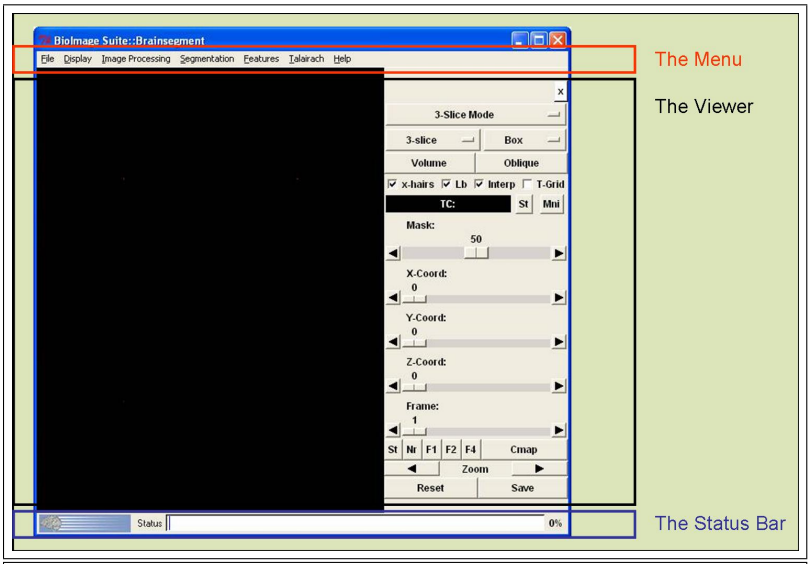


Figure 16.2: The framework for a BioImage Suite application. This consists of a menu bar, the viewer and a status bar. All these are created for you by invoking the pxitclbaseimageviewer class. The new components are attached to this framework using a customized menu entry.

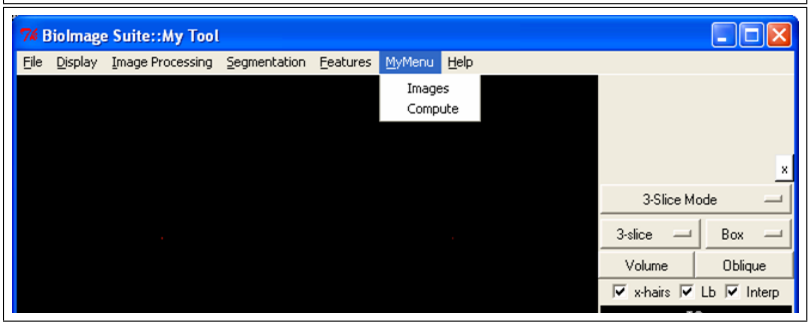


Figure 16.3: A customized application based on the core BioImage Suite framework. Note the new menu entry "My Menu" that provides access to the new functionality.

```
$baseviewer configure -enable_rendering_on_startup  0
```

The following is list of flags for enabling/disabling standard components (along with their default state: 1=ON, 0=OFF):

1. **enable_displaymenu 1** – if this is set to 0 the display menu is not added.
2. **enable_helpmenu 1** – if this is set to 0 the help menu is not added. This is useful for adding your own menu before adding the help menu as in the standard example.
3. **enable_histcontrol 1** – controls whether the histogram tool is added.
4. **enable_overlaytool 1** – controls whether the overlay/registration tool is added.
5. **enable_brainstriptool 1** – controls whether the segmentation tool is added.
6. **enable_headercontrol 1** – controls whether the analyze header editor tool will appear under the file menu.
7. **enable_importcontrol 1** – controls whether the image import tool will appear under the file menu.
8. **enable_imageutility 1** – controls whether the basic image processing tool will be available.
9. **enable_landmarkcontrol 1** – controls whether the landmark tool is included.
10. **enable_polydatacontrol 1** – controls whether the surface tool is included.
11. **enable_vvlinkgadget 0** – controls whether the VectorVision Link tool is included (if available)
12. **enable_talaraichoption 0** – shows/hides the option for displaying Talairach Coordinates.

The next step is to create a viewer. BioImage Suite has four viewers (described in the previous Chapter). One (and only one) can be initialized in this application, using one of the commands below. Make sure that 3/4 are commented out or deleted.

```
# Create a Viewer
# Uncomment one of the next four lines to select the viewer of your choice
$baseviewer InitializeOrthogonalViewer .[ pxvtable::vnewobj ] 1
#$baseviewer InitializeObjectmapViewer .[ pxvtable::vnewobj ] 1
#$baseviewer InitializeMosaicViewer .[ pxvtable::vnewobj ] 1
#$baseviewer InitializeOrthogonal4DViewer .[ pxvtable::vnewobj ] 1
```

The next step is to create our menu for adding new tools

```
# Add a submenu for your own tools
set menubase [ $baseviewer cget -menubase ]
set mb [ menu $menubase.vesselm -tearoff 0  ]
$menubase add cascade  -label "MyMenu" -menu $menubase.vesselm -underline 0
```

Each tool must be packaged into a class derived from pxitclbaseimagecontrol.tcl (more later). Here we initialize a control of type myutility. We next add it to the menu (using the AddToMenuButton method), and register it to the main application (using the AddControl method).

```
# Create your own tool (or tools), add it to the menu, and register it with baseviewer
set myutil [ myutility \#auto $baseviewer ]
$myutil Initialize [ $baseviewer GetBaseWidget ].[ pxvtable::vnewobj ]
$myutil AddToMenuButton $mb
$baseviewer AddControl $myutil
```

The rest is standard. Now that our menu is added, we create the help menu. Next we show the main window and if an argument is specified attempt to load an image from it into the main viewer.

```
# Finally create the help menu on the far right
$baseviewer CreateHelpMenu
# Show the Main Window
$baseviewer ShowWindow
# If an argument is specified, attempt to Load an Image from it
set argc [llength $argv]
if { $argc > 0 } {   $baseviewer LoadImage [lindex $argv 0]   }
# Enable rendering
update idletasks
[ $baseviewer GetViewer ] SetEnableRendering 1
```

The customized application is shown in Figure 16.3.

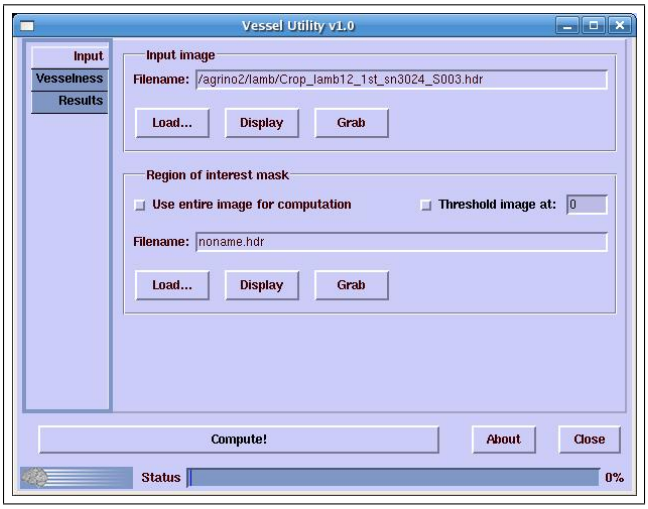


Figure 16.4: A typical BioImage Suite control. This consists of a (optionally) a menu – not used in this case, a tabbed notebook widget with multiple tabs that define the workflow through the control, potentially a button bar at the bottom and a status bar. Each tab may be directly accessed from the main application menu, if this is desired.

16.4 A Custom Control

Custom controls in BioImage Suite are most easily constructed by creating a class derived from pxitclbaseimagecontrol.tcl. This is also an [Incr] Tcl object. The typical control consists of multi-tab notebook window with different tabs providing for, for example, user interface elements for specifying input data, setting parameters and manipulating/saving output data. An example is shown in Figure 16.4.

The rest of this section describes a custom shell control (myutility.tcl) which can be used as a boilerplate for your own controls. The first part simply specifies the requirements etc.

```
package provide myutility 1.0
package require  Itcl 3.2
package require  Iwidgets 4.0
package require  pxitclbaseimagecontrol 1.0
```

Next we define the class header. This derives from pxitclbaseimagecontrol (to be found in bioimagesuite/main). The common array 'thisparam' is use to store variables which are linked to widgets directly, in the standard fashion of object-oriented GUIs.

The constructor simply calls the parent class constructor and then the InitializeControl method – this methods sets all the default parameters.

The Initialize method is used to build the graphical user interface. Since this is a tabbed-notebook design, it makes sense to have individual methods for each tab (CreateImageControl and CreateComputeControl). The AddToMenuButton method is used to add this object to a menubar.

Next there are two methods for interfacing to the main application: (i) GetPointsFromLandmarkControl – which gets the current set of landmarks from the Landmark Tool. This is useful for setting seeds etc. (ii) GrabImage, which grabs the current image from the viewer and stores it into one of the image-gui controls (more on this later) in the tool.

Finally, there are two methods for actually executing our algorithm (Compute) and storing the output image (StoreResult).

```
itcl::class myutility {
    # derives from pxitclbaseimagecontrol
    inherit pxitclbaseimagecontrol
    protected common thisparam

    #-----------------------------------
    # construction and descruction
    #-----------------------------------
    constructor { par } { pxitclbaseimagecontrol::constructor $par } { InitializeControl }
    destructor { }

    #-----------------------------------
    # initialization methods
    #-----------------------------------
    public method InitializeControl { }
    public method Initialize { inpwidg }

    #-----------------------------------
    # interface creation methods
    #-----------------------------------
    protected method CreateImagesControl { name }
    protected method CreateComputeControl { name }

    # -----------------------------------
    # Add this control to a Menu Button
    # -----------------------------------
    public method AddToMenuButton { mb args }

    # -----------------------------------
    # Computational Utility Stuff
    # -----------------------------------
    public method GetPointsFromLandmarkControl { }
    public method GrabImage { image control }

    # -----------------------------------
    # Do Something
    # -----------------------------------
    public method Compute { }
    public method StoreResult { image { name "" } { orientation 0 } }
}
```

The Initialize Control method is straightforwards and simply sets some default parameter values.

```
itcl::body myutility::InitializeControl { } {

    set appname "My Tool v1.0"
    set aboutstring "(c)  2006"


    # All Parameters that need to be appear in the GUI need to be part of this array
    # ------------------------------------------------------------------------
```

```
    set thisparam($this,enablesmoothing) 1
    set thisparam($this,initialdistance) 2.0

    # We need three image controls
    set thisparam($this,inputgui)      0
    set thisparam($this,initialphigui) 0
    set thisparam($this,outputgui)     0
}
```

The Initialize method creates the control window. This sits inside a new toplevel widget – specified using the widget parameter. Note that the initialized flag is set when this is done, multiple calls of the Initialize method only create one set of GUI-elements.

Once the toplevel is created, we create a notebook widget using iwidgets::tabnotebook. Each tab in the notebook is added using the notebook add command and passed to a helper method for creating the appropriate GUI inside it.

```
itcl::body myutility::Initialize { widget } {
    if { $initialized == 1 } { return $basewidget }
    #  ------------------------------------
    #  Create User Interface
    #  ------------------------------------
    set basewidget [toplevel $widget ]
    wm geometry $basewidget 610x450
    wm withdraw $basewidget

    set notebook $basewidget.notebook
    iwidgets::tabnotebook $notebook -tabpos w
    set widget_list(notebook) $notebook

    set mb [ frame $basewidget.mb ]
    pack $mb -side top -fill x -expand false

    CreateImagesControl     [ $notebook add -label "Images"  ]
    CreateComputeControl    [ $notebook add -label "Compute" ]
    pack $notebook -side top -fill both -expand t -padx 5

    set initialized 1
    SetTitle "My Utility"

    # this is critical
    eval "wm protocol $basewidget WM_DELETE_WINDOW { wm withdraw $basewidget }"
    return $basewidget
}
```

The CreateImagesControl is used to create the GUI inside the Images tab. BioImage Suite provides a small additional number of classes, in particular pxitclimage – an [Incr] Tcl wrapper around vtkImageData which provides additional storage for a filename and an orientation flag (see bioimagesuite/main/pxitclimage.tcl) and pxitclimageGUI (stored in the same file) which is a basic GUI for handling images.

Three pxitclimageGUI's are created. To each we add two new methods (buttons), a Display method which sends the image stored in the imageGUI to the viewer and (optionally) a Grab method which takes the image from the viewer and puts it in the image GUI.

```
itcl::body myutility::CreateImagesControl { base } {
    set names [ list "inputgui" "initialphigui" "outputgui" ]
    set titles [ list "Input Image" "Initial LevelSet" "Output" ]

    for { set i 0 } { $i < [ llength $names ] } { incr i } {
      set guiname [ lindex $names $i ]
      set title   [ lindex $titles $i ]
      set  thisparam($this,$guiname) [ [ pxitclimageGUI \#auto  ] GetThisPointer ]
      $thisparam($this,$guiname) configure -description $title

      # $base.$i is a new frame that gets created and the control put in it
      $thisparam($this,$guiname) Initialize $base.$i
      pack $base.$i -side top -expand f -fill x

      set bbut [ $thisparam($this,$guiname) cget -browsebutton ]
      pack forget $bbut

      $thisparam($this,$guiname) AddFunction "$parent SetImageFromObject" "Display" "$this"

      if { $guiname != "Output" } {
        $thisparam($this,$guiname) AddFunction "$this GrabImage" "Grab" \
                                    "$thisparam($this,$guiname)"
      }
    }
}
```

The Compute control is straight forward, standard [Incr] Tcl.

```
itcl::body myutility::CreateComputeControl { base } {

    iwidgets::labeledframe $base.frame0 -labelpos -labeltext "Parameters"
    pack $base.frame0 -fill both -expand f -pady 5

    set frame0 [ $base.frame0 childsite ]

    checkbutton   $frame0.c -variable [ itcl::scope thisparam($this,enablesmoothing) ] \
       -text "Enable Smoothing"
    iwidgets::entryfield $frame0.e -labeltext "Initial Distance:" \
         -textvariable [ itcl::scope thisparam($this,initialdistance) ] \
         -relief sunken -width 6 -validate real
    pack $frame0.c $frame0.e -side left -padx 2 -fill x -expand false

    eval "button $base.but -text \"Compute\" -command { $this Compute }"
    pack $base.but -side bottom -expand t -fill x
```

```
}
```

The control can be added to the menu, using the AddToMenuButton command. This creates one menu entry for each tab of the notebook as shown below:

```
itcl::body myutility::AddToMenuButton { mb args } {
    eval "$mb add command -label \"Images\" -command {$this ShowWindow \"Images\"}"
    eval "$mb add command -label \"Compute\" -command {$this ShowWindow \"Compute\"}"
}
```

The GrabImage function takes the current image in the viewer and puts it into one of the image gui's. The main application object is stored in the parent member variable. The pxitclbaseimageviewer GetDisplayedImage method gets the currently displayed image in the viewer and returns it as a pxitclimage object. (From this we can get the raw vtkImageData object using it's GetImage method, i.e. [pxitclimg GetImage]).

This image is copied into the image stored into the image-GUI (obtained using [$gui GetObject]) and then the gui is updated.

```
itcl::body myutility::GrabImage { image control } {
    # image is a pxitclimage of the control
    # control is the image gui
    set gui $control
    set img [ $parent GetDisplayedImage ]
    if { [ $img GetImageSize ] > 0 } {
         [ $gui GetObject ]  ShallowCopy $img
         $gui Update
    }
}
```

The landmarks in the landmark control are stored in a special internal class called vtkpxBaseCurve. The vtkpxBaseCurve class is too complex to describe here in any detail. The code in the next function takes care of converting a vtkpxBaseCurve object to a naked vtkPoints class.

```
itcl::body myutility::GetPointsFromLandmarkControl{ } {
  set landmarkcontrol [ [ $parent GetLandmarkControl ] GetLandmarkControl ]
  set tempc_lv [ vtkpxBaseCurve [ pxvtable::vnewobj ]]
  $tempc_lv Copy [ $landmarkcontrol GetCollection -1 ] $tempc_lv Compact set
  pts [ $tempc_lv GetPoints ]

  set points [ vtkPoints [ pxvtable::vnewobj ]]
  $points DeepCopy $pts
  $tempc_lv Delete
  return $points
}
```

The Compute function is just a placeholder for invoking your own algorithm! The pair pxtkconsole, pxtkprint use the BioImage Suite console (found under the help menu) to print the output.

```
itcl::body myutility::Compute { } {
    set smoothing $thisparam($this,enablesmoothing)
    set distance  $thisparam($this,initialdistance)

    # This opens the console
    pxtkconsole

    # This prints to the console
    pxtkprint "Parameters\n-----------------\n"
    pxtkprint "Use Smoothing : $smoothing\n"
    pxtkprint "Initial Dist  : $distance\n"

    set points [ $this GetPointsFromLandmarkControl ]
    if { $points != 0 } {
        set np [ $points GetNumberOfPoints ]
        pxtkprint "Number of Points : $np\n"
        for { set i 0 } { $i < $np } { incr i } {
            pxtkprint "Point $i : [ $points GetPoint $i ]\n"
        }
        pxtkprint "\n"
    }
    if { $points !=0 } { $points Delete }
}
```

The StoreResult method can be used to put an image (vtkImageData), with an associated filename (name) and orientation into the outputgui. This is for storing the output of your procedure.

```
itcl::body myutility::StoreResult { image { name "" } { orientation 0 } } {
    set gui $thisparam($this,outputgui)
    [ $gui GetObject ]  ShallowCopy $img
    [ $gui GetObject ]  configure -filename $name
    [ $gui GetObject ]  configure -orientation $orientation
    $gui Update
}
```

Finally after the class is defined, the following piece of code informs anyone who tries to execute this file directly that this is not a stand-alone program.

```
if { [ file rootname $argv0 ] == [ file rootname [ info script ] ] } {

    puts "\n[ file rootname $argv0 ] is not a stand-alone program.\n"
    exit
}
```

Part V

C++ Techniques

Chapter 17

Cross-Platform Compiling with CMAKE

CMake, is a cross-platform, open-source make system. It takes as an input as set of, relatively simple, configuration files and generates as an output native makefiles (UNIX) or Visual Studio projects (MS-WINDOWS) for the application. It was developed as part of the NLM Insight Segmentation and Registration Toolkit, and it has become another *de facto* standard in open source medical image analysis software development.

17.1 Introduction

In the dark old days of software development, writing a program, or a library, that could compile and run on different operating systems was a nightmare even for the most experienced programmer. Large software development (i.e. anything larger than 3-4 source files) uses some form of batch or automated compilation and build system. The UNIX standard is something called a makefile which is executed using the make command. In most MS-Windows development, the standard currency is MS Visual Studio project files. The key contribution of CMake is that it can take a single set of configuration files and generate, depending on the platform, either makefiles or Visual Studio project files, thus eliminating the need for the programmer to do both by hand (with the usual issues of propagating changes from one to the other as the project evolves).

More information on CMake can be found on its web-page: `www.cmake.org`. The material in this Chapter covers CMake 2.2, the latest version is 2.4.

17.2 A Simple Introduction

CMake takes as an input a file called CMakeLists.txt. The syntax of this can be a little confusing at first. For the most part, however, people simply use existing files and edit them appropriately for their application, so getting used to this should not take too long. Consider the simple C++ program "hello" which consists of a single source file called hello.cpp:

```
#include <stdio.h>

int main(int argc,char *argv[])
{
  fprintf(stderr,"Hello (CMAKE) World!");
  return 0;
}
```

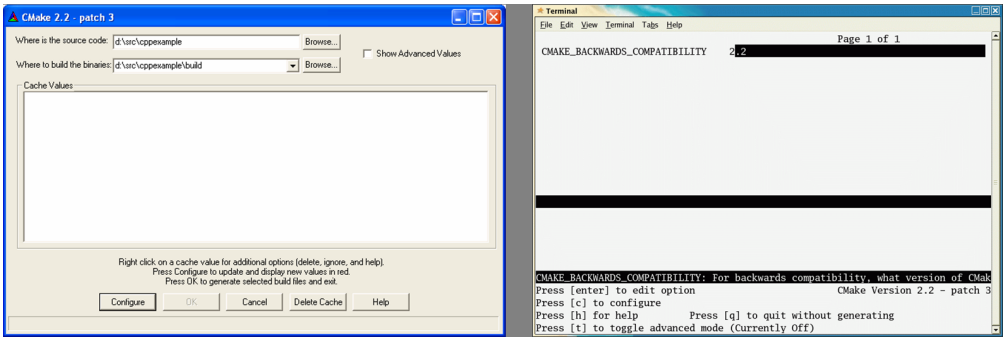


Figure 17.1: The CMake configuration program. (Left) Windows version cmakesetup.exe. (Right) Linux version ccmake.

To compile this program using CMake we first need to generate a CMakeLists.txt file in the same directory, which reads:

```
# First name the project
PROJECT(HELLOWORLD)

# Define the C++ Files that will go into the library, just one in this case
SET (PROGRAM_SRCS
hello.cpp
)

# Next add an executable
ADD_EXECUTABLE(hello ${PROGRAM_SRCS})
```

Once this is in place, it needs to be parsed to generate the appropriate development files (makefile of Visual Studio project files). This is accomplished using one of: (i) cmakesetup.exe (Windows) or (ii) ccmake (Unix), both of which are shown in Figure 17.1.

On Windows the first step is to specify the source code directory and the build directory – it is a good idea to keep these separate – using the two Browse buttons at the top. On Linux, the build directory is assumed to be the current directory and source directory is specified as an argument. For example, if you have two directories src and build, which contain the source code and the build files, we execute ccmake as follows:

```
cd build
ccmake ../src
```

Working with ccmake involves multiple parsings of the CMakeLists.txt file using the configure option (or 'c' on Unix). Each time the file is configured CMake may return requests for different pieces of information (e.g. where is Tcl located etc.). Once the necessary information is specified then the "OK" button (Windows) or the 'g' (generate and exit) option (Unix) appears, which can be used to complete the process.

The result of this is either a Visual Studio project file – a "solution" file with the suffix .sln or a Unix makefile.

Building on Linux: The project can be simply build using the `make` command. This is described in more detailed in the last section.

Building on Windows: Visual Studio must be first started. Then open the file "HELLOWORLD.sln" using the File/Open Solution option in MS Visual Studio. Then Build this as usual.

17.3 A Second Example

In the programming style that I am attempting to "sell" in this class, C++ is primarily to be used to create new Tcl commands which can then be invoked from Tcl scripts. VTK has a very sophisticated mechanism for automatically generating the necessary wrapper code to accomplish this wrapping, which we will leverage later. However, it is useful to once see the manual version of this process. The process involves creating a dynamic library (.so or .dll) which will include our new code, which we will then load into the Tcl interpreter.

The new command to be added is a simple hello-world like statement. The C++ code takes the form:

```
#include "tcl.h"
#include "tclDecls.h"
#include <stdio.h>
#include <string.h>

int HelloTcl(ClientData, Tcl_Interp* interp,int objc, Tcl_Obj *CONST objv[])
{
  fprintf(stderr,"Hello From C++\n");
  return TCL_OK;
}
```

The first part of the program defines the new command HelloTcl. It has a fixed parameter structure. When called it will simply print "Hello from C++" and return the appropriate error code (TCL_OK) signifying that all is OK.

The next step is to let the Tcl interpreter know that this command is available and to provide the mechanism for calling it. We need to do two arcane pieces of coding first: (i) we need to provide Tcl with an initialization procedure – which must be in "C" linkage to ensure portability (don't worry if you don't understand this!) and (ii) we need to, in the case of MS-WINDOWS, declare the initialization procedure as 'dllexport' so it is accessible from outside the DLL.

```
// Initialization Code
#ifdef _WIN32
#define HELLOTCL_EXPORT __declspec( dllexport )
#else
#define HELLOTCL_EXPORT
#endif

extern "C" {
int  HELLOTCL_EXPORT Hellotcl_Init(Tcl_Interp* interp);
int  HELLOTCL_EXPORT Hellotcl_SafeInit(Tcl_Interp* interp);
}
```

```
int Hellotcl_Init(Tcl_Interp* interp)
{
  Tcl_CreateObjCommand(interp, "cpphello", HelloTcl,(ClientData) NULL,
                       (Tcl_CmdDeleteProc *) NULL);
  return TCL_OK;
}

int Hellotcl_SafeInit(Tcl_Interp* interp)
{
  return Hellotcl_Init(interp);
}
```

The initialization procedure is called Hellotcl_Init, where hellotcl is the name of the dynamic library (hellotcl.dll or libhellotcl.so). We need two of these procedures for both ordinary and safe-execution (another detail which you can safely ignore) cases.

The key command is the invocation of the Tcl library function Tcl_CreateObjCommand which tells the Tcl intepreter that when a user invokes the 'cpphello' command, the interpreter must call the HelloTcl function just defined above.

A First Attempt at a CMakeLists.txt file

```
PROJECT(HELLOTCL)
# Set this flag variable first
SET (HELLO_TCL_CANBUILD 1)
# Define the C++ Files that will go into the library, just one in this case
SET (Library_SRCS
hellotclcpp.cpp
)
# Set the default location for outputting the library
SET (LIBRARY_OUTPUT_PATH ${HELLOTCL_SOURCE_DIR})
# Look for tcl.h
FIND_PATH(TCL_INCLUDE_PATH tcl.h PATHS)
IF (TCL_INCLUDE_PATH)
   INCLUDE_DIRECTORIES(${TCL_INCLUDE_PATH})
   FIND_LIBRARY(TCL_LIBRARY NAMES tcl tcl84 tcl8.4)
ELSE (TCL_INCLUDE_PATH)
   SET (HELLO_TCL_CANBUILD 0)
ENDIF (TCL_INCLUDE_PATH)
# If tcl.h has been found and also libtcl then we can build
IF (HELLO_TCL_CANBUILD)
# Add a new shared library based on the source files above
   ADD_LIBRARY(hellotcl SHARED ${Library_SRCS})
# Ensure that the TCL library is linked to it
   TARGET_LINK_LIBRARIES(hellotcl ${TCL_LIBRARY})
ENDIF (HELLO_TCL_CANBUILD)
```

Our library has two dependencies: (i) the tcl.h file and (ii) the tcl library.[1] The first step in the CMakeLists.txt is to find where tcl.h is located on the current machine. This is accomplished using the `FIND_PATH(TCL_INCLUDE_PATH tcl.h PATHS)` statement. Once tcl.h is specified, it's location is added to the include directories used for compilation using the `INCLUDE_DIRECTORIES` command. **There should be no white-space** between directives, e.g. INCLUDE_DIRECTORIES, and the parenthesis sign that follows them!

Next, conditional upon finding tcl.h, we look for the Tcl library. This is accomplished using the `FIND_LIBRARY()` statement. If the Tcl library is found we can now proceed to building our library otherwise we set the CAN_BUILD variable to zero to indicate that a dependency is missing.

To build our library we need four parts: (i) We need to list the files that need to be compiled into the library, this is accomplished using:

```
SET (Library_SRCS
hellotclcpp.cpp
)
```

Next, we, optionally, define the default library location using:

```
SET (LIBRARY_OUTPUT_PATH ${HELLOTCL_SOURCE_DIR})
```

The HELLOTCL_SOURCE_DIR variable (where HELLOTCL is the project name) points to the current source tree. On windows, MS Visual Studio, will add a Debug or Release subdirectory under this depending on the type of build.

The third step involves defining the library itself: small

```
ADD_LIBRARY(hellotcl SHARED ${Library_SRCS})
```

The first argument is the library name, The second is either SHARED or STATIC depending on the type of library, and th third is the list of C++ files that will be used.

In the final step we specify any libraries that need to be linked to it, in this case the Tcl library, using:

```
TARGET_LINK_LIBRARIES(hellotcl ${TCL_LIBRARY})
```

Then we compile our library using the usual ccmake and make steps.

Loading the Library: The use of this small library is demonstrated by the following short script (testhello.tcl):

[1] When using BioImage Suite, these files are in /usr/local/vtk44_yale/include and /usr/local/vtk44_yale/lib respectively. On MS-Windows replace /usr/local with c:/yale as usual.

```
}
lappend auto_path [ file dirname [ info script ]]

if { $tcl_platform(platform) == "windows" } {
    load debug/hellotcl.dll
} else {
    load libhellotcl.so
}
cpphello
```

The 'load' command is used to load the DLL into the interpreter. This makes the new command 'cpphello' available which we can then invoke.

A More Sophisticated CMakeLists.txt file CMake has special macros for finding commonly used packages such as Tcl, VTK or ITK. In this case we search for the TCL package as follows:

```
INCLUDE (${CMAKE_ROOT}/Modules/FindTCL.cmake)

IF(TCL_LIBRARY)
   INCLUDE_DIRECTORIES(${TCL_INCLUDE_PATH})
   ADD_LIBRARY(hellotcl SHARED ${Library_SRCS})
   TARGET_LINK_LIBRARIES(hellotcl ${TCL_LIBRARY})
ENDIF(TCL_LIBRARY)
```

17.4 Some Additional Comments

CMake has lots of other options which we can not cover here. A reference guide is available at

`http://cmake.org/HTML/Documentation.html`. The following commands are particularly important:

- ADD_EXECUTABLE: Add an executable to the project using the specified source files.
- ADD_LIBRARY: Add a library to the project using the specified source files.
- IF..ELSE..ENDIF: These constructs enable conditional execution.
- FILE: This enables outputting of information to text files
- FIND_FILE: Find the full path to a file.
- FIND_LIBRARY: Find a library.
- FIND_PACKAGE: Load settings for an external project.
- FIND_PATH: Find the directory containing a file.
- INCLUDE: Read and include CMake code from the given file. This is a way of modularizing complex CMakeLists.txt files.
- LINK_DIRECTORIES: Specify directories in which to search for libraries.
- LINK_LIBRARIES: Link libraries to all targets added later.
- MESSAGE: Display a message to the user.
- OPTION: Provides an option that the user can optionally select.
- PROJECT: Set a name for the entire project.
- SET_SOURCE_FILES_PROPERTIES: Source files can have properties that affect how they are built. This is useful for declaring classes as abstract etc.
- TARGET_LINK_LIBRARIES: Link a target to given libraries.

17.5 Appendix: A Brief Overview of the Make Utility

On Unix systems, CMake results in a makefile which needs to be processed with the Unix Make Utility (most commonly GNU Make). Makefiles are primarily used to compile large programs they are a great mechanism for large system development because they:

- Can be used to define dependencies – i.e. file B must be recompiled because file A changed.
- Use of multiple processors at the same time – i.e. compile two or more files at once.

The standard name for a makefile is unsurprisingly "makefile". Given a makefile, the build procedure is executed using:

```
make
```

On Linux systems make=gmake i.e. typing gmake is equivalent to typing make – this may appear in some examples. Additional useful flags include:

1. "-n" - do a dry run i.e. simply print a list of commands to be executed without doing anything
2. "-j" - specify how many jobs to run at once – typically equal to the number of processors available e.g. to use 2 processors type `make -j2`.

In addition makefiles contain a number of "jobs" which may be explicitly specified. For example "make clean" will clean all results of previous compilations, so that the whole project can be compiled from scratch upon invocation of the next make command.

Assignment

- Ensure that you can compile and run the three examples in the repository (helloworld, cpphelloworld, cpphelloworld2)
- Modify helloworld to print some additional statements.
- Add a second function in hellotcl.cpp (e.g. goodbye world) and make this also available as a Tcl command.

Chapter 18

C++ Techniques and VTK

In this Chapter we will quickly review basic C++ coding conventions, with respect to object orientated programming and templates. We also present examples of the use of VTK with C++. This Chapter is meant to serve as a transition from the more Tcl oriented material presented in the previous Chapters to C++-based programming that will form the core of the following Chapters. All examples are meant to be built using the CMake utility described in the previous lecture.

18.1 Introduction

C++ has become the *de facto* choice of programming language for large projects. While other languages such as Java and C# have gained popularity, especially for web-related programming, C++ is still for all intents and purposes the premier programming language out there. One of the great strenghts and weaknesses of C++ is that is a "big" language which allows for a multitude of programming styles. It can be used as simply a better version of C. However, C++, when expanded by use of object-oriented and more recently generic programming techniques, becomes a very different beast. We try to discuss some of these aspects in this lecture.

New and Delete: The most common problem in C/C++ that programmers face is memory allocation and de-allocation. In VTK this is dealt with by the use of reference counted objects, but in many cases one needs to understand how memory allocation works at the raw level. Memory in C++ is allocated using the 'new' operator. For example, a single object obj (see later examples) and an array of 10 integers can be allocated using:

```
obj* newobj= new obj;      int *i = new int[10];
```

Freeing the allocated memory takes two forms depending on whether one is deleting a single object or an array of objects (or variables). For a single object we use the `delete` command whereas for an array we use the `delete []` command. This is illustrated below:

```
delete  newobj;            delete [] i;
```

18.2 Object-Oriented Programming with C++

The Infant Revisited: In Chapter 7 we introduced the concept of object-oriented programming using the [Incr] Tcl extensions of Tcl. This was done on purpose, so as to introduce a fairly complex concept without the need to use a heavy duty compiled language, like C++, which adds additional baggage related to compiling and building programs. In this section we revisit the material of Chapter 7 and convert it (mostly line-by-line) to C++. (All code for this section is in the oopexample subdirectory). The infant class described in Chapter 7, when converted to C++ takes the form:

infant.tcl – top portion

```
itcl::class Infant {

    # Class Variables
    protected variable myWeight 2.0
    protected variable myName   "Anonymous"

    # Constructor and Destructor
    constructor { newname } {set myName $newname}
    destructor {set myName "";set myWeight ""}

    # Interface, public methods
    public method GetName { }
    public method GetWeight { }
    public method SetWeight { wgt }
    public method DailyRoutine { }
    public method PrintSelf { }

    # Protected methods
    protected method Cry { }
};
```

infant.h

```
class Infant {

protected:
  // Member Variables
  float myWeight;
  char* myName;

public:
  Infant(char* newname);
  virtual ~Infant();

  // Interface
  virtual char*  GetName();
  virtual float  GetWeight();
  virtual void   SetWeight(float wgt);
  virtual void   DailyRoutine();
  virtual void   PrintSelf();

protected:
  virtual void Cry();
};
```

The "virtual" construct implies that this method may be overridden by a derived class in the future. The exact behavior of this is harder to explain, but a simple rule of thumb is: make all object methods virtual unless you have a really good reason not to (in which case you probably also understand exactly what virtual does!). The class constructor has the same name as the class, e.g. Infant. The destructor takes the name `~classname`. It is a good idea to also make the destructor virtual. Also note that unlike [Incr Tcl], we do not use a protected or public attribute at the start of each definition, rather protected and public methods/attributes are declared in separate sections beginning with the "public:" or "protected:" (or "private:" for that matter) declaration.

Finally in C++ the standard convention is to put the class definition in a header file (.h e.g. infant.h) and the implementation in a separate file. The C++ implementation for the infant class is in a file infant.cpp which is presented below:

```
#include "infant.h"
#include <string.h>
#include <stdio.h>

Infant::Infant(char* newname)
{
  this->myWeight=2.0;
  this->myName=new char[200];
  strncpy(this->myName,newname,200);
```

```
}

Infant::~Infant() { delete [] this->myName;}

char* Infant::GetName() { return this->myName;}

float Infant::GetWeight() { return this->myWeight;}

void Infant::SetWeight(float wgt) { this->myWeight=wgt;}

void Infant::DailyRoutine()
{
  fprintf(stderr,"infant ... Daily Routine Start\n");
  this->Cry();
  fprintf(stderr,"infant ... Daily Routine End\n");
}

void Infant::Cry() { fprintf(stderr, "infant ... WaWa!\n");}

void Infant::PrintSelf()
{
  fprintf(stderr,"infant ... myName = %s\n",this->myName);
  fprintf(stderr,"infant ... myWeight = %.2f\n",this->myWeight);
}
```

Note the use of `this->` prefix to access all member variables and methods. While this is not strictly necessary, it is good programming style – it is also the standard convention in VTK. The code itself has no complex constructs worth discussing, make sure you understand what each line does!

The final step is to instantiate a version of the infant class. We present below the original [Incr] Tcl code side-by-side with the C++ code.

script7-1.tcl

```
lappend auto_path [ file dirname [ info script ]]
package require Itcl 3.2
package require Infant

set leanboy [ Infant \#auto "A" ]
$leanboy SetWeight 7.0

set fatboy [ Infant \#auto "B" ]
$fatboy SetWeight 11.0

puts stderr "\nLet's see the details on $leanboy"
$leanboy PrintSelf
$leanboy DailyRoutine

puts stderr "\nLet's see the details on $fatboy"
$fatboy PrintSelf

itcl::delete object $leanboy
itcl::delete object $fatboy
```

main1.cpp

```
#include <stdio.h>
#include "infant.h"
int main(int argc,char *argv[])
{
  Infant* leanboy=new Infant("A");
  leanboy->SetWeight(7.0);

  Infant* fatboy=new Infant("B");
  fatboy->SetWeight( 11.0);

  fprintf(stderr,"\n ... details on leanboy\n");
  leanboy->PrintSelf();
  leanboy->DailyRoutine();

  fprintf(stderr,"\n ... details on fatboy\n");
  fatboy->PrintSelf();

  delete leanboy;
  delete fatboy;
  return 0;
```

```
}
```

Note that the new object is instantiated by calling its constructor (the `new Infant()` construct) and deleted using the delete command.

A Derived Class – FourYearOld: [Incr] Tcl uses the "inherit" statement to declare that a class is derived from another. In C++, by contrast, this is performed at the class definition itself i.e.

`class FourYearOld : public Infant` This is illustrated below:

fouryearold.tcl – top portion

```
itcl::class FourYearOld {
    inherit Infant

    ....
};
```

fouryearold.h

```
class FourYearOld : public Infant {

...

};
```

We will not discuss this in any more detail, the code is available in fouryearold.h. The only other point worth touching upon is calling a method of the parent class explicitly. For example, while FourYearOld redefines the PrintSelf method, it still needs to call the PrintSelf method from the Infant class. This is accomplished using `::Infant::PrintSelf()` as shown below:

```
void FourYearOld::PrintSelf()
{
  ::Infant::PrintSelf();
  fprintf(stderr, "fouryearold ... myFavoriteFood  = %s\n",this->myFavoriteFood);
  fprintf(stderr, "fouryearold ... myFavoriteColor = %s\n",this->myFavoriteColor);
}
```

Static Members and the IrsInfant class: In Chapter 7 (script7-4.tcl) we also demonstrate the use of static or common member functions and variables: functions and variables which belong to the collective as opposed to any individual object. The syntax in C++ is similar, compare for example:

script10-4.tcl – top portion

```
itcl::class IrsInfant {
  ...
    # Common
    protected common NumInfants 0
    protected proc NewSSN { }
    public    proc GetNumInfants { }


    ....
};
```

irsinfant.h

```
class IrsInfant : public Infant
{
public:
   static int GetNumberOfInfants();
...

protected:
  // Static Stuff
  static int NewSSN();
  static int NumInfants;
};
```

One minor difference is that in the C++ case, IrsInfant is derived from Infant, to save coding. Note that static member variables in [Incr] Tcl are declared using the "common" statement and static functions are declared using the "proc" statement (as opposed to method). In C++ both methods and member variable declarations use the static qualifier.

The code of irsinfant.cpp is worth looking at:

```
#include "irsinfant.h"
#include <stdio.h>

// This is critical, it sets the default value of the static variable and in
//  some respects creates it.
int IrsInfant::NumInfants=0;

IrsInfant::IrsInfant(char* newname):Infant(newname) {
  this->mySSN=IrsInfant::NewSSN();}

int IrsInfant::GetSSN() {  return this->mySSN; }

void IrsInfant::PrintSelf() { ::Infant::PrintSelf();
  fprintf(stderr, "irsinfant ... mySSN  = %d\n",this->mySSN);
}

int IrsInfant::GetNumberOfInfants() {  return IrsInfant::NumInfants;}

int IrsInfant::NewSSN() {  ++IrsInfant::NumInfants;
  return 1000+IrsInfant::NumInfants;}
```

Note a couple of things: (i) the static member variable NumInfants needs to be explicitly declared in the .cpp file (otherwise it never exists!). It's value can also be initialized (although beware, if this code is compiled in a Windows DLL function, this initialization will not occur!) (ii) All static member functions/variables are accessed using the `IrsInfant::` prefix whereas ordinary member variables are accessed using `this->`.

Compiling these examples: The four classes: Infant, FourYearOld, Teenager and IrsInfant are defined in the file pairs infant.h/infant.cpp, fouryearold.h/fouryearold.cpp etc and can be found in the `oopexample` directory. You may notice that at the top and bottom of each header file there is a construct of the form:

```
#ifndef _IrsInfant
#define _IrsInfant

.... alll code here ...

#endif /* _IrsInfant */
```

This is a well-known trick for speeding up compilation. It essentially says: If the constant/macro `_IrsInfant` is not defined (#ifndef), define it and process the code following. This is terminated by the #endif statement. On the other hand if `_IrsInfant` is defined, that means that the compiler has already parsed this code probably because irsinfant.h was included from multiple files that are being compiled, hence the compiler should skip the code below, it has already been compiled! If you use this (which I recommend) make sure that you use a unique identifier (`_IrsInfant` in this case) for each header file, otherwise you will get unexpected results (unexpected = bad! when programming is concerned).

The compilation process is based on CMAKE (see Chapter 17). The CMakeLists.txt file for this set of examples has the form:

```
PROJECT(DEMO)
# Define the C++ Files that will go into the library, just one in this case
SET (Library_SRCS
infant.cpp
fouryearold.cpp
teenager.cpp
irsinfant.cpp
)

INCLUDE_DIRECTORIES(${DEMO_SOURCE_DIR})
ADD_LIBRARY(mylib STATIC ${Library_SRCS})

ADD_EXECUTABLE(main1 main1.cpp)
TARGET_LINK_LIBRARIES(main1 mylib)

ADD_EXECUTABLE(main2 main2.cpp)
TARGET_LINK_LIBRARIES(main2 mylib)

ADD_EXECUTABLE(main3 main3.cpp)
TARGET_LINK_LIBRARIES(main3 mylib)

ADD_EXECUTABLE(main4 main4.cpp)
TARGET_LINK_LIBRARIES(main4 mylib)
```

We first compile the four classes (infant, fouryearold, teenager and irsinfant) into a static library `mylib`. This is common programming practice – in which utility code is placed into libraries for reuse. There are two main kinds of libraries, (i) static, which end in .a on UNIX or .lib on Windows, which must be explicitly added to the main executable at build time, or (ii) dynamic/shared which end in .so on Unix, .dll on Windows (and .dylib on Mac OS!) which are linked with the main program at run time. Small libraries are usually compiled as static, whereas larger libraries, especially those shared by multiple programs, are compiled in dynamic format to save disk space (among other benefits). Note also that on Windows, you need static libraries in addition to dynamic libraries for linking, dynamic libraries alone are insufficient.

To create the library we first put all the C++ files into a list `Library_SRCS` and then create the library using:
`ADD_LIBRARY(mylib STATIC ${Library_SRCS})`

The executables main1.cpp to main4.cpp (see their code) are created using the `ADD_EXECUTABLE` statement, and the `TARGET_LINK_LIBRARIES` statement is used to ensure that mylib is linked to each of them.

18.3 VTK with C++

In the previous chapters we have invoked VTK objects from Tcl. In this section we directly instantiate such objects in C++. (All code for this section is in the vtkexample subdirectory.) The syntax is *very similar* and it is often trivial to go back and forth. One caveat, however, is that some functions of VTK objects are only accessible from C++ as they require the passing of pointers to arrays. We will revisit some of the example scripts from Chapter 10 and convert them to C++. The first example we will look at is script10-1.tcl which maps to example1.cpp of this section.

example10-1.tcl

```
lappend auto_path [ file dirname [ info script ]]
package require newname
```

example1.cpp

```
#include <vtkFloatArray.h>
```

```
wm withdraw .

set arr [ vtkFloatArray [ newname::vnewobj ]]
$arr SetNumberOfComponents 2
$arr SetNumberOfTuples    12
$arr FillComponent 0  0.0
$arr FillComponent 1 10.0

$arr SetComponent 10 0 3.0
$arr SetTuple2    11 9.0 2.0
$arr SetComponent 4 4 -2.1

puts stdout "The array has"
puts stdout "[ $arr GetNumberOfTuples ] tuples"
puts stdout " and [ $arr GetNumberOfComponents ]"
puts stdout "Here are its contents"


set nt [ $arr GetNumberOfTuples ]
set nc [ $arr GetNumberOfComponents ]
for { set i 0 } { $i < $nt } { incr i } {
    puts -nonewline stdout "Tuple $i : ("
    for { set j 0 } { $j < $nc } { incr j } {
puts -nonewline stdout "\t [ $arr GetComponent $i $j ]"
    }
    puts stdout "\t)"
}
exit
```

```
int main(int argc,char *argv[])
{
  vtkFloatArray* arr=vtkFloatArray::New();
  arr->SetNumberOfComponents(2);
  arr->SetNumberOfTuples(12);
  arr->FillComponent(0,0.0);
  arr->FillComponent(1,10.0);

  arr->SetComponent(10, 0, 3.0);
  arr->SetTuple2(11,9.0, 2.0);
  arr->SetComponent(4, 1, -2.1);

  fprintf(stderr,
    "Array has %d tuples and %d components\n\n",
     arr->GetNumberOfTuples(),
     arr->GetNumberOfComponents());
  fprintf(stderr,"Here are its contents\n");

  int nt=arr->GetNumberOfTuples();
  int nc=arr->GetNumberOfComponents();
  for (int i=0;i<nt;i++)
    {
      fprintf(stderr,"Tuple %d : (",i);
  for (int j=0;j<nc;j++)
fprintf(stderr,"\t %.2f ",arr->GetComponent(i,j));
      fprintf(stderr,"\t)\n");
    }
  return 0;
}
```

The first key item is the creation of a new object. To enable proper use of reference counting in VTK, the constructor is *protected!* The new object is constructed by calling a public static member `New()`. Hence the array is created using:

```
vtkFloatArray* arr=vtkFloatArray::New();
```

Also note that VTK constructors (and the New methods) take no arguments, all parameters must be specified subsequently. The object is deleted using its delete method (this is not shown in the code) which takes the form:

```
arr->Delete();
```

The rest of the code is very similar and it is a simple matter of translating Tcl Syntax to C++ syntax (which is what I did in creating example1.cpp).

A Second Example: This example (example4.cpp) derives from (script10-5.tcl) in which we construct a cone and render it on the scene. First we include all the header files for all the VTK classes we are going to invoke:

```
#include <vtkPoints.h>
#include <vtkCellArray.h>
#include <vtkPolyData.h>
#include <vtkMath.h>
#include <vtkRenderer.h>
#include <vtkRenderWindow.h>
#include <vtkPolyDataMapper.h>
```

```
#include <vtkProperty.h>
#include <vtkCamera.h>
#include <vtkRenderWindowInteractor.h>
```

The main function follows. As an aside, since the value of π is not defined on many operating systems by default, one way to get this is to use the static member function of the vtkMath class Pi(). Next we create the points for the cone as before:

```
int main(int argc,char *argv[]) {
  double  pi=vtkMath::Pi().

  vtkPoints* pts=vtkPoints::New();
  pts->SetNumberOfPoints(9);
  for (int i=0;i<=7;i++) {
      double rad= 2.0*double(i)*pi/8.0;
      double x  = 10.0* sin(rad);
      double y  = 10.0* cos(rad);
      pts->SetPoint(i,x,y,0.0);
      double* p=pts->GetPoint(i);
      fprintf(stdout,"Point %d = (%.2f,%.2f,%.2f)\n",i,p[0],p[1],p[2]);
    }
  pts->SetPoint(8,0.0,0.0,10.0);
```

In creating the cells, we can use a shorthand that enables the passing of an array of points to the cell using a different version InsertNextCell command. Recall that in Tcl, each triangle was created using:

```
 $triangles InsertNextCell 3
 $triangles InsertCellPoint $p1;
 $triangles InsertCellPoint $p2
 $triangles InsertCellPoint 8
```

The C++ code is below:

```
  vtkCellArray* triangles=vtkCellArray::New();
  triangles->Allocate(10, 5);
  int p[3];
  for (int i=0;i<=7;i++) {
      p[0]=i; p[1]=i+1; p[2]=8;
      if ( p[1] > 7) p[1]=0;
      triangles->InsertNextCell(3,p);
      fprintf(stdout,"Set Triangle %d = (%d,%d,%d)\n",i,p[0],p[1],p[2]);
    }
```

Next we create the surface and do some clean up – note the use of the Delete method.

```
  vtkPolyData* surface=vtkPolyData::New();
  surface->SetPoints(pts);
  surface->SetPolys(triangles);

  // Since VTK uses reference counting delete pts and triangles
  // as these are counted inside surface
  pts->Delete();
  triangles->Delete();
```

The rest of the code deals with the display. It is essentially a line-by-line translation of the original Tcl code and there is nothing particular worth highlighting.

```
  vtkPolyDataMapper* map=vtkPolyDataMapper::New();  map->SetInput(surface);

  //Create the actor and set it to display wireframes
  vtkActor* actor=vtkActor::New();
  actor->SetMapper(map);
  map->Delete();
  vtkProperty* property=actor->GetProperty();
  property->SetColor(1,1,1);  property->SetAmbient(1.0);
  property->SetDiffuse(0.0);  property->SetSpecular(0.0);
  property->SetRepresentationToWireframe();

  // Create the renderer
  vtkRenderer* ren= vtkRenderer::New();  ren->AddActor(actor);

  // Set camera mode to Orthographic as opposed to Perspective
  ren->GetActiveCamera()->ParallelProjectionOn();

  // Rest is standard window/interactor etc.
  // Render Window
  vtkRenderWindow* renWin=vtkRenderWindow::New();
  renWin->AddRenderer(ren);  renWin->SetSize(300,300);

  // Interactor
  vtkRenderWindowInteractor* iren=vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);
  iren->Initialize();  iren->Start();
}
```

Compiling the examples: As before use CMAKE. The CMakeLists.txt file is:

```
PROJECT(VTKEXAMPLES)
```

```
INCLUDE (${CMAKE_ROOT}/Modules/FindVTK.cmake)
FIND_PACKAGE(VTK)
IF (USE_VTK_FILE)
  INCLUDE(${USE_VTK_FILE})
ELSE(USE_VTK_FILE)
MESSAGE(FATAL_ERROR,"Please specify VTK_DIR")
ENDIF (USE_VTK_FILE)

LINK_LIBRARIES(
vtkCommon
vtkGraphics
vtkIO
vtkRendering)

ADD_EXECUTABLE(example1 example1.cpp)
ADD_EXECUTABLE(example2 example2.cpp)
ADD_EXECUTABLE(example3 example3.cpp)
ADD_EXECUTABLE(example4 example4.cpp)
```

There are two interesting aspects of this file. First we need to link against the VTK libraries, which means we need to first find them! This is accomplished using the `FIND_PACKAGE(VTK)` command. If VTK is not found we ouput a fatal error message to the user and the process stops.

Once VTK is found, we define a set of libraries that are to be linked to all executables (and libraries if any were defined) using the `LINK_LIBRARIES` command. Finally we add the executables one-by-one. Since they are not described in this text, we mention here that example2.cpp corresponds to script10-3.tcl and example3.cpp corresponds to script10-4.tcl.

18.4 Templates

Templates are a means of creating many similar methods from one piece of code. For example consider a method that returns the maximum value of two inputs. Strictly speaking we would have to write one such function for each type of variable (e.g. float, double, int etc.). Templates are a way to generate these functions automatically from a simple piece of code.[1] The following example (templateexample/main.cpp) illustrates the concept:

```
#include <stdio.h>

// T stands for the arbitrary type
template<class T>
T maxvalue(T a,T b) {
  if (a>=b)
    return a;
  return b;
}
```

[1] One of the downsides of using templates, is that a syntax error in a templated function will result in multiple compiler errors, one for each time the compiler created a new specific function from the templated function! Try writing 'iff' instead of 'if' in the code below and see what happens.

The function maxvalue is designed to accept values of the arbitrary type "T". When invoked (see below) using a specific type (e.g. double), a new function `double maxvalue(double a,double b)` is created (automatically) and used. Hence if there is a syntax error in a templated function, the compiler will return the same error multiple times, one for each new version of the templated function that it has to create.

The main program is below. We call the same function, once with double arguments and once with integer arguments.

```
int main(int argc,char *argv[])
{
  double a=1.0,b=2.0;
  fprintf(stderr,"Maximum of %f, %f =%f\n",a,b,maxvalue(a,b));


  int a1=44,b1=22;
  fprintf(stderr,"Maximum of %d, %d =%d\n",a1,b1,maxvalue(a1,b1));

  return 0;
}
```

The CMakeLists.txt file to compile this is trivial – but it offers non-trivial functionality.

```
PROJECT(DEMO)
ADD_EXECUTABLE(main main.cpp)
```

Templates are used heavily by the C++ Standard Template Library to support many different kinds of objects such as vectors, stacks etc. In addition to templated functions, it is also possible to have templated classes and derived classes. The use of templates enables a style of programming known as generic programming, which is heavily invoked in ITK (we discuss this in Chapter 24). VTK does not use templates in any of its interface code, but templated functions are used as part of the implementation of many filters.

Templated code presents unusual problems to the C++ compiler and many of them are not capable of handling all features of templated programming correctly. An easy solution to most of these problems is to include all definitions of templated functions and implementation in a single file – essentially put the code in the .h file!

18.5 The C++ String Class and C++ Streams

In many respects the code in this handout is old fashioned (i.e. more C-like than modern C++-line) in that it still uses C style strings and IO. The standard template library in C++ defines a modern string class and an alternative, more type-safe, stream based input/output setup (replacing fprintf etc.). We may return to this topic later in the semester. For now, take a look at:

String class – `http://www.cprogramming.com/tutorial/string.html`

and:

C++ File I/O – `http://www.cprogramming.com/tutorial/lesson10.html`

In general, the tutorial at the Cprogramming.com webpage `http://www.cprogramming.com/tutorial/` is a

great resource.

Assignment

This is a rehash of Assignments 7 and 10, but using C++ rather than [Incr] Tcl. You may simply translate your previous Tcl scripts to C++. All solutions must be submitted with an appropriate CMakeLists.txt file for compilation.

- Create an additional class (collegestudent) derived from Teenager. Add at least one additional member variable (e.g. Age) with appropriate Get/Set Methods for modifying and obtaining it's value. Appropriately modify the PrintSelf method to print this also.
- Using example2.cpp as a base, change the code to create a helix, i.e. a curve where the z-coordinate increase with each point as opposed to a constant 0.0. Save the output in a file helix.vtk
- Using the last assignment as a base and the display code from example4.cpp (or your own version) write a script that creates and displays a helix.
- Modify the last assignment to display both a helix and a cone. Hint: you will need two actors.

Part VI

VTK Programming with C++ and Tcl

Chapter 19

Extending VTK using C++

In this section we give examples of some simple VTK classes. In particular we will first demonstrate how to add your own procedural code to VTK – in this case there is no attempt to integrate into the VTK piepline. Next we present two simple image-to-image filters that do integrate with the pipeline. The first filter is one in which the input image has the same size as the output, whereas in the second filter the size of the output image is different from that of the input. Please note that all code is based on VTK 4.4 – there have been some substantial changes in VTK 5.0.

19.1 Introduction

In this chapter we describe how to write your own C++ classes derived from VTK classes.[1] There are two key advantages of deriving classes from VTK classes as opposed to simply writing classes from scratch:

- The ability to write filters that will integrate into the VTK pipeline
- The ability to have your C++ code (if the header is properly written) be wrapped for use with Tcl with no effort on your part.

We will discuss three examples in this class. The first example shows how to add procedural code to VTK – the key trick being the addition of these procedures as static member functions of a class which serves simply as a container. This is used in VTK for the math utility class vtkMath, and it is the only way to add procedural code in pure OOP -languages such as Java. Next we will discuss two image-to-image filters. The first performs a simple thresholding operation, whereas the second resamples the image to extract a volume-of-interest (VOI).

19.2 Adding Procedural Code to VTK

We will create a class vtkMyUtility which will act as a container for different procedures/functions. These functions will be added as static members of vtkMyUtility. The reason for going down this route, as opposed to simply writing procedural code, is that we can leverage the VTK Tcl wrapping code to make our new functions available from Tcl.

The Interface File: The header file (vtkMyUtility.h) is listed below:

[1] The code in tthe examples is writen for VTK 4.4. It may not work (or even compile), as is, on VTK 5.0 as there have been some significant changes in the pipeline architecture. At some point, in the future, I expect to update this code for VTK 5.0 – perhaps once VTK 5.1 comes out.

```
#include <vtkObject.h>
class vtkImageData;

class vtkMyUtility : public vtkObject
{
public:
  static vtkMyUtility *New();
  vtkTypeMacro(vtkMyUtility,vtkObject);

  // Description:
  // A few simple functions
  static double Average(double a,double b);

  /* The BTX and ETX directives enclose code that should not be parsed
     by vtkWrapTcl to generate Tcl code.  Use these to hide stuff that
     might confuse the wrappers.
  */
  //BTX
  static double Average(double* a,int n);
  //ETX
  static vtkImageData* ThresholdImage(vtkImageData* input,double minv,
                                      double maxv,int binary);
protected:
  vtkMyUtility() {};
  virtual ~vtkMyUtility() {};
};
```

The first thing to notice is that we use vtkObject as a parent class. This is one of the most generic classes in VTK and does very little (other than some basic definitions of reference counting), but for our purposes that's OK! The next two methods are the `New` method and the vtkTypeMacro, both of which are boiler-plate code and are needed for all VTK classes.

Next we define the three static methods. The first two simply perform number averaging, whereas the last one is a simple image thresholding procedure. All these methods are public as they are meant to be accessed from the outside.

Finally, in the protected section, we define the constructor and the destructor. These are both dummy functions (i.e. they do nothing), but they must be defined as protected to prevent their default use from the outside – as this would break the reference counting scheme.

The Implementation File: The implementation file (vtkMyUtility.cpp) is listed below. The first part simply includes the appropriate header files:

```
#include "vtkMyUtility.h"
#include "vtkObjectFactory.h"
#include <stdio.h>
#include "vtkImageData.h"
#include "vtkDataArray.h"
#include "vtkPointData.h"
```

Next we use a standard VTK macro to generate the static member New function appropriately.

```
vtkStandardNewMacro(vtkMyUtility);
```

Next we define the two average methods, nothing exciting here:

```
// This function is accessible both from C++ and Tcl
double vtkMyUtility::Average(double a,double b) {
  return 0.5*(a+b);
}

// This function is only accessible from C++
double vtkMyUtility::Average(double* a,int n) {
  if (a==NULL)  return 0.0;
  double sum=0.0;
  for (int i=0;i<n;i++)
    sum+=a[i];
  return sum/double(n);
}
```

Finally the thresholding function. Here, a key step is to make sure that a valid image has been passed as an input. This is key, our code has to be defensively oriented and assume that the user is "out to get us". Next we create an output image of the same dimensions and type as the input using the `CopyStructure` and `AllocateScalars` methods.

After this, we simply get the two data arrays and do the thresholding voxel-by-voxel:

```
vtkImageData* vtkMyUtility::ThresholdImage(vtkImageData* input,double minv,
                                           double maxv,int binary)
{
  if (input==NULL) {
      fprintf(stderr,"Bad Input Image\n");  return NULL;
    }

  vtkImageData* output=vtkImageData::New();
  output->CopyStructure(input);
  output->AllocateScalars();

  vtkDataArray* inarray=input->GetPointData()->GetScalars();
  vtkDataArray* outarray=output->GetPointData()->GetScalars();

  int numcomp=inarray->GetNumberOfComponents();
  int numvox =inarray->GetNumberOfTuples();

  for (int component=0;component<numcomp;component++) {
    outarray->FillComponent(component,0.0);
    for (int i=0;i<numvox;i++) {
      double out=0.0;
      double v=inarray->GetComponent(i,component);
      if (v>=minv && v<=maxv) {
```

```
        if (binary)
          out=1.0;
        else
          out=v;
        outarray->SetComponent(i,component,out);
       }
    }
  }
  return output;
}
```

19.3 A Simple Thresholding Filter

While the thresholding function in the previous section is perfectly adequate for most tasks, such code can not be directly integrated into the usual VTK pipeline as a regular filter.

Interface: To accomplish this we rewrite our thresholding filter as a derived class of vtkSimpleImageToImageFilter[2] The header file (vtkImageSimpleThreshold.h) is listed below. First the usual class header etc. Note that the vtkTypeMacro takes two arguments, the current class names and the name of its parent class.

```
#include "vtkSimpleImageToImageFilter.h"
#include "vtkImageData.h"

class vtkImageSimpleThreshold : public vtkSimpleImageToImageFilter
{
public:
  static vtkImageSimpleThreshold *New();
  vtkTypeMacro(vtkImageSimpleThreshold,vtkSimpleImageToImageFilter);
```

Next we define some interface members, using the vtkGetMacro and vtkSetMacro macros. These make the parsing of the C++ code to create Tcl code easier. The —vtkGetMacro(VariableName,type)— creates the function:

```
type GetVariableName()
```

whereas the vtkSetMacro, creates the function:

```
void SetVariableName(type T) { this->VariableName=T;}
```

One additional complexity is provided by the vtkSetClampMacro which makes sure that the new variable set in the valid range (in the case of BinaryOutput below, the range is 0 to 1). If the user specifies a value below the minimum or above the maximum allowed values, the variable is set to the minimum or maximum allowed value respectively. Finally the vtkBooleanMacro creates two functions of the form `BinaryOutputOn(), and BinaryOutputOff()` that result in the variable BinaryOutput being set to 1 or zero respectively.

```
  // Description:
  // Set the thresholds between which output is 1
```

[2]This very useful teaching tool class has unfortunately been removed from VTK 5.0.

```
  vtkGetMacro(LowerThreshold,float);
  vtkSetMacro(LowerThreshold,float);
  vtkGetMacro(UpperThreshold,float);
  vtkSetMacro(UpperThreshold,float);

  // Description:
  // If Binary = 1, then output is 0,1 else output is 0, original
  vtkSetClampMacro(BinaryOutput,int,0,1);
  vtkGetMacro(BinaryOutput,int);
  vtkBooleanMacro(BinaryOutput,int);
```

In the protected part of the definition, we first define the constructor, which in this case is not a dummy function, and make sure that things like the copy construrctor and the "=" operator are protected, to prevent their use which will mess up the reference counting scheme.

```
protected:

  vtkImageSimpleThreshold();
  vtkImageSimpleThreshold(const vtkImageSimpleThreshold&) {};
  void operator=(const vtkImageSimpleThreshold&) {};
```

Finally we define the `SimpleExecute` function which will be called when this filter is updated (this is defined in the parent class) and the three member variables.

```
  // Description:
  // Do Thresholding Operation
  virtual void SimpleExecute(vtkImageData* inp,vtkImageData* out);

  //BTX
  float       LowerThreshold,UpperThreshold;
  int         BinaryOutput;
  //ETX
};
```

Implementation: The first part is the usual, include the header files and call the vtkStandardNewMacro, so we will skip it. Next we define the constructor, whose job it is to set the default values for all member variables and to perform any other initialization:

```
vtkImageSimpleThreshold::vtkImageSimpleThreshold()
{
  this->LowerThreshold=-1.0e+9;
  this->UpperThreshold= 1.0e+9;
  this->BinaryOutput=1;
}
```

The work of the filter is performed by the SimpleExecute function. This is very similar to the function vtkMyUtility::ThresholdImage, but here the code is placed in the context of the VTK pipeline. Also since this is not a static member function, but a proper member function, we can use the vtkErrorMacro to report errors (e.g. in this case bad input). The code is below:

```
void vtkImageSimpleThreshold::SimpleExecute(vtkImageData* input,vtkImageData* output )
{
  if (input==NULL) {
      vtkErrorMacro(<<"Bad Input to vtkImageSimpleThreshold");
      return;
    }

  int dim[3]; input->GetDimensions(dim);
  int numvox=dim[0]*dim[1]*dim[2];
  int numcomp=input->GetNumberOfScalarComponents();

  vtkDataArray* inarray=input->GetPointData()->GetScalars();
  vtkDataArray* outarray=output->GetPointData()->GetScalars();

  for (int component=0;component<numcomp;component++) {
     outarray->FillComponent(component,0.0);
     for (int i=0;i<numvox;i++) {
        double out=0.0;
        double v=inarray->GetComponent(i,component);
        if (v>=this->LowerThreshold && v<=this->UpperThreshold) {
           if (this->BinaryOutput)
              out=1.0;
           else
              out=v;
           outarray->SetComponent(i,component,out);
         }
       }
    }
}
```

19.4 A Slightly More Complex Filter

In the previous example, we derived the simplest possible type of image-to-image filter, one in which the image output has the same type and dimensions as the input image. If either of these last two conditions do not hold (i.e. image dimensions or type are different), then we need to let the pipeline controlling processes know, by overriding the ExecuteInformation method. As an example, consider the vtkImageExtractVOI class which takes an image as an input and extracts a piece of it – the volume of interest or VOI – as the output. In this case we, potentially, have a change in image size.

The Interface: The interface (vtkImageExtractVOI.h) is listed below. The only two highlights here are: (i) the use of vtkGetVectorMacro and vtkSetVectorMacro in the case of fixed length arrays, in this case the volume of interest VOI (int[6]) and the sampling rate SampleRate (int[3]) and (ii) the presence of the ExecuteInformation method.

```
#include "vtkSimpleImageToImageFilter.h"
class vtkImageExtractVOI : public vtkSimpleImageToImageFilter
{
public:
  vtkTypeMacro(vtkImageExtractVOI,vtkSimpleImageToImageFilter);
  void PrintSelf(ostream& os, vtkIndent indent);

  static vtkImageExtractVOI *New();

  // Description:
  // Specify i-j-k (min,max) pairs to extract. The resulting structured points
  // dataset can be of any topological dimension (i.e., point, line, image or volume).
  vtkSetVector6Macro(VOI,int);
  vtkGetVectorMacro(VOI,int,6);

  // Description:
  // Set the sampling rate in the i, j, and k directions.
  vtkSetVector3Macro(SampleRate, int);
  vtkGetVectorMacro(SampleRate, int, 3);

protected:
  vtkImageExtractVOI();
  ~vtkImageExtractVOI() {};
  vtkImageExtractVOI(const vtkImageExtractVOI&) {};
  void operator=(const vtkImageExtractVOI&) {};

  // Description:
  // Execute Stuff
  virtual void  ExecuteInformation();
  virtual void  SimpleExecute(vtkImageData* input, vtkImageData* output);

  // Description:
  // Data Members
  int VOI[6];
  int SampleRate[3];
};
```

Implementation: The listing for vtkImageExtractVOI.cpp is listed below. We skip the header insertion and standard macro invocation. The constructor sets the initial values in the usual way:

```
vtkImageExtractVOI::vtkImageExtractVOI()
{
  this->VOI[0] = this->VOI[2] = this->VOI[4] = 0;
  this->VOI[1] = this->VOI[3] = this->VOI[5] = 1;
  this->SampleRate[0] = this->SampleRate[1] = this->SampleRate[2] = 1;
}
```

Next up is the PrintSelf macro, which prints the contents of the member variables. This is very useful for debugging purposes (we skipped this for the previous two classes which is not ideal!)

```
void vtkImageExtractVOI::PrintSelf(ostream& os, vtkIndent indent)
{
  vtkSimpleImageToImageFilter::PrintSelf(os,indent);
  os << indent << "  VOI " << this->VOI[0] << ":" << this->VOI[1] << " ";
  os << indent << "  VOI " << this->VOI[2] << ":" << this->VOI[3] << " ";
  os << indent << "  VOI " << this->VOI[4] << ":" << this->VOI[5] << "\n";
  os << indent << "  SampleRate "<<this->SampleRate[0]<<''
  os << ","<<this->SampleRate[1]<<","<<this->SampleRate[2]<<"\n";
}
```

Next up is the ExecuteInformation Method. First we call the parent class to do the defautls. Then we set the Spacing, Origin, Dimensions and the WholeExtent (an array of six numbers 0, dim[0]-1,0,dim[1]-1,0,dim[2]-1) of the output image to reflect the user settings for the VOI and the SampleRate.

```
void vtkImageExtractVOI::ExecuteInformation()
{
  int i, outDims[3], voi[6], rate[3], wholeExtent[6];
  this->vtkSimpleImageToImageFilter::ExecuteInformation();

  vtkImageData* input=this->GetInput();
  vtkImageData* output=this->GetOutput();
  if (this->GetInput() == NULL) { vtkErrorMacro("Missing input");     return;    }

  input->GetWholeExtent( wholeExtent );
  for ( i=0; i < 6; i++ )
     voi[i] = this->VOI[i];

  for ( i=0; i < 3; i++ ) {
     if ( voi[2*i+1] > wholeExtent[2*i+1] ) { voi[2*i+1] = wholeExtent[2*i+1]; }
     else if ( voi[2*i+1] < wholeExtent[2*i] ) { voi[2*i+1] = wholeExtent[2*i]; }

     if ( voi[2*i] < wholeExtent[2*i] ) { voi[2*i] = wholeExtent[2*i]; }
     else if ( voi[2*i] > wholeExtent[2*i+1] ) { voi[2*i] = wholeExtent[2*i+1]; }

     if ( voi[2*i] > voi[2*i+1] ) { voi[2*i] = voi[2*i+1]; }

     if ( (rate[i] = this->SampleRate[i]) < 1 ) { rate[i] = 1; }

     outDims[i] = (voi[2*i+1] - voi[2*i]) / rate[i] + 1;
     if ( outDims[i] < 1 ) outDims[i] = 1;
   }

  wholeExtent[0] = 0;  wholeExtent[1] = outDims[0] - 1;
  wholeExtent[2] = 0;  wholeExtent[3] = outDims[1] - 1;
  wholeExtent[4] = 0;  wholeExtent[5] = outDims[2] - 1;
  output->SetDimensions(outDims);
  output->SetWholeExtent( wholeExtent );

  double spacing[3],origin[3]; input->GetSpacing(spacing); input->GetOrigin(origin);
```

```
  output->SetSpacing(spacing[0]*rate[0],spacing[1]*rate[1],spacing[2]*rate[2]);
  output->SetOrigin(voi[0]*spacing[0]+origin[0],voi[2]*spacing[1]+origin[1],
    voi[4]*spacing[2]+origin[2]);
}
```

Finally the SimpleExecute method itself. Here we "cheat" and simply call vtkImageReslice, since it can do the job just fine.

```
void vtkImageExtractVOI::SimpleExecute(vtkImageData* input, vtkImageData* output)
{
  if (input==NULL)  { vtkErrorMacro(<< "Bad Input");      return;    }
  int dim[3]; output->GetDimensions(dim);
  vtkImageReslice* resl=vtkImageReslice::New();
  resl->SetInput(input);
  resl->SetOutputOrigin(output->GetOrigin());
  resl->SetOutputSpacing(output->GetSpacing());
  resl->SetInterpolationMode(0);
  resl->SetOutputExtent(0,dim[0]-1,0,dim[1]-1,0,dim[2]-1);
  resl->OptimizationOff();
  resl->Update();
  output->ShallowCopy(resl->GetOutput());
  resl->Delete();
}
```

19.5 Compiling and Using

The CMakeLists.txt for this project is below. The first part is fairly standard:

```
# create the vtk executable
PROJECT(MYLIB)

SET(KITBASE MyLib)
SET(KIT     vtk${KITBASE})

INCLUDE (${CMAKE_ROOT}/Modules/FindVTK.cmake)
FIND_PACKAGE(VTK REQUIRED)
IF (USE_VTK_FILE)
  INCLUDE(${USE_VTK_FILE})
ENDIF (USE_VTK_FILE)

INCLUDE_DIRECTORIES(${MYLIB_SOURCE_DIR})
# Set the default location for outputting the library
SET (LIBRARY_OUTPUT_PATH ${MYLIB_SOURCE_DIR})

SET(LIBRARY_SRCS
vtkMyUtility.cpp
```

```
vtkImageSimpleThreshold.cpp
vtkImageExtractVOI.cpp)

LINK_LIBRARIES(vtkCommon vtkCommonTCL vtkImaging vtkImagingTCL)
ADD_LIBRARY(${KIT} STATIC ${LIBRARY_SRCS})
```

The exciting part comes at the end. First we use the VTK_WRAP_TCL command to wrap all the header files in LIBRARY_SRCS to create Tcl interface code. Next we add one more library that is the dynamic library that can be loaded into the vtk interpreter. This is similar to the cpphelloworld example of Chapter 17.

```
VTK_WRAP_TCL (${KIT}TCL LIBRARY_TCL_SRCS ${LIBRARY_SRCS})
ADD_LIBRARY (${KIT}TCL SHARED ${LIBRARY_TCL_SRCS} ${LIBRARY_SRCS})
```

Example 1: Once the library exists it can be loaded and exercised. See scripts19-1.tcl, script19-2.tcl script19-3.tcl and script19-4.tcl for examples. The simplest on of these (script19-1.tcl) is listed below:

```
lappend auto_path [ file dirname [ info script ]]
package require newname 1.0

if { $tcl_platform(platform) == "windows" } {
    load debug/vtkMyLibTCL.dll
} else {
    load libvtkMyLibTCL.so
}

set util [ vtkMyUtility  [ newname::vnewobj ] ]
set out  [ $util Average 4 11 ]
puts stderr "The average of 4 and 11 is $out"
exit
```

The library is loaded (much like the examples in Chapter 17) using the load command. Then a new vtkMyUtility object is created. This is then used to invoke the average command. (While in C++ we do not need an instance of the object to invoke a static member function, e.g. the Average function could have been invoked as `vtkMyUtility::Average(4,11)`, in Tcl this is necessary – most likely a limitation of the VTK Tcl Wrappers.

Example 2: The following script (script19-3.tcl) illustrates the use of the vtkImageSimpleThereshold class to threshold an image and generate an output binary mask:

```
lappend auto_path [ file dirname [ info script ]]
package require loadbioimagesuite 1.0
if { $tcl_platform(platform) == "windows" } {
    load debug/vtkMyLibTCL.dll
} else {
```

```
    load libvtkMyLibTCL.so
}
wm withdraw  .

set ana [ vtkpxAnalyzeImageSource [ pxvtable::vnewobj ]]
$ana Load axial.hdr

set img [ $ana GetOutput ]
set orient [ $ana GetOrientation ]
puts stderr "Image Dimensions = [ $img GetDimensions ], Orientation = $orient"

set thr [ vtkImageSimpleThreshold  [ pxvtable::vnewobj ]]
$thr SetInput $img
$thr SetLowerThreshold 150
$thr SetUpperThreshold 250
$thr BinaryOutputOn
$thr Update

set anaw [ vtkpxAnalyzeImageWriter [ pxvtable::vnewobj ]]
$anaw SetInput [ $thr GetOutput ]
$anaw SetOrientation $orient
$anaw Save thr.hdr
```

Bot theese examples fully illustrated the dual-language approach, write in C++, use in Tcl.

19.6 Overall Comments

In general the style of programming that I am advocating (and this class is designed to teach) is one in which core components/classes are written in C++ for reasons of efficiency and reuse. Such components are then to be packaged into shared libraries and wrapped such that they can be accessed from Tcl. The main program, including the user interface should most likely be written in Tcl – optimally using the [Incr Tcl] extensions – which ensures both platform portability as well as relative ease of development. It is fairly common to observe that the majority of tweaking of a piece of software has to do with the user interface, so the use of an interpreted language such as Tcl can have a great potential benefit in this respect.

The other major advantage of this approach is the separation between algorithms and applications. In this way the algorithms (neatly packaged in shared libraries) can be reused by other applications by simply changing the user interface. Essentially a programmer develops: (i) a toolkit - a collection of useful tools and (ii) an application or set of applications based on the toolkit with additional user interface enhancements. This is the philosophy used in the design and development of BioImage Suite, for example.

Assignment 19

First ensure that you can compile and run the examples in this chapter. Next:

- Add two new methods to vtkMyUtility.h. One should compute the average intensity in an image. The second should return the maximum and minimum intensity in an image. (You should most likely return a vtkFloatArray in the second case.)
- Create a new class (optionally based on vtkImageSimpleThreshold) that takes an image as an input and two thresholds t_1 and t_2. The output should be 0 if the intensity is less than t_1, 1 if the intensity is between

t_1 and t_2 and 2 if the intensity is greater than t_2. The class should ensure that $t_1 < t_2$.

- Create a new class (optionally based on vtkImageExtractVOI.h) that samples an image and cuts its size in half by talking every other voxel in x,y and z-directions respectively. For example a $16 \times 24 \times 32$ input image should result in an $8 \times 12 \times 16$ image output.
- Add your classes to the CMakeLists.txt file and write Tcl scripts that exercise these new classes.

Chapter 20

Point-based Registration with ICP

In this Chapter, we will discuss the implementation of one of the most popular surface matching algorithms: the Iterative Closest Point algorithm (ICP)[4]. Part of the design exercise will involve the division of the implementation into two parts: (i) an abstract parent class containing functionality common to point-based registration methods and (ii) a derived class which implements the ICP algorithm. In addition, we will also look at the design of a graphical user interface for interacting with this algorithm. Please note that all code is based on VTK 4.4 – there have been some substantial changes in VTK 5.0.

20.1 The Iterative Closest Point Algorithm

We review here, briefly, the Iterative Closest Point algorithm [4]. This is a registration method whose goal is to estimate a transformation T that best aligns two point sets $X = x_0, x_1, \ldots, x_n$ and $Y = y_0, y_1, \ldots, y_m$, where in general $n \neq m$. The "exciting" part of this algorithm is that it does not assume that the correspondences between X and Y are known, if they were known, the problem reduces to a simple least squares fit of some form to estimate the transformation. Alternatively, if the transformation T were known, it could be used to estimate the corresponding points. The way out of this chicken-and-egg problem is to perform an alternating estimation of (i) the set of correspondences and (ii) the transformation. More formally the algorithm iteratively minimizes the following two equations (k=number of iteration), until convergence:

$$\text{Correspondence:} \quad c_i^k = \underset{y_j \in Y}{\arg\min} |T^k(x_i) - y_j|^2 \tag{20.1}$$

$$\text{Transformation:} \quad T^{k+1} = \underset{T}{\arg\min} \sum_{i=1}^{n} |T(x_i) - c_i^k|^2 \tag{20.2}$$

In the correspondence estimation step, we take each point $x_i \in X$, and transform it using the current estimate of the transformation T^k. Then we look for the point in Y that is closest to $T^k(x_i)$. We label this point as the corresponding point, at iteration k, c_i^k. This results in a sets of pairs of correspoding points (x_i, c_i^k).

In the transformation estimation step, we look for the transformation T that best explains (or describes) this set of correspondences. T can be either a linear or a non-linear transformation, we will focus on linear transformations in this Chapter.

Once a better version of T is estimated we can use it to estimate a better set of correspondences and so on until convergence. Note that ICP needs to be initialized fairly close to the true transformation for convergence. Alternative methods, e.g. the Robust Point Matching method [6] have superior capture range and accuracy. However, ICP is a simple algorithm which works OK and is a great classroom example.

20.2 Implementation Design

While there is already an existing implementation of ICP in VTK 4.4 – vtkIterativeClosestPointTransform – from which some of the code used in our design is based, we will describe a different approach here. The goal of this approach, is to develop both a generic framework for point-based registration methods as well as an implementation for ICP itself. This is a useful exercise as it demonstrates how object-oriented methodology can be used to implement a family of algorithms in a clean way.

Generic vs Specific Functionality: In designing our implementation, we can list the following methods and variables that need to be specified/implemented:

1. Source and Target Data sets
2. Parameters such as the maximum number of iterations, convergence threshold, maximum number of points.
3. Methods for invoking the algorithm and returning the output transformation in an algorithm-independent way (so the rest of the program can use different algorithms with no major changes.)
4. Utility methods for sampling a data-set (to reduce the number of points), estimating the centroid of a data set and translating points.
5. Parameters specific to the Linear ICP algorithm, such as the exact type of the transformation (rigid, similarity or affine), the initial step (matching centroids).
6. The method for the main loop of the ICP algorithm.

If we look at this list, one way to break it up is that the first four items refer to functionality that is generally useful for point-based registrations, whereas the last two describe linear-ICP specific functionality. A good way to break up the design, is then to implement first an abstract class that incorporates items 1–4 and then derive from it an ICP implementation that defines 5–6 and overrides the methods in 3. We discuss both of these class next:

The abstract parent class – vtkAbstractPointBasedRegistration

Formally speaking, an abstract class in C++, is one which can never be instantiated. It is simply there as a place to derive functional classes from. Abstract classes are extremely useful for defining a common interface to more concrete children classes, such that other parts of the program may interact with any number of these derived classes using a similar interface.

The Interface: The interface file (vtkAbstractPointBasedRegistration.h) has the following form. We derive this class from vtkProcessObject which is a fairly lightweight VTK object[1] that does not impose too many constraints.

```
class vtkAbstractPointBasedRegistration : public vtkProcessObject
{
public:
  vtkTypeRevisionMacro(vtkAbstractPointBasedRegistration,vtkProcessObject);
  void PrintSelf(ostream& os, vtkIndent indent);
```

Next comes the specification of the two data-sets, the source dataset (X) and the target data-set (Y):

[1] Perhaps in VTK 5.0, such a class could be derived from the new vtkAlgorithm class.

```
  vtkGetObjectMacro(Source, vtkDataSet);
  vtkGetObjectMacro(Target, vtkDataSet);
  vtkSetObjectMacro(Source, vtkDataSet);
  vtkSetObjectMacro(Target, vtkDataSet);
```

The GetObjectMacro simply creates functions of the form:

```
 vtkDataSet* GetSource() { return this->Source; }
```

The SetObjectMacro is more interesting. In full this generates a function (we abbreviate somewhat here) that takes the form:

```
virtual void SetSource(vtkDataSet* ds) {
  if (this->Source!=NULL)
       this->Source->UnRegister(this);
    this->Source = ds;
    if (this->Source != NULL)
       this->Source->Register(this);
    this->Modified();
}
```

The Register/UnRegister pair increment and decrement, respectively, the reference count on the objects. In setting the new source object, we first decrement the reference count of the previous object used as source. Next we set the new source object and and then increment its reference counter!

Next come a set of methods for setting and getting the value of various parameters:

```
  vtkSetMacro(MaximumNumberOfIterations, int);
  vtkGetMacro(MaximumNumberOfIterations, int);
  vtkSetMacro(MaximumNumberOfPoints, int);
  vtkGetMacro(MaximumNumberOfPoints, int);
  vtkSetMacro(Epsilon, double);
  vtkGetMacro(Epsilon, double);
```

Finally, two extremely important functions, the `GetTransformation()` and the `Run()` methods. These are defined as pure virtual methods (signified by the `=0;`) at the end, which means two things: (i) this class can not be instatiated and (ii) any derived class from this class must define these methods in order to be able to be instantiated. This is a formal way, in C++, of defining a pure interface for derived classes to implement:

```
  // Get the output transformation. This is purposefully an abstract
  // transformation so That derived classes can return different types of transformations
  virtual vtkAbstractTransform* GetTransformation() = 0;
  // The Run Method Computes the Registration
  virtual int Run() = 0;
```

The rest of the header is fairly trivial:

```
protected:
  vtkAbstractPointBasedRegistration();
  virtual ~vtkAbstractPointBasedRegistration();
  // Data Members
  vtkDataSet* Source,*Target;
  int MaximumNumberOfIterations, MaximumNumberOfPoints;
  double Epsilon;
  // Utility Method
  virtual   vtkPoints* SampleDataSet(vtkDataSet* input,int NumberOfPoints);
  virtual   void       GetCentroid(vtkDataSet* input,double centroid[3]);
  virtual   void       ShiftPoints(vtkPoints* input,double shift[3],double scale=1.0);
private:
  vtkAbstractPointBasedRegistration(const vtkAbstractPointBasedRegistration&);
  void operator=(const vtkAbstractPointBasedRegistration&);
};
```

The Implementation: We next highlight some key points in the implementation (vtkAbstratPointBasedRegistration.cpp). First the constructor, which sets the default values for all the parameters (including the *critical* NULL settings for all pointer input variables.)

```
vtkAbstractPointBasedRegistration::vtkAbstractPointBasedRegistration() : vtkProcessObject()
{
  this->Source = NULL; this->Target = NULL;
  this->MaximumNumberOfIterations = 50;
  this->MaximumNumberOfPoints = 200;  this->Epsilon=0.001;
}
```

The destructor simply sets the input pointers to NULL, this is a way of decrementing the reference count of any objects used as a Source and Target up to this point:

```
vtkAbstractPointBasedRegistration::~vtkAbstractPointBasedRegistration() {
  this->SetSource(NULL);  this->SetTarget(NULL);
}
```

The rest of the code is fairly straightforward and should be easy to follow.

The concrete derived class – vtkLinearICPRegistration:

This class implements a form of the ICP algorithm that estimates a linear transformation.

The Interface: The header file (vtkLinearICPRegistration.h) has the form. The first part is straight-forward:

```
class vtkLinearICPRegistration : public vtkAbstractPointBasedRegistration
{
public:
  static vtkLinearICPRegistration *New();
  vtkTypeRevisionMacro(vtkLinearICPRegistration,vtkAbstractPointBasedRegistration);
  void PrintSelf(ostream& os, vtkIndent indent);

  // Starts the process by translating source centroid to target centroid.
  vtkSetMacro(StartByMatchingCentroids, int);
  vtkGetMacro(StartByMatchingCentroids, int);
  vtkBooleanMacro(StartByMatchingCentroids, int);
```

The specification of the transformation type is marginally more interesting, in that we define three more convenience methods. These are useful for making the code more readable to a human!

```
  // Transformation Type, Rigid, Similarity, Affine
  vtkGetMacro(TransformationType,int);
  vtkSetClampMacro(TransformationType,int,0,2);
  // Three convenience methods
  virtual void SetTransformationTypeToRigid() { this->SetTransformationType(0);}
  virtual void SetTransformationTypeToSimilarity() { this->SetTransformationType(1);}
  virtual void SetTransformationTypeToAffine() { this->SetTransformationType(2);}
```

Next we explicitly override the two pure virtual methods; this is the heart of the new functionality.

```
  virtual vtkAbstractTransform* GetTransformation();
  virtual int Run();
```

The rest of the definition is fairly straight-forward. Our algorithm will store the transformation in an instance of `vtkTransform`

```
protected:
  vtkLinearICPRegistration();
  virtual ~vtkLinearICPRegistration();

  // Data Members
  int StartByMatchingCentroids, TransformationType;
  vtkTransform          *OutputTransformation;

private:
  vtkLinearICPRegistration(const vtkLinearICPRegistration&);  // Not implemented.
  void operator=(const vtkLinearICPRegistration&);  // Not implemented.
};
```

The Implementation: First the constructor/destructor pair. There is one pointer data member that is allocated in the constructor and deleted in the destructor. (This symmetry is a useful check, make sure that anything created in the constructor is deleted in the destructor!)

```
vtkLinearICPRegistration::vtkLinearICPRegistration() : vtkAbstractPointBasedRegistration(){
  this->StartByMatchingCentroids=1;  this->TransformationType=2;
  this->OutputTransformation=vtkTransform::New();
}
vtkLinearICPRegistration::~vtkLinearICPRegistration(){
  this->OutputTransformation->Delete();
}
```

Next we define one of the two originally pure-virtual methods, the GetTransformation method. We simply return the member variable OutputTransformation. Here, we have an example of what is known in C++ as polymorphism. The returned variable (OutputTransformation) is not an instance of vtkAbstractTransform, but an instance of vtkTransform which is a descendent of vtkAbstractTransform. You can always safely return an instance of derived class in the place of an instance of the specified class. (Naturally the reverse is not possible, one cannot return a vtkAbstractTransform when a vtkTransform is requested!)

```
vtkAbstractTransform* vtkLinearICPRegistration::GetTransformation() {
  return this->OutputTransformation;
}
```

The heart of the implementation is the `Run()` method. We will omit some of the code here. We will use the notation `// OMITTED: code to do something` where this takes place. First we get our act together by checking that the input data sets exist. Then we initialize two key objects: (i) the `Locator` object for quickly finding nearest points and (ii) the `LandmarkTransform` for estimating transformations from sets of corresponding points. Next we get the OutputTransformation in shape (note the *PostMultiply* call!) and sample the input data-set to reduce the number of points for computational reasons (The `SampledSourcePoints` are the x's of the equations). Finally, we allocate the `CorrespondingPoints` (the c's of the equations):

```
int  vtkLinearICPRegistration::Run() {
  // OMITTED: Code to check that inputs are OK
  // Create locator
  vtkPointLocator* Locator = vtkPointLocator::New();
  Locator->SetDataSet(this->Target);  Locator->BuildLocator();
  // Get The Landmark Transform All Set
  vtkLandmarkTransform* LandmarkTransform=vtkLandmarkTransform::New();
  switch(this->TransformationType)  {
    case 0:  LandmarkTransform->SetModeToRigidBody();      break;
    case 1:  LandmarkTransform->SetModeToSimilarity();     break;
    case 2:  LandmarkTransform->SetModeToAffine();         break;
  }
  this->OutputTransformation->Identity();  this->OutputTransformation->PostMultiply();

  // Get The Point Sets Ready
```

```
  vtkPoints* SampledSourcePoints=this->SampleDataSet(this->Source,
                                          this->MaximumNumberOfPoints);
  vtkPoints* CorrespondingPoints=vtkPoints::New();
  CorrespondingPoints->SetNumberOfPoints(SampledSourcePoints->GetNumberOfPoints());
```

The next step performs some initial alignment by matching the centroids, this can oftentimes be helpful as a crude estimate of the overall translation:

```
  double offset[3] = { 0.0,0.0,0.0};
  if (this->StartByMatchingCentroids) {
      double source_centroid[3]; this->GetCentroid(this->Source,source_centroid);
      double target_centroid[3]; this->GetCentroid(this->Target,target_centroid);
      for (int ia=0;ia<=2;ia++)
offset[ia]=target_centroid[ia]-source_centroid[ia];
      this->ShiftPoints(SampledSourcePoints,offset,1.0);
  }
  int     NumberOfLandmarks=SampledSourcePoints->GetNumberOfPoints();
  int     NumberOfIterations = 1;
```

With the preliminaries out of the way we now move on to the alternating estimation itself. The `UpdateProgress` method is used to provide feedback to the GUI – we will see how to respond to this from the Tcl-based graphical user interface later.

```
  // Provide Feedback to GUI
  this->UpdateProgress(0.0);
  // Begin Alternating Estimation
  double previousmaxdist=0.0;
  while (NumberOfIterations <= this->MaximumNumberOfIterations)    {
```

The first part is the correspondence estimation. Note the use of the `Locator` object which makes a complex search problem trivial!

```
  // 1. Find Correspondences
  for(int i = 0; i < NumberOfLandmarks;i++) {
    // Get a Point
    double x[3]; SampledSourcePoints->GetPoint(i,x);
    // Find where this point was at the last estimate!
    double tx[3]; LandmarkTransform->TransformPoint(x,tx);
    // Find Corresponding point using locator
    int id=Locator->FindClosestPoint(tx);
    CorrespondingPoints->SetPoint(i,this->Target->GetPoint(id));
    }
```

Next we estimate the transformation using an instance of `vtkLandmarkTransform`.

```
      LandmarkTransform->SetSourceLandmarks(SampledSourcePoints);
      LandmarkTransform->SetTargetLandmarks(CorrespondingPoints);
      LandmarkTransform->Update();
```

We concatenate this with any initial offset (from pre-aligning the centroids) to get the estimate of the complete transformation, and call `UpdateProgress` to notify the user interface that we have "news":

```
      this->OutputTransformation->Identity();
      this->OutputTransformation->Translate(offset);
      this->OutputTransformation->Concatenate(LandmarkTransform);
      double progress=double(NumberOfIterations)/double(this->MaximumNumberOfIterations);
      this->UpdateProgress(progress);
```

Next comes some code for checking for convergence. When we are done, we clean up any temporary objects allocated in the `Run` method.

```
      // OMITTED: Code to check for convergence
      vtkDebugMacro(<<"End of Iteration " << NumberOfIterations <<"\n");
      ++NumberOfIterations;
    }
  Locator->Delete();  SampledSourcePoints->Delete();
  CorrespondingPoints->Delete();  LandmarkTransform->Delete();
  this->UpdateProgress(1.0);
  return 1;
}
```

Compiling using CMake and Loading into Tcl

We use a fairly straight-forward CMakeLists.txt. The only new point here, is that we need to explicitly tell the VTK Tcl Wrappers that vtkAbstractPointBasedRegistration as an abstract class (i.e. it cannot be instantiated). This is accomplished using the `SET_SOURCE_FILES_PROPERTIES` command as shown below:

```
SET(LIBRARY_SRCS
vtkAbstractPointBasedRegistration.cpp
vtkLinearICPRegistration.cpp
)

SET_SOURCE_FILES_PROPERTIES(
vtkAbstractPointBasedRegistration.cpp
ABSTRACT
```

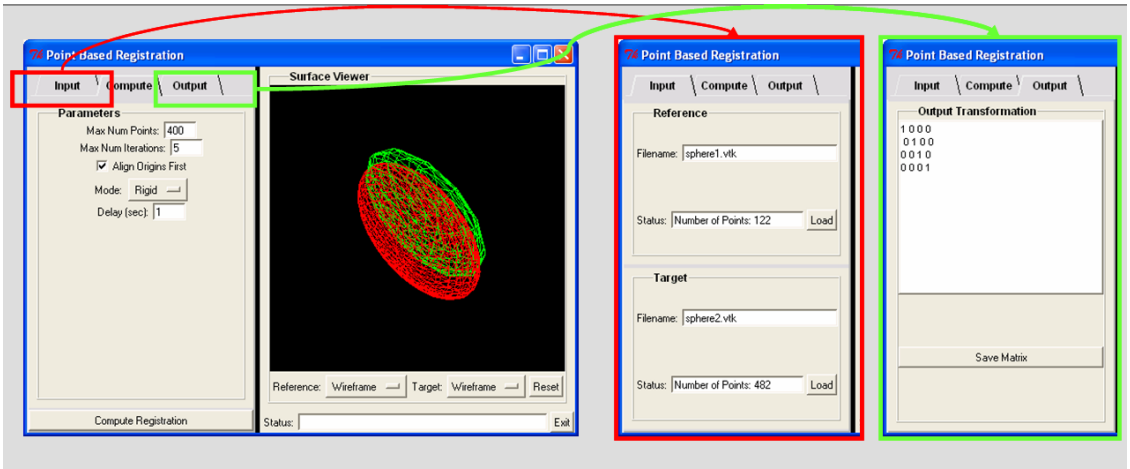


Figure 20.1: The GUI for the ICP algorithm.

)

The end result of the compilation is a shared library that can be accessed from our Tcl user-interface code. This is loaded in the usual way as demonstrated by the package file (`loadpointreg.tcl`) below:

```
package provide loadpointreg 1.0
if { $tcl_platform(platform) == "windows" } {
    set name debug/vtkPointRegTCL.dll
} else {
    set name libvtkPointRegTCL.so
}
load $name;unset name
```

On MS-Windows, you should change the "debug" to "release", if you compile the release version of the DLL. There is often a substantial speed up upon switching from debug to release versions.

20.3 Designing the Graphical User Interface

One of the goals of this class is to provide you with the skills to implement a complete application. This includes both the algorithms and the graphical user interface which makes using the algorithms easier.

The overall GUI is shown in Figure 20.1. It consists of a viewer on the right and a set of controls for setting parameters etc. on the left. The GUI is implemented using two simple [Incr] Tcl Classes, the SimpleSurfaceViewer class (simplesurfaceviewer.tcl) which captures the functionality of the viewer (on the right) and the master class PointBasedRegistration (pointbased.tcl) which is the overall control that contains the viewer.

We will not go through this code line-by-line. Instead, we will focus on some key elements.

The Simple Surface Viewer

We will use the vtkTkRenderWidget class to embed a vtk Viewer into our Tcl-based GUI. This has some issues with the standard vtk Interactors, a workaround is provided by the vtkinteraction Tcl package – this is part of the

VTK distribution. The header includes all the usual good stuff, including the request for vtkinteraction.

```
package provide SimpleSurfaceViewer 1.0
lappend auto_path [ file dirname [ info script ]]
package require vtk
package require vtkinteraction
package require Itcl
package require Iwidgets
package require newname
```

The class definition follows. Essentially we keep handles to the following objects: (i) referencesurface and targetsurface – these are vtkPolyData objects of the two surfaces to be aligned, (ii) the transformfilter, a vtkTransformPolyData filter for transforming the surface, (iii) the transformation, (iv) the two actors used to display the surfaces and (v) the renderer and the renderwidget. These are defined first, followed by the constructor and the destructor:

```
itcl::class SimpleSurfaceViewer {
    protected variable referencesurface [ vtkPolyData [ newname::vnewobj ] ]
    protected variable transformfilter  [ vtkTransformPolyDataFilter [ newname::vnewobj ] ]
    protected variable targetsurface    [ vtkPolyData [ newname::vnewobj ] ]
    protected variable transformation   0
    protected variable referenceactor    [ vtkActor [  newname::vnewobj ] ]
    protected variable targetactor    [ vtkActor [  newname::vnewobj ] ]
    protected variable renderwidget   0
    protected variable renderer       0
    protected variable basewidget  0
    private   common   thisparam

    constructor { base }  { set basewidget $base
        set thisparam($this,referencemode) "Wireframe"
        set thisparam($this,targetmode) "Wireframe"
        $this Initialize  }
    destructor  {
       $referencesurface Delete; $transformfilter  Delete
       $targetsurface Delete;
       $referenceactor Delete $targetactor Delete
       $renderer Delete; $renderwidget Delete  }
```

The rest of the header defines some interface and implementation methods. The first method, is the method SetSurfacesAndTransformation, which will be used by the main GUI class to provide updates as to the current state of the registration.

```
    public method SetSurfacesAndTransformation { reference target transform }
    public method SetSurfaceMode { md }
    public method ResetRenderer { }
    public method UpdateDisplay { }
```

```
    public method GetThisPointer { } { return $this }
    protected method Initialize { }
    protected method CreatePipelines { }
    protected method AddInteractor { renderwidget renwin   }
};
```

The Implementation: We describe the implementation below, highlighting key points:

```
itcl::body SimpleSurfaceViewer::CreatePipelines { } {
  # Omitted code to generate the two actors and their pipelines
  # Good idea to use a simple object such as a sphere source
  # to initialize the surfaces with default shapes until
  # the user properly specifies them!
}
```

The Initialize method demonstrates how to create and add a `vtkTkRenderWidget`.

```
itcl::body SimpleSurfaceViewer::Initialize { } {
    iwidgets::Labeledframe $basewidget -labelpos nw -labeltext "Surface Viewer"
    pack $basewidget -side top -expand true -fill both
    set labelframe [ $basewidget childsite ]
    set buttonbar [ frame $labelframe.[newname::vnewobj ] ]
    pack $buttonbar -side bottom -expand false -fill x -pady 2

    # Create the Viewer First and add the actors to it
    $this CreatePipelines
    set renderwidget [ vtkTkRenderWidget $labelframe.[newname::vnewobj ] ]
    pack $renderwidget -side top -expand true -fill both
    set renwin [ $renderwidget GetRenderWindow ]
    set renderer [ vtkRenderer [ newname::vnewobj ]]
    $renwin AddRenderer $renderer
    $renderer AddActor $referenceactor;  $renderer AddActor $targetactor

    # Create the Buttons Next
    # Omited code -- see the file for details

    # Key function to properly add interaction -- see below
    $this AddInteractor $renderwidget $renwin
}
```

The interaction is added to the vtkTkRenderWidget using a call to the Tcl function `::vtk::bind_tk_widget` as shown below:

```
itcl::body SimpleSurfaceViewer::AddInteractor { renderwidget renwin } {
```

```
  ::vtk::bind_tk_widget $renderwidget $renwin
  update idletasks
  [ $renwin GetInteractor ] ExposeEvent
}
```

The rest of the world communicates with this viewer primarily using the method below. It can be used to provide the triad of (i) the reference surface, (ii) the target surface and (iii) the current state of the transformation (or 0 to use an identity). Note the use of the ShallowCopy method to properly and efficiently create a link to the input surfaces.

```
itcl::body SimpleSurfaceViewer::SetSurfacesAndTransformation { reference
                                                    target transformation } {
    $referencesurface ShallowCopy $reference;     $targetsurface    ShallowCopy $target
    if { $transformation != 0 } {
       $transformfilter  SetTransform $transformation; $transformfilter Update
    } else {
       set temp  [ vtkIdentityTransform [ newname::vnewobj ] ];
       $transformfilter SetTransform $temp; $temp Delete
    }
}
```

The rest of the code provides additional display functionality:

```
itcl::body SimpleSurfaceViewer::SetSurfaceMode { md } {
  # Omitted code to switch surface display from surface to wireframe or points
}
itcl::body SimpleSurfaceViewer::ResetRenderer { } {
  # Reset the camera to show all objects
    $renderer ResetCamera;  $renderer ResetCameraClippingRange
    $this UpdateDisplay
}
itcl::body SimpleSurfaceViewer::UpdateDisplay { } {
  # Force a render update
    [ $renderer GetRenderWindow ] Render
}
```

The Master Class – PointBasedRegistration

This class leverages the viewer class defined above to provide a complete application around our ICP implementation. We use a notebook widget to break up the user interface into three tabs: (i) The "Input" tab – where we define the input surfaces, (ii) the "Compute" tab – where we specify algorithm parameters and execute the algorithm and (iii) the "Output" tab where we can see and save the output matrix.

The script header contains the usual package require statements. The last two statements require the Surface Viewer and our newly compiled shared library which includes the ICP code. This is where the C++ code is loaded into the Tcl script.

```
package provide PointBasedRegistration 1.0
lappend auto_path [ file dirname [ info script ]]
# Omitted standard stuff
package require SimpleSurfaceViewer
package require loadpointreg
```

The Class Interface: The class interface has the usual mix of methods and variables. We highlight the ExitCommand which is invoked by the exit button. This contains code to do proper cleanup on MS-Windows. The Initialize command creates the user interface one tab at a time:

```
itcl::class PointBasedRegistration {
    protected variable referencesurface [ vtkPolyData [ newname::vnewobj ] ]
    protected variable targetsurface [ vtkPolyData [ newname::vnewobj ] ]
    protected variable currentregistration  0

    protected variable simplesurfaceviewer  0
    protected variable notebook             0
    private   common   thisparam
    protected variable basewidget           0
    protected variable outputbox            0
    protected variable outputmatrix         "1 0 0 0\n 0 1 0 0\n0 0 1 0\n0 0 0 1"

    constructor { base } { set basewidget $base; $this Initialize    }
    destructor  { $referencesurface Delete; $targetsurface Delete
catch [ $currentregistration Delete ]
catch [ itcl::delete obj $simplesurfaceviewer  ]    }

    # Omitted method definitions
    public    method ExitCommand { }
    protected method Initialize { }
};
```

The Initialize method (see below) creates the Graphical User Interface (GUI). This consists of three widgets along the top, with the viewer on the right, a notebook widget on the left and a divider along the middle. Each tab on the notebook is created using a separate helper method. Finally we create the viewer frame.

Creating the GUI:

```
itcl::body PointBasedRegistration::Initialize { } {
    $this ResetParameters
    toplevel $basewidget;    wm geometry $basewidget 600x400
    wm title $basewidget "Point Based Registration";    update

    set notebook  [ iwidgets::tabnotebook $basewidget.left  -tabpos n -width 250 ]
    set middle    [ frame $basewidget.middle -width 5 -bg black ]
    set viewframe [ frame $basewidget.right  -width 500 ]
    pack $notebook $middle -side left -expand false -fill y
```

```
    pack $viewframe -side right -expand true -fill both

    # Create Notebook Tabs
    $this CreateInputGUI    [ $notebook add -label "Input" ]
    $this CreateComputeGUI  [ $notebook add -label "Compute" ]
    $this CreateOutputGUI   [ $notebook add -label "Output" ]
    $notebook view "Input"

    # The Viewer Frame
    frame $viewframe.2;     pack $viewframe.2 -side bottom -expand false -fill x
    eval "button $viewframe.2.2 -text Exit -command { $this ExitCommand }"
    pack $viewframe.2.2 -side right -padx 1 -expand false
    iwidgets::entryfield $viewframe.2.1 -textvariable \
    [ itcl::scope thisparam($this,icp_status) ] -width 50 -labeltext "Status:"
    pack $viewframe.2.1 -side left -padx 1 -expand true -fill x

    set simplesurfaceviewer [ CreateSurfaceViewer $viewframe.1 ]
    eval "wm protocol $basewidget WM_DELETE_WINDOW { $this ExitCommand }"
}
```

The following methods create the individual tab-GUIs. These are shown here in a highly abbreviated form:

```
itcl::body PointBasedRegistration::CreateSurfaceViewer { par } {
    return [ [ SimpleSurfaceViewer \#auto $par ] GetThisPointer ]
}
itcl::body PointBasedRegistration::ResetParameters { } {
  # Omitted Code to set default values for various parameters
}
itcl::body PointBasedRegistration::CreateInputGUI { base } {
  # Omitted Code to create the input GUI for loading the
  # two surfaces
}
itcl::body PointBasedRegistration::CreateComputeGUI { base } {

    eval "button $base.bot -text \"Compute Registration\" \
                -command { $this ComputeRegistration }"
    pack $base.bot -side bottom -expand false -fill x

    iwidgets::Labeledframe $base.top -labelpos nw -labeltext "Parameters" -relief ridge
    pack $base.top -side top -expand true -fill both -pady 2

    set w [ $base.top childsite ]
    set k 0

    iwidgets::entryfield $w.$k -labeltext "Max Num Points:"  -width 5
    -validate integer \
    -textvariable [ itcl::scope thisparam($this,icp_numberofpoints) ] -relief sunken
    pack $w.$k; incr k

    # Omitted code to add more entryfields for the rest of the ICP Parameters
}
```

```
itcl::body PointBasedRegistration::CreateOutputGUI { base } {
  # Omitted code to create the GUI for displaying and saving the matrix
}
```

Finally, the methods for loading and saving objects:

```
itcl::body PointBasedRegistration::LoadSurface { mode inputfilename } {
  # Omitted Code to load the surface from a .vtk file
}
itcl::body PointBasedRegistration::SaveMatrix { } {
  # Omitted Trivial code for saving the matrix
}
```

Invoking and Interacting with vtkLinearICPRegistration: The heart of the PointBasedRegistration class is the
`ComputeRegistration` and `RegistrationProgressUpdate` methods that follow:

```
itcl::body PointBasedRegistration::ComputeRegistration { } {
  # Omitted code that checks that both surfaces are OK
  # Initialize the Display
  $simplesurfaceviewer SetSurfacesAndTransformation $referencesurface $targetsurface 0
  $simplesurfaceviewer UpdateDisplay

  # Create the ICP Registration Class
  set icp [ vtkLinearICPRegistration [ newname::vnewobj ]]
  set currentregistration $icp
  $icp SetSource $referencesurface;  $icp SetTarget $targetsurface
  $icp SetMaximumNumberOfIterations $thisparam($this,icp_numberofiterations)
  $icp SetMaximumNumberOfPoints     $thisparam($this,icp_numberofpoints)
  $icp SetStartByMatchingCentroids  $thisparam($this,icp_alignoriginsfirst)

  if { $thisparam($this,icp_transformationmode) == "Rigid" } {
    $icp SetTransformationTypeToRigid
  } elseif {  $thisparam($this,icp_transformationmode) == "Similarity" } {
    $icp SetTransformationTypeToSimilarity
  } else {
    $icp SetTransformationTypeToAffine
  }
```

Next we attach an observer to the class. This results in the function RegistrationProgressUpdate being called each time the C++ code invokes the update progress method (this generates the ProgressEvent event), or when it is done. Finally the registration is invoked using the Run method:

```
  # Observer Stuff
```

```
    eval "$icp AddObserver ProgressEvent { $this RegistrationProgressUpdate }"
    eval "$icp AddObserver EndEvent      { $this  RegistrationProgressUpdate }"

    # Execute
    $icp Run

    # Omitted code to capture the matrix etc
    # Finally switch to the output pane
    set thisparam($this,icp_status) "Done with Registration"
    $notebook view "Output"
}
```

The Progress Update method first updates the viewer with the current transformation estimate (currentregistration = icp, see above). Next we update the display and the status label on the bottom:

```
itcl::body PointBasedRegistration::RegistrationProgressUpdate { } {
    $simplesurfaceviewer SetSurfacesAndTransformation \
        $referencesurface $targetsurface [ $currentregistration GetTransformation ]
    $simplesurfaceviewer UpdateDisplay
    set thisparam($this,icp_status) [ format "Registration Progress %.2f "\
                    [ expr 100.0 * [ $currentregistration GetProgress ] ] ]
    update idletasks
}
```

The Exit Command: Finally a clean exit command to avoid those annoying crash-on-exit issues in MS-Windows:

```
itcl::body PointBasedRegistration::ExitCommand { } {
    vtkCommand DeleteAllObjects
    exit
}
```

Invoking: An example is shown in script20-2.tcl. This includes loading two sphere surfaces by default:

```
lappend auto_path [ file dirname [ info script ]]
package require PointBasedRegistration
wm withdraw . ; update
set pb [ PointBasedRegistration \#auto .a ]
$pb LoadSurface Reference sphere1.vtk
$pb LoadSurface Target sphere2.vtk
```

20.4 An Alternative Implementation Using BioImage Suite Interactors

Instead of using the standard vtk interactors via the `::vtk::bind_tk_widget` command, we can use our custom BioImage Suite interactors via the use of the `vtkpxGUIRenderer` class. This provides a rich GUI for manipulating the viewer, as shown in Figure 20.2.

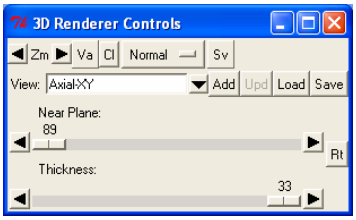


Figure 20.2: The helper GUI of the vtkpxGUIRenderer class.

The use of the BioImage Suite viewers is described in two classes (i) BioImageSuiteSimpleSurfaceViewer – this derives from SimpleSurfaceViewer and replaces some of the functionality there and (ii) BioImageSuitePointBasedRegistration – this similarly derives from PointBasedRegistration and replaces some of the functionality there.

The heart of the new functionality is in three methods in BioImageSuiteSimpleSurfaceViewer. First the new AddInteractor method creates a vtkpxGUIRenderer class (its GUI is accessed by pressing the shift key while clicking with the right mouse button in the viewer.)

```
itcl::body BioImageSuiteSimpleSurfaceViewer::AddInteractor { renderwidget renwin } {
    update idletasks;  set bioimagesuite_viewer  [ vtkpxGUIRenderer [ pxvtable::vnewobj ] ]
    $bioimagesuite_viewer BindMouseEvents $renderwidget \
            "$this HandleMouseEvent" "$this HandleUpdateEvent"
    $bioimagesuite_viewer Initialize $renderwidget $renderer 0
    set renderwindow $renwin
}
```

This requires two additional helper methods for processing expose and mouse events:

```
itcl::body BioImageSuiteSimpleSurfaceViewer::HandleUpdateEvent { args } {
    if {$renderwindow != 0 } { $renderwindow Render    }
}
itcl::body BioImageSuiteSimpleSurfaceViewer::HandleMouseEvent { mouse stat \
                                    x1 x2 widgetname args} {
    if { $bioimagesuite_viewer == 0 } { return    }
    # Need to flip y-axis vtk counts from bottom tk counts from top !!!
    set wd [ $renderwindow GetSize ]
    set x2 [ expr [lindex $wd 1 ] - $x2 ]
    $bioimagesuite_viewer HandleMouseButtonEvent $mouse $stat $x1 $x2
}
```

The BioImageSuitePointBasedRegistration overrides a total of two short methods from its parent class: (i) The CreateSurfaceViewer method which creates the instance of the BioImageSuiteSimpleSurfaceViewer. (ii) A slightly different ExitCommand to handle properly the GUI constructs in BioImage Suite.

```
itcl::body BioImageSuitePointBasedRegistration::CreateSurfaceViewer { par } {
    return [  [ BioImageSuiteSimpleSurfaceViewer \#auto $par ]  GetThisPointer ]
}
itcl::body BioImageSuitePointBasedRegistration::ExitCommand { } {
```

```
    destroy $basewidget ;    update idletasks;    exit
}
```

The use of these classes is illustrated in script20-3.tcl.

Chapter 21

Intensity Based Segmentation

In this chapter, we will discuss the implementation of three common image segmentation, or labeling, algorithms. For the purposes of this chapter, segmentation is the process of assigning a label to each voxel that defines the class (often equivalent to tissue type) it belongs in. For example, in MRI brain images the goal is to classify each voxel as belonging to white matter, gray matter, cerebrospinal fluid (CSF) or background (often the last two are combined into a single class). The three algorithms are (i) a manual multiple threshold segmentation method, (ii) an automated k-means clustering based approach [9] and (iii) an automated method that uses a Markov Random Field model for image smoothness [11, 32]. A BioImage Suite component for accessing these algorithms is also described.

21.1 Introduction – The Three Algorithms

The Multiple Threshold Algorithm

This is a fairly trivial method that may be useful for initialization purposes. Given (a) an input image I which is a collection of voxels $i(\mathbf{x})$, where $\mathbf{x}$ is the voxel index, and (b) a set of M thresholds t_i, where $i = 0 : M-1$ and $t_i > t_{i-1}$ we output a label image L (similarly a collection of voxels $l(\mathbf{x})$ which takes values $0 : M$. The segmentation uses the following simple rule:

$$l(\mathbf{x}) = \begin{cases} 0: & i(\mathbf{x}) \leq t_0 \\ r: & t_{r-1} \leq i(\mathbf{x}) < t_r,\ r \in (1, M-1) \\ M: & i(\mathbf{x}) > t_{M-1} \end{cases}$$

This type of algorithm is often useful for CT images where the intensities are calibrated.

The K-means Clustering Algorithm

The K-means clustering algorithm (see [9]) can be used to essentially estimate the thresholds above. In our implementation we assume that the intensity in each tissue class can be modeled as having a normal distribution with mean μ and standard deviation σ. We will use the notation $p(i|l = c) = \mathcal{N}(\mu_c, \sigma_c)$ to define this, where l is the label and c is the class number. The goal of the version of the k-means algorithm that we will describe is to estimate the class parameters (μ, σ) for all classes and then to assign each voxel to the class that maximizes the function:

$$\begin{aligned}\hat{l}(\mathbf{x}) &= \overset{\arg\max}{l} \quad p(i(\mathbf{x})|l(\mathbf{x}) = c) \\ &= \overset{\arg\max}{l} \quad \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{\frac{-(i(\mathbf{x})-\mu_c)^2}{2\sigma_c^2}}\end{aligned} \tag{21.1}$$

A simpler form of the algorithm assumes that all σ_c's have the same value. This reduces the problem to estimating the means only. The procedure can be described in recipe form as:

1. Decide on the number of classes M
2. Assign class centroids μ_c and optionally standard deviations σ_c for each class. The most common way to do this is to equally space the μ_i's and set all σ_i's to some constant value.
3. Estimate the labels $l(\mathbf{x})$ using equation 21.1. This is an exhaustive optimization – compute $p(i|l = c)$ for all l's and pick the l that maximizes this probability.
4. Estimate a new set of μ_c's and σ_c's by computing the mean and standard deviation of the intensities of the voxels labeled as having class c.
5. Repeat steps 3-4 until the parameters μ_c and σ_c converge.

Note that, since the spatial position of the voxels $\mathbf{x}$ does not enter into the calculation, a quick way to implement this method is by first forming the image histogram and performing all operations on the histogram itself. This can speed up the iterations by a factor of 15000 or so in the case of an $128 \times 128 \times 128$ image whose histogram can be described by 128 bins.

Imposing Local Smoothness using Markov Random Fields

The key weakness of the previous method is that, as noted, the spatial position of each voxel is not used during the segmentation. Spatial homogeneity is often a powerful constraint that can be used to improve the segmentation in the presence of image noise. This can be loosely thought of as finding the label at each voxel that is an optimal constraint between (i) the intensity at that voxel and (ii) the labels assigned to its neighboring voxels.

Markov Random Fields: The probabilistic structure used, most frequently, to capture such homogeneity is to model the label (classification) image as a Markov Random Field. This (and we are skipping a lot of math here) basically reduces to describing the probability of each voxel belonging to class l, as having a Gibbs distribution of the form:

$$p(l(\mathbf{x})) = k_1 \exp(-W(L(\mathcal{R}_\mathbf{x}), l)) \tag{21.2}$$

where k_1 is a normalization constant, $L(\mathcal{R}_\mathbf{x})$ is the set of labels of the neighboring voxels and W is a positive semi-definite function. This is a way of converting an "energy-function" like smoothness term into a probability distribution for integration into a probabilistic framework. The function W can take many forms, the one we will use here is[32]:

$$W(l(\mathbf{X}_n) = \sum_{\mathbf{x}' \in \mathcal{R}_\mathbf{x}} \delta(l(\mathbf{x}') - l(\mathbf{x})) \tag{21.3}$$

This essentially counts the number of voxels in the neighborhood of the voxel at location $\mathbf{x}$ that have labels different from it.

Overall Formulation using the Expectation-Minimization Framework: We first define the vector Θ as the collection of the means and standard deviations of all C classes, i.e. $\Theta = [\mu_0, \sigma_0, \ldots, \mu_{c-1}, \sigma_{c-1}]$. The goal of the segmentation is to estimate both the set of labels L and the class parameters Θ, given the image I. We can express this mathematically as:

$$\hat{L}, \hat{\Theta} = \overset{\arg\max}{L, \Theta} p(L, \Theta|I) \tag{21.4}$$

As is commonly done, this can be solved iteratively in the same spirit as the EM-framework as:

$$\textbf{E-Step:} \quad \Theta^k = \underset{\Theta}{\arg\max}\, p(\Theta|I, L^{k-1}), \quad \textbf{M-Step:} \quad L^k = \underset{L}{\arg\max}\, p(L|I, \Theta^k) \tag{21.5}$$

where k is the iteration number. In the E-Step we estimate a new set of parameters Θ^k given the current classification L^{k-1}. In the M-Step, using the newly estimated Θ^k we estimate a new classification L^k.

E-Step: This is straightforward. For each class i we estimate the mean and standard deviation of the intensities I of all the voxels where $M = i$. This is identical to the procedure used in the k-means algorithm above.

M-Step: This takes the form of a Bayesian a-posterior maximization. First we express

$$\begin{aligned} \hat{l}(\mathbf{x}) \quad = \underset{l}{\arg\max} \quad & \log p(l(\mathbf{x})|i(\mathbf{x}), \Theta^k, L(\mathcal{R}_\mathbf{x}) \\ & k_1 + \underbrace{\log p(i(\mathbf{x}), \Theta^k|l(\mathbf{x}))}_{\text{Data Adherence Term}} + \underbrace{logp(l|L(\mathcal{R}_\mathbf{x}))}_{\text{MRF Smoothness Term}} \end{aligned} \tag{21.6}$$

where k_1 is a constant. This equation is easily maximized by a greedy search strategy as M can only take values of $1 \ldots C$. The prior term on the classification, $p(L)$, can be defined by modeling L as a Markov random field (see discussion above and equation 21.3). We express the likelihood (or data-adherence) term for each possible value of $l(\mathbf{x}) = c$ as:

$$p(i(\mathbf{x}), \Theta^k|l(\mathbf{x}) = c) = p(i(\mathbf{x})|\Theta^k, l(\mathbf{x}) = c) \tag{21.7}$$

which is similar to the model previously used in equation 21.1.

Hopefully the math will become easier to follow with a quick look at the code implementing it.

21.2 Algorithm Implementation

All three algorithms are implemented as classes deriving from the vtkSimpleImageToImageFilter class. The scope is similar to material described in chapter 19, and you are strongly urged to re-read that chapter again.

The Multi Threshold Method

The header file of this method (vtkMultiThresholdSegmentation.h) is fairly standard. In addition to an input image the user needs to specify an array of M thresholds that will results in $M + 1$ output classes.

```
class vtkMultiThresholdSegmentation : public vtkSimpleImageToImageFilter
{
public:
  static vtkMultiThresholdSegmentation *New();
  vtkTypeMacro(vtkMultiThresholdSegmentation,vtkSimpleImageToImageFilter);
  // The Input here is N Thresholds resulting in N+1 output classes
  vtkSetObjectMacro(Thresholds,vtkDoubleArray);
  vtkGetObjectMacro(Thresholds,vtkDoubleArray);

protected:
  vtkMultiThresholdSegmentation();
  virtual ~vtkMultiThresholdSegmentation();
  virtual void ExecuteInformation();
  virtual void SimpleExecute(vtkImageData* in,vtkImageData* out);
```

```
  vtkDoubleArray* Thresholds;
};
```

The implementation (vtkMultiThresholdSegmentation.cpp) is fairly straightforward. First we define the ExecuteInformation method to explicitly specify that the output image will always have type short and a single frame regardless of what the input image is like:

```
void vtkMultiThresholdSegmentation::ExecuteInformation() {
  this->vtkSimpleImageToImageFilter::ExecuteInformation();
  vtkImageData *output=this->GetOutput();
  output->SetScalarTypeToShort();
  output->SetNumberOfScalarComponents(1);
}
```

The thresholding takes place in the SimpleExecute method. This is fairly trivial:

```
void vtkMultiThresholdSegmentation::SimpleExecute(vtkImageData* input,vtkImageData* output )
{
  // Omitted Code to check that inputs are OK
  int num=this->Thresholds->GetNumberOfTuples();
  int NumberOfClasses=num+1;
  double* thr=new double[num];
  double* outthr=new double[num];
  for (int ia=0;ia<num;ia++)
      thr[ia]=this->Thresholds->GetComponent(ia,0);

  // Omitted code that sorts thresholds and places them into array outthr[]
  vtkDataArray* inpdata=input->GetPointData()->GetScalars();
  vtkDataArray* outdata=output->GetPointData()->GetScalars();
  outdata->FillComponent(0,0.0);  int numvoxels=inpdata->GetNumberOfTuples();
  for (int voxel=0;voxel<numvoxels;voxel++)    {
      int c=0,done=0;      double v=inpdata->GetComponent(voxel,0);
      while (c<num && done==0) {
         if (v<outthr[c]) {
            outdata->SetComponent(voxel,0,c);
            done=1;
          }
          else
            c++;
      }
      if (done==0)
         outdata->SetComponent(voxel,0,NumberOfClasses-1);
   }
   delete [] thr;  delete [] outthr;
}
```

The K-Means Method

This is a more interesting class. It takes a number of arguments, namely (i) Number of Class – this specifies the number of unique class labels, (ii) Iterations – the maximum number of iterations, (iii) Number Of Bins – this defines the "resolution" of the histogram, (iv) Convergence – this sets the thereshold that if the maximum change in the the estimation of the class falls below which, the algorithm is set to have converged and (v) MaxSigmaRatio which constrains the range of the values of the standard deviations. Setting this to 1 ensures that all standard deviations are equal. Setting this to 0 results in an unconstrained estimation of the standard deviations.

```
class vtkKMeansSegmentation : public vtkSimpleImageToImageFilter
{
public:
  static vtkKMeansSegmentation *New();
  vtkTypeMacro(vtkKMeansSegmentation,vtkSimpleImageToImageFilter);
  // Omitted Code
  // Set and Get Macros for the five input parameters
  // NumberOfClasses, Iterations, NumberOfBins, Convergence, MaxSigmaRatio

  // Get The Histogram and the class Parameters if desired
  vtkGetObjectMacro(Histogram,vtkImageData);
  vtkGetObjectMacro(Parameters,vtkDoubleArray);
protected:
  vtkKMeansSegmentation();
  virtual ~vtkKMeansSegmentation();
  virtual void ExecuteInformation();
  virtual void SimpleExecute(vtkImageData* in,vtkImageData* out);
  // Helper Methods
  virtual double Metric(double x,double m,double sigma2);
  virtual vtkImageData* CreateHistogram(vtkImageData* input,int NumBins);
  virtual int InitializeParameters(vtkImageData* histogram,int numclasses,
                                   vtkDoubleArray* params);

  // Omitted class parameter definition
};
```

Implementation: This is found in vtkKMeansSegmentation.cpp. We will only highlight key pieces of code here.

The first interesting point is that we define the Gaussian distribution in a virtual method called Metric. If we wanted to replace this with a different distribution, we could derive a new class from vtkKMeansSegmentation and simply override the `Metric` method.

```
double vtkKMeansSegmentation::Metric(double x,double m,double sigma2) {
  return (x-m)*(x-m)/(-2.0*sigma2)* 1.0/sqrt(2.0*vtkMath::Pi()*sigma2);
}
```

The next interesting point is the creation of the histogram. This uses the `vtkImageAccumulate` filter. Note that, the vtkImageAccumulate Filter is designed to generate a histogram for color images (RGB). It assumes that the

input image will have upto three components and produces a histogram for each. We specify the position, spacing and number of bins using the SetComponentOrigin, SetComponentSpacing and SetComponentExtent methods of the vtkImageAccumulate class respectively. Since we are dealing with only grayscale images we set the extent for components 2 and 3 (which do not exist) to have a single bin.

```
vtkImageData* vtkKMeansSegmentation::CreateHistogram(vtkImageData* input,int NumBins) {
if (input==NULL) return NULL; if (NumBins<4) NumBins=4; double range[2];
input->GetPointData()->GetScalars()->GetRange(range); double minv=range[0],
maxv=range[1]; this->HistogramOrigin=minv; this->HistogramSpacing=1.0;
this->NumberOfBins=NumBins; int drange=int(maxv-minv+1); if (drange<
this->NumberOfBins) this->NumberOfBins=drange; while(drange>
this->NumberOfBins*this->HistogramSpacing) this->HistogramSpacing+=1.0;

  this->NumberOfBins=int(drange/this->HistogramSpacing+0.5);
  vtkImageAccumulate* accumulate=vtkImageAccumulate::New();
  accumulate->SetInput(input);
  accumulate->SetComponentOrigin(this->HistogramOrigin,0.0,0.0);
  accumulate->SetComponentSpacing(this->HistogramSpacing,1.0,1.0);
  accumulate->SetComponentExtent(0,this->NumberOfBins-1,0,0,0,0);
  accumulate->Update();

  vtkImageData* out=vtkImageData::New();  out->ShallowCopy(accumulate->GetOutput());
  accumulate->Delete();
  return out;
}
```

The class parameters are initialized in the method `InitializeParameters`. The means are evenly spaced in the intensity range and the standard deviation is set to be constant. The parameters are stored in a 3-component double array which has one tuple for each class. The tuples form the vector [NumVoxels, Mean, Standard Deviation] where NumVoxels is the number of voxels having this class label.

Finally the segmentation itself is implemented in the `SimpleExecute` method as usual. The first part deals with initialization issues:

```
void vtkKMeansSegmentation::SimpleExecute(vtkImageData* input,vtkImageData* output )
{
  // Omitted code: check for valid input image
  this->Histogram=this->CreateHistogram(input,this->NumberOfBins);
  this->Parameters=vtkDoubleArray::New();
  this->InitializeParameters(this->Histogram,this->NumberOfClasses,this->Parameters);
  this->UpdateProgress(0.05);
  int dim[3]; this->Histogram->GetDimensions(dim);
  vtkDataArray* data=this->Histogram->GetPointData()->GetScalars();

  double* mean =new double[this->NumberOfClasses], *sigma2=new double[this->NumberOfClasses];
  double* sum  =new double[this->NumberOfClasses], *sum2 =new double[this->NumberOfClasses];
  double*  num =new double[this->NumberOfClasses],
  for (int j=0;j<this->NumberOfClasses;j++) {
      sigma2[j]=this->Parameters->GetComponent(j,2);      sigma2[j]*=sigma2[j];
```

```
    mean[j] =this->Parameters->GetComponent(j,1);
  }
```

Next we launch into the iterative estimation of labels and class parameters. Note that, rather than iterating on a voxel-by-voxel basis, we operate instead on the image histogram for efficiency.

```
int iter=1;  double error=this->Convergence+1.0;
while (iter <= this->Iterations && error > this->Convergence)
  {
  error=0.0;      double totalnum=0.0;
  for (int i=0;i<this->NumberOfClasses;i++) { sum[i]=0.0;sum2[i]=0.0;num[i]=0.0;}
```

For each bin, we compute the actual intensity v and then find which class has the best "Metric" i.e. the highest likelihood. We assign the 'bin' to this class and use it to form the sums needed to compute the class mean and standard deviation:

```
    for (int bin=0;bin<dim[0];bin++) {
        double v=this->HistogramOrigin+double(bin)*this->HistogramSpacing;
        double numv=data->GetComponent(bin,0);
        double bestp=0.0;  int   bestc=0;
        for (int c=0;c<this->NumberOfClasses;c++) {
            double p=this->Metric(v,mean[c],sigma2[c]);
            if (p>bestp) { bestp=p;  bestc=c;  }
        }
        num[bestc]+=numv;  sum[bestc]+=v*numv;   sum2[bestc]+=v*v*numv;
        totalnum+=numv;
      }
```

Once the sums are complete we compute the class means and standard deviations. We also check for convergence. If the means remain close to their previous values, this implies that the algorithm has converged and we need no more iterations. An interesting twist here (omiited from this code) is that we compute the maximum standard deviation of all the classes and can enforce all other classes to have standard deviations at least as big as a certain fraction (MaxSigmaRatio) of this. This type of constraint adds stability to the estimation:

```
    for (int c=0;c<this->NumberOfClasses;c++)  {
       double m=sum[c]/num[c];
       if (fabs(m-mean[c])>error) error=fabs(m-mean[c]);
       mean[c]=m; sigma2[c]=(sum2[c]/num[c]-mean[c]*mean[c]);
       double s=sqrt(sigma2[c]);
    }
    ++iter;
}
```

Finally, once the iterative estimation of class parameters has converged, we re-estimate class labels, this time on a voxel-by-voxel basis so that we can create the final segmentation map:

```
  // Omitted code sort classes to have ascending means
  vtkDataArray* inpdata=input->GetPointData()->GetScalars();
  vtkDataArray* outdata=output->GetPointData()->GetScalars();
  outdata->FillComponent(0,-1.0);
  int numvoxels=inpdata->GetNumberOfTuples();
  for (int voxel=0;voxel<numvoxels;voxel++) {
      double bestp=0.0; int    bestc=0;
      double v=inpdata->GetComponent(voxel,0);
      for (int c=0;c<this->NumberOfClasses;c++) {
          double p=this->Metric(v,mean[c],sigma2[c]);
          if (p>bestp || c==0)  {  bestp=p;  bestc=c;  }
        }
      outdata->SetComponent(voxel,0,bestc);
     }
   delete [] mean;  delete [] sigma2;  delete [] sum;  delete [] sum2;  delete [] num;
}
```

The MRF Segmentation Method

Things will begin to appear to get more complex here. There is, however, nothing particularly complex about any of the code following, but it might be a little longer than can be easily described in a handout.

The class definition (vtkMRFSegmentation.h) is fairly straight-forward. The class takes a number of key parameters namely (i) NumberOfIterations, (ii) NumberOfClasses (iii) Smoothness – the weight of the MRF term. Secondary parameters include ConvergencePercentage and MRFIterations (the number of iterations in the M-Step). Finally, we also use an ImageNoise term that adds some stability to the estimation – more on this later.

```
class vtkMRFSegmentation : public vtkSimpleImageToImageFilter {
public:
  static vtkMRFSegmentation *New();
  vtkTypeMacro(vtkMRFSegmentation,vtkSimpleImageToImageFilter);
  // Omitted Code, Set/Get Macros for Class Parameters such as
  // NumberOfIterations, MRFIterations, NumberOfClasses, Smoothness,ImageNoisePercentage
  // ConvergencePercentage

  // Initial Segmentation Map as Image
  vtkSetObjectMacro(InitialSegmentation,vtkImageData);
  vtkGetObjectMacro(InitialSegmentation,vtkImageData);
  // Initial Segmentation Map as Image
  vtkGetObjectMacro(Parameters,vtkDoubleArray);
protected:
  vtkMRFSegmentation();
  virtual ~vtkMRFSegmentation();
  virtual void ExecuteInformation();
  virtual void SimpleExecute(vtkImageData* in,vtkImageData* out);
```

```
  // E-Step Is Easy for the most_part
  virtual void DoExpectationStep(vtkImageData* intensities,vtkImageData*  classification,
                                 vtkDoubleArray* params,int numclasses);

  // M-Step (...) indicates lots of parameters!
  virtual double Metric(double x,double m,double sigma2);
  virtual double ComputeLogLikelihoodProbability(...)
  virtual double ComputeLogMRFPrior(...)
  virtual double ComputeTotalMinusLogProbability(...)
  virtual int  UpdateVoxel(int eveni,int evenj,int evenk,int pass);
  virtual int  ComputeMRFIncrementsAndWeights(vtkImageData* img,int incr[6],double wgt[6]);
  virtual int  ClassifyVoxel(....)
  virtual double DoMaximizationStep(vtkImageData* intensity_image, vtkImageData* label_image,
                                    vtkDoubleArray* params, int numclasses,
                                    double smoothness, int maxiter);
  //Omitted Code -- definition of class variables
};
```

The Implemetation: This consists of about 400 lines of code in total which we will not describe in huge detail. Essentially the flow goes as follows: SimpleExecute initializes the process and calls the Expectation and Maximization procedures in an alternating fashion until convergence:

```
void vtkMRFSegmentation::SimpleExecute(vtkImageData* input,vtkImageData* output)
{
  # Omitted Code to check that inputs are valid
  # Compute Image Noise Variance
  double range[2]; input->GetPointData()->GetScalars()->GetRange(range);
  this->ImageNoiseVariance=pow(range[0]+
                               this->ImageNoisePercentage*0.01*(range[1]-range[0]),2.0);

  # Create Ouput Image
  vtkImageData* classification=vtkImageData::New();
  classification->CopyStructure(output);
  classification->AllocateScalars();
  classification->GetPointData()->GetScalars()->CopyComponent(0,
                       this->InitialSegmentation->GetPointData()->GetScalars(),0);

  int iterations=1;
  while(iterations<=this->NumberOfIterations)
    {
      this->DoExpectationStep(input,classification,this->Parameters,this->NumberOfClasses);
      double changed=this->DoMaximizationStep(input,classification,this->Parameters,
 this->NumberOfClasses,this->Smoothness,this->MRFIterations);
      if (changed<this->ConvergencePercentage)
iterations=this->NumberOfIterations+1;
      ++iterations;
    }
  output->ShallowCopy(classification);  classification->Delete();
  this->UpdateProgress(1.0);
}
```

The Expectation Step: The expectation step is also fairly straight-forward and similar in scope to the code in vtkKMeansSegmentation for estimating parameters.

```
void vtkMRFSegmentation::DoExpectationStep(vtkImageData* intensities,
                                           vtkImageData* classification,
                                           vtkDoubleArray* params,int numclasses)
{
  vtkDataArray* intens=intensities->GetPointData()->GetScalars();
  vtkDataArray* labels=classification->GetPointData()->GetScalars();
  double* sum  =new double[numclasses],*sum2 =new double[numclasses];
  int*    count=new int[numclasses];
  for (int ia=0;ia<numclasses;ia++) {
       sum[ia]=0.0; sum2[ia]=0.0; count[ia]=0;  }
  int numvoxels=intens->GetNumberOfTuples();
  for (int i=0;i<numvoxels;i++) {
      double v=intens->GetComponent(i,0);
      int    l=(int)labels->GetComponent(i,0);
      if (l>=0 && l < numclasses) {
         sum[l]+=v; sum2[l]+=v*v; count[l]+=1; }
      }

  double totalnp=double(numvoxels);
  for (int i=0;i<numclasses;i++) {
      double numv=count[i];       params->SetComponent(i,0,numv);
      double mean=sum[i]/numv;    params->SetComponent(i,1,mean);
      double mean2=sum2[i]/numv;  params->SetComponent(i,2,sqrt(mean2-mean*mean));
    }
}
```

The Maximization Step: The M-Step is a little bit more involved. We first note that we are using first-order neighborhoods for the MRF, i.e. the voxels immediately adjacent in the x,y and z-directions. Hence we can update the labels in two passes by thinking of the image as a chess-board. We first update all the black squares (keeping the labels of the white squares fixed) and then update all the white squares (keeping the black squares fixed). In addition, we randomize the order in which we start.

This method (DoMaximizationStep) can be simply thought of as a smart way to call the ClassifyVoxel method (described next) in the appropriate order.

```
double vtkMRFSegmentation::DoMaximizationStep(vtkImageData* intensity_image,
                                              vtkImageData* label_image,
                                              vtkDoubleArray* params,  int  numclasses,
                                              double smoothness,int maxiter)
{
  vtkDataArray* intensities=intensity_image->GetPointData()->GetScalars();
  vtkDataArray* labels=label_image->GetPointData()->GetScalars();
  int nt=intensities->GetNumberOfTuples();
  int dim[3]; intensity_image->GetDimensions(dim);
  int incr[6]; double weights[6];
  this->ComputeMRFIncrementsAndWeights(intensity_image,incr,weights);
```

```
  int done=0,iter=0;  int tenth=nt/11;

  double sumchanged=0.0;
  while (done==0 && iter<maxiter) {
      double total=0.0, changed=0.0;
      int order=(vtkMath::Random()>0.5);
      for (int pass=0;pass<=1;pass++) {
          int realpass=pass;
          if (order==1) realpass=1-pass;

  // Omitted Code, loop over k (z-axis), j (y-axis)
  // Compute voxel index
  int vox=k*incr[5]+j*incr[3]+1;
          for (int i=1;i<dim[0]-1;i++) {
  // Check for voxel color (i.e. black square or white square)
  // If yes update
          if ( this->UpdateVoxel(eveni,evenj,evenk,pass)==1) {
              changed+=this->ClassifyVoxel(vox,intensities,labels,params,
                                           numclasses,incr,weights,smoothness);
               ++total;
         }
      ++vox;
      ....
      }
      changed=100.0*changed/total;      sumchanged+=changed;
      if (changed<this->ConvergencePercentage) done=1;
      ++iter;
    }
  return sumchanged;
}
```

Computing the Posterior: The hard work of the M-Step is done by the `ClassifyVoxel` method. This takes one voxel, specified by `voxel_index` and computes it's posterior probability (equation 21.6). for each possible class assignment. Then, we set the class label for this voxel to be the one that corresponds to the class that maximimizes the posterior probability:

```
int  vtkMRFSegmentation::ClassifyVoxel(int voxel_index,vtkDataArray* intensities,
                                vtkDataArray* labels,vtkDoubleArray* params,
                                      int numclasses,int incr[6],double wgth[6],
                                      double smoothness)
{
  int current_label=(int)labels->GetComponent(voxel_index,0);
  int    bestclass=0;
  double bestprob=0.0;
  double v=intensities->GetComponent(voxel_index,0);
  for (int cl=0;cl< numclasses;cl++) {
      double prob=this->ComputeTotalMinusLogProbability(v,voxel_index,cl,labels,
                                                         params,numclasses,incr,
                                                         wgth,smoothness);
      if (cl==0 || prob<bestprob) {
          bestprob=prob; bestclass=cl;
```

```
      }
    }
  if (bestclass!=current_label) {
      labels->SetComponent(voxel_index,0,bestclass);  return 1;
    }
  return 0;
}
```

The probability is computed by the `ComputeTotalMinusLogProbability` method. This is also straightforward as it delegates all the work to two other functions (i) ComputeLogLikelihoodProability – the data term and (ii) ComputeLogMRFPrior – the smoothness term. We examine these, in order, next:

```
double vtkMRFSegmentation::ComputeTotalMinusLogProbability(double intensity,
                int current_voxel, int current_label,
                vtkDataArray* labels,vtkDoubleArray* parameters,
                int numclasses, int incr[6],double wgth[6],double smoothness)
{
  double mlterm=this->ComputeLogLikelihoodProbability(intensity,current_label,parameters);
  double pmrf  =smoothness*this->ComputeLogMRFPrior(labels,current_voxel,
                                 current_label,numclasses,incr,wgth);
  return -mlterm+pmrf;
}
```

Computing the Log Likelihood: The log likelihood probability has an interesting twist. We model each voxel as having intensity $y = x + n$, where x is the true intensity and n is the image noise (zero mean, gaussian). The distribution of x depends on its current label l and has mean μ_l and standard deviation σ_l. The distribution of y has mean μ_l and variance $\sigma_l^2 + \sigma_n^2$ where σ_n is the standard deviation of the noise. This added term is useful because it adds stability to the estimation, i.e. all classes will have standard deviation at least equal to $sigma_n$. Finally, as before, the distribution is implemented in a separate function, Metric, so that it can easily be changed.

```
double vtkMRFSegmentation::ComputeLogLikelihoodProbability(double intensity,
                                int current_label,vtkDoubleArray* params)
{
  double totalprob=0.0;
  double mean=params->GetComponent(current_label,1);
  double sigma=params->GetComponent(current_label,2);
  double variance=sigma*sigma+this->ImageNoiseVariance;
  totalprob=log(this->Metric(intensity,mean,variance));
  return totalprob;
}
```

Computing the MRF Term: This implements the MRF model desribed in equation 21.3. The only added twist is that we weight the effect of each voxel based on its distance from the current voxel. (If the images are isotropic all weights will take value=1).

```
double vtkMRFSegmentation::ComputeLogMRFPrior(vtkDataArray* labels,int vox,
                                              int current_label,int numclasses,
                                              int incr[6],double wgth[6])
{
  double sum=0.0;
  for (int i=0;i<=5;i++) {
        int l=(int)labels->GetComponent(vox+incr[i],0);
          if (l!=current_label)  sum+=wgth[i];
}
    }
  return sum;
}
```

Other Functions: The Metric function implements the Gaussian distribution. The UpdateVoxel method determines whether a voxel is on the right color of the checkerboard and hence whether it should be updated (this is called from within the DoMaximizationStep method). Finally the ComputeMRFIncrementsAndWeights method computes both the offsets (in raster-voxel order) and the weights of neighboring voxels for use in the MRF model computation.

Compiling using CMAKE

These classes are placed in a library called MRFSegm. The CMakeLists.txt file is straight-forward and will not be described here. The libraries can be loaded into Tcl using the script loadmrfsegm.tcl shown below:

```
package provide loadmrfsegm 1.0
if { $tcl_platform(platform) == "windows" } {
    set name debug/vtkMRFSegmTCL.dll
} else {
    set name libvtkMRFSegmTCL.so
}
load $name;unset name
```

21.3 A Complete Intensity-Based Segmentation Application

We implement this as a BioImage Suite application, shown in Figure 21.1, following the guidelines in chapter 16. The implementation consists of two files, the main script (segmtool.tcl) and the segmentation control class (mrfutility.tcl). We examine these in turn next.

The Main Application

This is defined in the file `segmtool.tcl` and is similar to the mytool.tcl script described in Chapter 16. There is nothing exciting here, other than the fact that we will use the Objectmap viewer so that we can overlay the results of the segmentation on the original image, if we choose to.

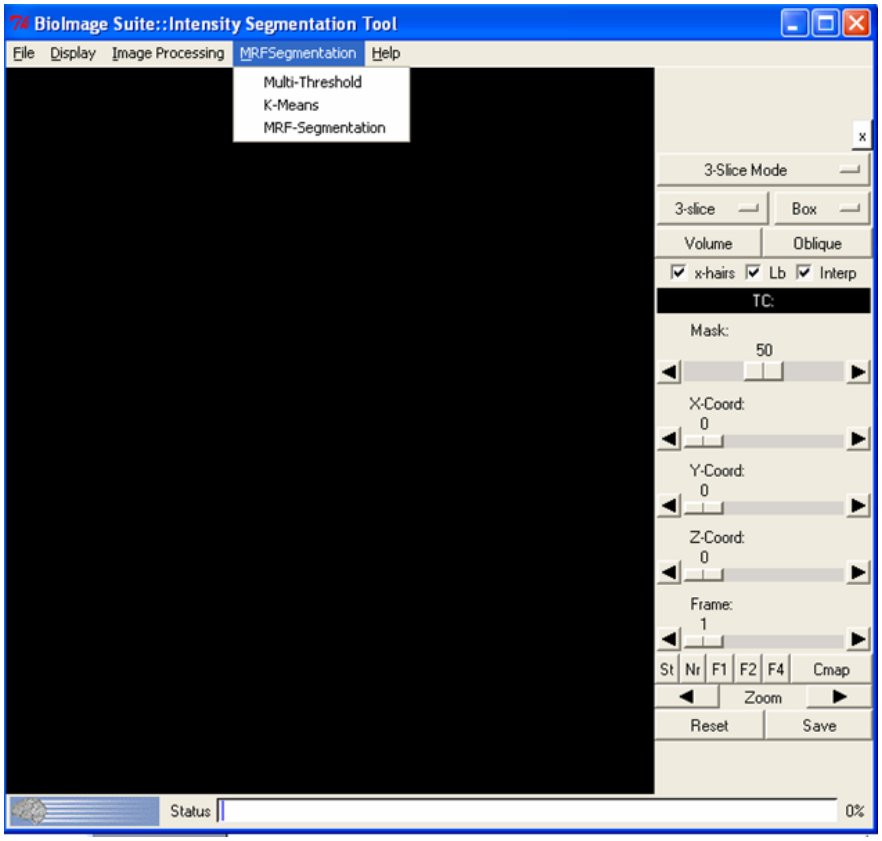


Figure 21.1: The Main application with the added MRFSegmentation menu.

```
lappend auto_path [ file dirname [ info script ]]
package require loadbioimagesuite 1.0
package require pxitclbaseimageviewer 1.0
package require mrfutility 1.0

# Eliminate the default tk window
wm withdraw .

# Initialize a base application with some default settings
# See bioimagesuite/main/pxitclbaseimageviewer.tcl for  all the options
set baseviewer [ pxitclbaseimageviewer \#auto 0 ]
$baseviewer configure -appname "BioImage Suite::Intensity Segmentation Tool"

# Omitted code .. define default choices of controls
$baseviewer configure -show_standard_images 1
....

# Create an Objectmap  Viewer
$baseviewer InitializeObjectmapViewer .[ pxvtable::vnewobj ] 1

# Add a submenu for our Segmentation Control
set menubase [ $baseviewer cget -menubase ]
set mb [ menu $menubase.[pxvtable::vnewobj] -tearoff 0  ]
$menubase add cascade  -label "MRFSegmentation" -menu $mb -underline 0
```

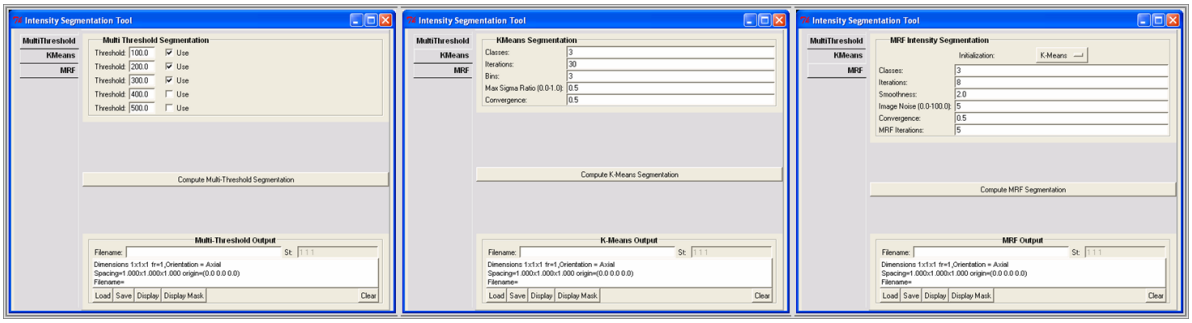


Figure 21.2: The three tabs of the Segmentation Control.

```
# Create the segmentation control, add it to the menu and register it with
# the viewer (addcontrol)
set myutil [ mrfutility \#auto $baseviewer ]
$myutil Initialize [ $baseviewer GetBaseWidget ].[ pxvtable::vnewobj ]
$myutil AddToMenuButton $mb
$baseviewer AddControl $myutil

# Omitted code -- the rest is similar to mytool.tcl
```

The Segmentation Control

The Segmentation Control is implemented as an [Incr] Tcl class deriving from pxitclbaseimagecontrol. This consists of three tabs each having controls for setting the parameters for one of the three algorithms described earlier. The three tabs are shown in Figure 21.2

In the class header there is a statement:

`package require loadmrfsegm`

for loading our newly implemented methods into the Tcl interpreter.

The class definition is as follows. It consists of essentially five types of methods, (i) the constructor/destructor pair, (ii) initialization methods (iii) methods for generating the GUI for the individual methods called by the initialization methods, (iii) a method to add this control the menu and (iv) methods to perform the segmentation and deal with the results.

```
itcl::class mrfutility {
    inherit pxitclbaseimagecontrol
    protected common thisparam
    constructor { par } { pxitclbaseimagecontrol::constructor $par } { InitializeControl }
    destructor { }

    # initialization methods
    public method InitializeControl { }
    public method Initialize { inpwidg }

    # interface creation methods
    protected method CreateOutputImageGUIControl { guiname name widget }
```

```
    protected method CreateMultiThresholdControl { name }
    protected method CreateKMeansControl { name }
    protected method CreateMRFControl    { name }

    # Add this control to a Menu Button
    public method AddToMenuButton { mb args }

    # Computational Utility Stuff
    public method CheckImage {  image name operation verbosity }
    public method ComputeMultiThresholdSegmentation { }
    public method ComputeKMeansSegmentation { }
    public method ComputeMRFSegmentation { }

    # Deal with Results
    public method ProcessResult { image guiname opname }
}
```

Initialization Code: Nothing special here. The Initialize function creates a toplevel widget, packs a notebook in it and then calls three helper methods to add the controls for each tab.

```
itcl::body mrfutility::InitializeControl { } {
  # Omitted Code which sets some default parameters
}

itcl::body mrfutility::Initialize { widget } {
    if { $initialized == 1 } { return $basewidget }
    set basewidget [toplevel $widget ];    wm geometry $basewidget 610x450
    set notebook $basewidget.notebook;   iwidgets::tabnotebook $notebook -tabpos w

    CreateMultiThresholdControl    [ $notebook add -label "MultiThreshold" ]
    CreateKMeansControl            [ $notebook add -label "KMeans" ]
    CreateMRFControl               [ $notebook add -label "MRF" ]
    pack $notebook -side top -fill both -expand t -padx 5

    set initialized 1; SetTitle "Intensity Segmentation Tool"
    eval "wm protocol $basewidget WM_DELETE_WINDOW { wm withdraw $basewidget }"
    return $basewidget
}
```

Creating the GUI for each Method: Three very similar methods (CreateMultiThresholdControl, CreateK-MeansControl and CreateMRFControl) create the interface for each algorithm. We will look at one of them in detail – CreateKMeansControl, the others are very similar.

This control consists of three parts. At the top there is a series of `iwidgets::entryfield` widgets for setting the values of the input parameters. In the middle there is a button (base.but) for executing the segmentation, and at the bottom there is a pxitclimageGUI control for storing the output segmentation.

```
itcl::body mrfutility::CreateKMeansControl { base } {
```

```
    iwidgets::labeledframe $base.frame0 -labelpos nw  \
             -labeltext "KMeans Segmentation"
    pack $base.frame0 -fill both -expand f -pady 5

    set w [ $base.frame0 childsite ]; set k 0
    iwidgets::entryfield  $w.$k -labeltext "Classes:" \
              -textvariable [ itcl::scope  thisparam($this,kmeans_numclasses) ] \
              -width 2 -validate integer
    pack $w.$k -side top -expand true -fill x; incr k

    # Omitted code for entryfileds for Iterations, Bins etc.

    # Create a compute button
    eval "button $base.but -text \"Compute K-Means Segmentation\" \
          -command { $this ComputeKMeansSegmentation }"
    pack $base.but -side top -expand t -fill x

    # Create a pxitclimageGUI to store output -- more later
    set widg [ $this CreateOutputImageGUIControl "kmeansoutput" "K-Means Output" $base.bot ]
    pack $widg -side bottom -expand f -fill x
}
```

The Image GUIs are created by the CreateOutputImageGUIControl Function below. We add two functions, one to display the segmentation as an image (Display) and one to display it as an overlay (Display Mask).

```
itcl::body mrfutility::CreateOutputImageGUIControl { guiname name widget } {
    set  thisparam($this,$guiname) [ [ pxitclimageGUI \#auto  ] GetThisPointer ]
    $thisparam($this,$guiname) configure -description $name
    $thisparam($this,$guiname) Initialize $widget
    set bbut [ $thisparam($this,$guiname) cget -browsebutton ]; pack forget $bbut
    $thisparam($this,$guiname) AddFunction "$parent SetResultsFromObject" "Display" "$this"
    $thisparam($this,$guiname) AddFunction "$parent SetMaskFromObject" "Display Mask" "$this"
    return $widget
}
```

Computational Code: The three algorithms are executed using the methods ComputeMultiThresholdSegmentation, ComputeKMeansSegmentation and method ComputeMRFSegmentation respectively. As before, we will look at one of these in detail, ComputeKMeansSegmentation. We first check that the current image of the viewer (stored in the currentimage variable) exists. If it does, we create the vtkKMeansSegmentation object and set it's parameters. The segmentation is invoked using the Update method. The result is then handled using the ProcessResult method. Nothing complicated here:

```
itcl::body mrfutility::ComputeKMeansSegmentation {  } {
    # Omitted code to check that currentimage exists
    WatchOn
    set segm [ vtkKMeansSegmentation [ pxvtable::vnewobj ]]
```

```
    $segm SetInput [ $currentimage GetImage ]
    $segm SetNumberOfBins    $thisparam($this,kmeans_numbins)
    $segm SetNumberOfClasses $thisparam($this,kmeans_numclasses)
    # ...
    $segm Update
    WatchOff
    $this ProcessResult [ $segm GetOutput ] "kmeansoutput" "km"
    $segm Delete
}
```

The fancy coding is in the ProcessResult method. This does three things: (i) it copies the segmentation result into the image currentresults, (ii) it sends the segmentation result to the viewers mask image (the overlay) and (iii) it stores it into the appropriated pxitclimageGUI for later use.

```
itcl::body mrfutility::ProcessResult { image guiname opname } {
  # Step 1
    $currentresults ShallowCopyImage  $image
    $currentresults configure -filename  [ AddPrefix [ $currentimage cget
      -filename ] $opname ]
  # Step 2
    $parent SetMaskFromObject $currentresults $this
  # Step 3
    set gui $thisparam($this,$guiname)
    [ $gui GetObject ]  ShallowCopy $currentresults;   $gui Update
}
```

Chapter 22

A Templated Image to Image Filter

In this Chapter we describe the implementation of multi-threaded templated image-to-image filter implementation. Such implementations, while somewhat more complex, can offer substantial speed improvements over the more simple filters previously described.

22.1 Introduction

In previous Chapters, we have considered simple image-to-image filters deriving from the class vtkSimpleImageToImageFilter. A common aspect of all the filters we discussed was the reliance on convenience methods such as `GetComponent`, `SetComponent`, `SetScalarComponentFromDouble` and `GetScalarComponentAsDouble` for manipulating images. These convenience methods hide the actual underlying data type of the image. They, at least to the programmer, treat images the same regardless of whether the image has type short or float.[1]

This convenience often comes at the cost of computational overhead. Ultimately the fastest way to access data is by direct pointer manipulation. This makes the programming arcane and practically returns the programmer to the age of assembly language. However, such operations can dramatically improve the computational speed of many operations. The usual path is to implement a class using the convenience methods and if it becomes too critical in a large system, re-implement it carefully, once and for all, with lower level operations.

Modern C++ is beginning to provide an additional mechanism to encapsulate pointer operations with less overhead, using a type of class called iterators. Many iterators are defined in the Standard Template Library (STL), which unfortunately we will not have time to get into this semester. However, VTK defines two simple iterators that we will explore in the context of the example to follow.

Iterators leverage templating for execution speed. An iterator is programmed in a 'generic' (not type specific) setup, unlike the Get/Set methods above which take double inputs. Templated implementations are used to generate specific versions of the iterators for specific types such as floats, ints etc.

In this Chapter, we will explore this topics by describing a VTK image-to-image filter called vtkImageShiftScale. This essentially takes an image I as input and outputs an image J of the form $J = (I + a) * b$, where a is the shift and b is called the scale.

[1] Naturally in the Set methods if the value specified is not valid for the type, we have unexpected results. For example, consider setting the value of a voxel in a short image to 10.2. This will most likely be truncated to 10 as short images do not handle decimal points. More interesting artifacts occur when the value is outside the range of the type. For example, consider the setting the value of a voxel in an 1-byte image to 300.

22.2 The Header – vtkImageShiftScale.h

This filter derives from `vtkImageToImageFilter` is the parent class of all efficient image-to-image filters in VTK. The header file is fairly straight-forward:

```
class vtkImageShiftScale : public vtkImageToImageFilter {
public:
  static vtkImageShiftScale *New();
  vtkTypeRevisionMacro(vtkImageShiftScale,vtkImageToImageFilter);
  void PrintSelf(ostream& os, vtkIndent indent);
  // Set/Get the shift value.
  vtkSetMacro(Shift,double);
  vtkGetMacro(Shift,double);
  // Set/Get the scale value.
  vtkSetMacro(Scale,double);
  vtkGetMacro(Scale,double);
  // Set the desired output scalar type. The result of the shift
  // and scale operations is cast to the type specified.
  vtkSetMacro(OutputScalarType, int);
  vtkGetMacro(OutputScalarType, int);
  void SetOutputScalarTypeToDouble();
  // Ommitted Code ...
  // When the ClampOverflow flag is on, the data is thresholded so that
  // the output value does not exceed the max or min of the data type.
  vtkSetMacro(ClampOverflow, int);
  vtkGetMacro(ClampOverflow, int);
  vtkBooleanMacro(ClampOverflow, int);

protected:
  vtkImageShiftScale();
  ~vtkImageShiftScale() {};
  double Shift, Scale;
  int OutputScalarType, ClampOverflow;
  void ExecuteInformation(vtkImageData *inData, vtkImageData *outData);
  void ThreadedExecute(vtkImageData *inData, vtkImageData *outData,int extent[6], int id);
};
```

22.3 The Implementation – vtkImageShiftScale.cxx

The implementation is more interesting. First the ExecuteInformation method. This is fairly straightforward. If the Output type is not the same as the input type (e.g. input is short, but output is float) we need to notify the pipeline.

```
void vtkImageShiftScale::ExecuteInformation(vtkImageData *inData,vtkImageData *outData) {
  this->vtkImageToImageFilter::ExecuteInformation( inData, outData );
  if (this->OutputScalarType != -1)
    outData->SetScalarType(this->OutputScalarType);
```

```
}
```

Templated Execution

Unlike ITK, which we will discuss in the Chapter 24, VTK does not use templates in the class interface. This results in the VTK filter classes not being templated in themselves. However, VTK does use templates in the implementation of some of the code for efficiency. Since, however, the classes themselves are not templated, such filters cannot have templated member functions. The solution to this quandary is the use of ordinary, specially named templated procedures, inserted in the VTK source which are called from within the member functions.

These procedures are often specially named to reduce the likelihood of naming conflicts at link time. (Naming conflicts occur when more than one function/procedure has the same exact name and argument list). In general their names begin with the class name of the class whose methods will call them. This is best illustrated by means of the following example.

In vtkImageShiftScale, the input point to the filter is the `ThreadedExecute` method (which is the rough equivalent of SimpleExecute in the classes deriving from vtkSimpleImageToImageFilter).

This then calls an ordinary procedure called `vtkImageShiftScaleExecute1` which is templated by the type of the input image. This in turn calls a second procedure `vtkImageShiftScaleExecute` which is templated over both the type of the input image *and* the type of the output image. The reason for this double hop, is that we use (implicitly) switch statements to check for data type. Doing this in a single step, would require N^2 cases (where N is the number of possible data types) whereas doing this in two steps reduces the coding complexity to N followed by N more ($= 2N$ total cases).

The key to understanding the code below is to understand the operation of the *vtkTemplateMacros*.

Step 1: ThreadedExecute: This is the "main" method of the filter. The method takes four inputs: (i) inData – the input, (ii) outData, the output, (iii) outExt[6] – which specifies the part of the image to update and (iv) id – this is the thread id.

All vtkImageToImageFilters are multi-threaded. This means that the execution of the filter can be split over a number of processors, if these are available. To accomplish this, the output is split into pieces, and each thread is responsible for computing the result for its piece. For example, in the case of this filter, vtkImageShiftScale, thread 1 may be computing the output for the top half of the image and thread 2 for the bottom half. (Naturally not all operations can be multi-threaded.). `outExt[6]` defines the region over which this thread is responsible. It is a six-component array of the form $(xmin, xmax, ymin, ymax, zmin, zmax)$.

```
void vtkImageShiftScale::ThreadedExecute(vtkImageData *inData,vtkImageData *outData,
                                         int outExt[6], int id) {
  switch (inData->GetScalarType())   {
    vtkTemplateMacro6(vtkImageShiftScaleExecute1, this,
                      inData, outData, outExt, id, static_cast<VTK_TT *>(0));
    default:
      vtkErrorMacro(<< "Execute: Unknown ScalarType");
      return;
    }
}
```

The messy part of this function is the use of the vtkTemplateMacro6. Expanding this macro would result in code of the form:

```
  switch (inData->GetScalarType())  {
    // BEGIN MACRO
    case double:
      vtkImageShiftScaleExecute1(this,inData,OutData,outExt,id,static_cast<double *>0);
      break;
    case float:
      vtkImageShiftScaleExecute1(this,inData,OutData,outExt,id,static_cast<float *>0);
      break;
     // OMMITTED CODE
     // similarly for long, unsigned long, int, unsigned int, short,
     //           unsigned short, char and unsigned char
     // END MACRO
    default:
      vtkErrorMacro(<< "Execute: Unknown ScalarType");
      return;
    }
```

We use the macro vtkTemplateMacro6 because the function that is being called by the macro – vtkImageShiftScale-Execute1 – takes six arguments. There are also other versions for 3 to 10 arguments (e.g. vtkTemplateMacro10!)

Also note, that since `vtkImageShiftScaleExecute1` is not a member function of vtkImageShiftScale, we pass the `this` pointer as an explicit argument, so that this function can access class data members for more information – otherwise they would all have to be passed as parameters.

Step 2: vtkImageShiftScaleExecute1

Please note that this function is not a member of vtkImageShiftScale. It is an ordinary c-like function which simply happens to be in the same file as the implementation of vtkImageShiftScale. This enable its use without necessary needing to declare it in any header file – this follows the solid principles of PIMPL or private implementation!

```
template <class T>
void vtkImageShiftScaleExecute1(vtkImageShiftScale *self,
                                vtkImageData *inData,
                                vtkImageData *outData,
                                int outExt[6], int id, T *)
{
  switch (outData->GetScalarType())
    {
    vtkTemplateMacro7(vtkImageShiftScaleExecute, self, inData,
                      outData,outExt, id,
                      static_cast<T *>(0), static_cast<VTK_TT *>(0));
    default:
      vtkGenericWarningMacro("Execute: Unknown input ScalarType");
      return;
    }
}
```

This function is templated using type T, which is the type of the input image (e.g. short, float etc.). It then uses the vtkTemplateMacro7 macro – which is similar to the vtkTemplateMacro6 described before (see vtkGetSet.h for more details on these macro definitions, in the VTK include directory). This macro creates another set of case

statements where this time the conditional is on the output type.

This switch method ends up calling the final function vtkImageShiftScaleExecute, where the actual work will take place.

Step 3: vtkImageShiftScaleExecute

vtkImageShiftScaleExecute is were the seemingly trivial work of addition and multiplication takes place. This is a double-templated function over input type `IT` and output type `OT`. It's first argument is a pointer to the instance of the class from which it was called, whereas the rest are information about the operation. The last two arguments are zeros and are unused, the important aspect of these last two arguments is that they explicitly define the input and output types for the templating.

The code follows – I have rearranged it marginally for greater clarity. First we get the key class parameters from the calling class, using the self pointer:

```
template <class IT, class OT>
void vtkImageShiftScaleExecute(vtkImageShiftScale *self,
                               vtkImageData *inData,
                               vtkImageData *outData,
                               int outExt[6], int id,  IT *, OT *)
{

  double shift = self->GetShift();
  double scale = self->GetScale();
  int clamp = self->GetClampOverflow();
```

Next we get the range of types for the current type. These would be 0 and 255 in the case of unsigned char, 0 to 65535 for short etc.

```
  // for preventing overflow
  doubletypeMin = outData->GetScalarTypeMin();
  double typeMax = outData->GetScalarTypeMax();
```

Next we create two iterators. Iterators are special classes (originally defined in the C++ Standard Template Library) that allow the "easy" looping through data structures of arbitrary complexity. Iterators are the modern way of direct "pointer manipulation".

In this case, the iterators are not pointers but are simply allocated on the stack (i.e. like ordinary variables). The first iterator (inIT) is simply used to traverse through the input image, whereas the second iterator (outIt) does the same job for the second image. The second iterator is of type vtkImageProgressIterator and in addition to looping over the image, periodically calls the `self->UpdateProgress()` method to keep the user informed of the process of the filter.

The iterator breaks the image region (defined by outExt) over which it will iterate into a number of continuous (in memory) parts called spans. For example if we have as input a $16 \times 16 \times 16$ image and outExt=$[0, 7, 0, 7, 0, 7]$, the first span would be from voxel $(0, 0, 0)$ to $(7, 0, 0)$. At this point, we will need to jump to $(0, 1, 0)$ and begin a second span.

```
  vtkImageIterator<IT> inIt(inData, outExt);
  vtkImageProgressIterator<OT> outIt(outData, outExt, self, id);


  // Loop through ouput pixels until end
  while (!outIt.IsAtEnd())
    {
      // Get start and end of contiguous piece or span
      IT* inSI = inIt.BeginSpan();
      OT* outSI = outIt.BeginSpan();
      OT* outSIEnd = outIt.EndSpan();

      // If not at the end
      while (outSI != outSIEnd) {
      // Pixel operation
      val = ((double)(*inSI) + shift) * scale;
      if (clamp){
         if (val > typeMax)
          val = typeMax;
         if (val < typeMin)
          val = typeMin;
      }
      // Set the Output
      *outSI = (OT)(val);
      // Increment the iterators -- similar to pointer incrementation
      ++outSI;
      ++inSI;
    }
      // Done with this span, onto the next one
    inIt.NextSpan();
    outIt.NextSpan();
   }
}
```

The work of the filter is done by the middle portion. The input value can be accessed by pointer dereferencing (.e.g *inSI). This could be written in longhand as:

```
  // Get input
  IT inval = (IT) (*inSI);
  // Perform the operation in double
  double val = ((double)(*inSI) + shift) * scale;

  // If we are checking for clamping verify legal range
      if (clamp){
         if (val > typeMax)
          val = typeMax;
         if (val < typeMin)
          val = typeMin;
      }
```

```
  // Set the Output
*outSI = (OT)(val);
```

Note: The two functions vtkImageShiftScaleExecute1 and vtkImageShiftScaleExecute do not appear in any header (.h) file. They are an example of private implementation, and are only known to other functions in the same source (.cpp) file. However, in this case, the order in which the functions appear in the .cpp file is important. A function can not call a function that has not already been defined – or at the very least it's interface has been defined. Hence, in the source file, vtkImageShiftScaleExecute appears first followed by vtkImageShiftScaleExecute1 and vtkImageShiftScale::ThreadedExecute appears last.

Unfortunately, for many filters, the VTK 4.4 iterators are not sufficiently powerful. Many filters (e.g. vtkImage-GaussianSmooth) rely on direct pointer manipulations instead.

STL Notes: If you want understand the operations of these iterators better, I suggest reading through some of the tutorials on the STL e.g. `http://www.cprogramming.com/tutorial/stl/iterators.html`. The STL defines many interesting classes, that can be of great use. It is, unfortunately beyond the scope of this introductory material.

Chapter 23

Copying Data Objects

23.1 Introduction

Most of the VTK examples you will see are essentially single pipelines. This gives the appearance that the only way to process data in VTK is to take raw data as an input, pass it through various filters for manipulation and display the output in a single connected pipeline. In larger projects, however, we often need to do a small amount of processing and return the result. Later this result, perhaps as a response to some user input via a GUI, will then be processed some more to yield a different output etc.

In this chapter we discuss ways of "breaking the single pipeline" and returning intermediate results. Key to this process is the ability to copy data-objects. We first discuss three different methods for doing this at varying levels of completenesss. This is followed by two concrete examples. The first shows how to *properly* extract an isosurface from an image and the second how to compute a gradient of smoothed image. In both examples, the focus is on proper implementation and returning of the data-object as opposed to the details of the algorithms themselves.

23.2 CopyStructure, ShallowCopy and DeepCopy

It is often desirable to make a copy of a dataset. Datasets such as vtkPolyData and vtkImageData support, at least, three different means of copying. These are:

1. **CopyStructure:** This copies the geometric structure of an input data-set. In the case of vtkImageData this simply copies the image dimensions, origin, spacing, number of scalar components and data type (and a few other miscellaneous members). It essentially creates an image of the same "size" as the input, but does not allocate memory. This is very useful as an initialization of a filter method, where the output image is of the same size and type as the input image:

   ```
   vtkImageData* out=vtkImageData::New();
   out->CopyStructure(input);
   // Now modify as desired
   out->SetScalarTypeToFloat();
   // At this point only allocate memory
   out->AllocateScalars();
   ```

2. **ShallowCopy:** ShallowCopy is essentially the same as CopyStructure with the *key addition* that all array data (e.g. intensities) are linked! Shallow Copy for a vtkImageData (in a very simplified form)

```
vtkImageData* out=vtkImageData::New();
out->ShallowCopy(input);

// or equivalently
out->CopyStructure(input);
out->GetPointPointData()->SetScalars(input->GetPointData()->GetScalars());
```

All pointer-based objects, which include all vtkDataArray structures are simply passed as pointers (with appropriate reference count increases). No new memory is allocated to store these. Any modification to the intensities in "out", also modifies the intensities in "input" as their intensities are stored in the exact same array!

ShallowCopy is extremely useful for preserving the results of a filter or a combination of filters (a pipeline) while destroying the actual pipeline. This is illustrated in the two examples later in this document.

3. **DeepCopy:** DeepCopy creates a complete duplicate version of a data-object. This allocates all necessary memory and creates a separate but identical object to the input.

```
vtkImageData* out=vtkImageData::New();
out->DeepCopy(input);
```

Use DeepCopy sparingly, unless you absolutely need a complete duplicate copy of an image prior to modification.

23.3 Example 1: Extracting a Surface from a LevelSet Function

Consider, for example, an implementation of a Levelset segmentation algorithm, which results in a distance map (or levelset function) stored in an image LevelsetImage. Either during the evolution of the levelset, or at the end, we may need to return a surface extracted from the zero-levelset. (The zero-levelset conventionally represents the output surface of the segmentation.) This may be conveniently implemented in a member function `GetZeroSurface`.

A first attempt at implementing this function can take the form:

```
vtkPolyData* vtkMyLevelsetFilter::GetZeroSurface() {
  // this->LevelsetImage is of type vtkImageData

  vtkContourFilter *ContourFilter = vtkContourFilter::New();
  ContourFilter->SetInput(this->LevelsetImage);
  ContourFilter->SetValue(1, 0.0);
  ContourFilter->Update();
  return ContourFilter->GetOutput();
}
```

This will work, but there is a key problem associated with it. The problem is that we return a pointer to the output of a filter, without returning the actual filter itself. This can mess up the reference counting scheme in VTK.

Consider the case where we call this filter once. The filter `ContourFilter` is first created. Then we set the image `this->LevelsetImage` as its input, which results in the reference counter of the image being incremented by one.

Then we call the filter's `Update` function. This, incidentally, *is critical*. VTK pipelines operate on a lazy executing scheme, so the filter will not do anything unless it has to, using `Update` forces the filter to go to work. (In normal pipelines, i.e. source to display, updating the display propagates an update event backwards through the pipeline and forces all intermediate filters to update!).

The filter results in a surface (vtkPolyData) which is then returned to the user.

Next time we call the filter, we create a new `ContourFilter` object and set `this->LevelsetImage` as its input, which results in the reference counter of the image, again, *being incremented by one*.

If we call this function 100 times then `LevelsetImage` will have a reference count of 100 which means that it will never be deleted even when the LevelsetFilter class is deleted, resulting in a potential memory leak in a large piece of software.

The correct solution to this problem, is to *copy the result of the filter output*, delete the filter, and return this copy.

```
vtkPolyData* vtkMyLevelsetFilter::GetZeroSurface()
{
  vtkContourFilter *ContourFilter = vtkContourFilter::New();
  ContourFilter->SetInput(this->LevelsetImage);
  ContourFilter->SetValue(1, 0.0);
  ContourFilter->Update();

  vtkPolyData* zerosurface=vtkPolyData::New();
  zerosurface->ShallowCopy(ContourFilter->GetOutput());
  ContourFilter->Delete();

  return zerosurface;
}
```

Here, we first create a temporary surface (zerosurface). Next we perform a shallow copy operation which copies the contents of the output of the ContourFilter to this temporary surface. Then the ContourFilter is deleted, cleaning up all reference counting issues. At this point we return the `zerosurface` object to the calling code. The calling function is then responsible to *delete* the `zerosurface` object when it is done with it, this object has no attachments to any lingering pipeline code.

The Recipe: In the many cases where one needs to use a VTK pipeline to generate an output data structure (as opposed to an output display or file) and return it, the following recipe can be very useful:

- Create the pipeline
- **Call the Update** function of the last filter – this will invoke, in turn, the Update functions of all the previous filters
- Create a new output structure (e.g. image or surface most likely)
- **ShallowCopy** the output of the final filter to the new output structure.
- **Delete** all filters in the pipeline.
- **Return** the output structure.

A slight variation on the above example, in which the level is specified as opposed to assumed to be zero, is given in the class vtkMyUtility.cpp.

23.4 Example 2: An Image Processing Example

Consider the case where one needs to compute the gradient of an image at a specific scale. This requires (i) first smoothing the image and (ii) computing the gradient. This operation can be accomplished by a pipeline shown below:

```
vtkImageData* vtkMyUtility::SmoothImageAndComputeGradient(vtkImageData* input,
  double sigma,int dimensionality){
  vtkImageGaussianSmooth* sm=vtkImageGaussianSmooth::New();
  sm->SetInput(input);
  sm->SetStandardDeviations(sigma,sigma,sigma);
  sm->SetDimensionality(dimensionality);

  vtkImageGradient* gradient=vtkImageGradient::New();
  gradient->SetInput(sm->GetOutput());
  sm->Delete();
  gradient->SetDimensionality(dimensionality);
  gradient->Update();

  vtkImageData* grad=vtkImageData::New();
  grad->ShallowCopy(gradient->GetOutput());
  gradient->Delete();
  return grad;
}
```

Note that we follow the same recipe. The pipeline is first created. Then, the `Update` function of the last filter `gradient->Update()` is called to force execution. Next, we create a new output data structure (grad) and perform the shallow copy operation. The pipeline is deleted as usual and the output image is then returned.

23.5 Implementation

Both of these examples are implemented in a class vtkMyUtility. The header file of this has the form:

```
class vtkMyUtility : public vtkObject {
public:
  static vtkMyUtility *New();
  vtkTypeMacro(vtkMyUtility,vtkObject);
  // Example 1 -- Extract Iso-Contour
  vtkPolyData* ExtractContour(vtkImageData *img,double level);
  // Example 2 --  Smooth Image and Compute Gradient
  static vtkImageData* SmoothImageAndComputeGradient(vtkImageData* img,double sigma,
                              int dimensionality);
protected:
};
```

A script (script23-1.tcl) exercises these functions and outputs a surface and a gradient image respectively.

Chapter 24

The Insight Toolkit

24.1 Introduction

The Insight Toolkit (ITK) is an open source software toolkit for registration and segmentation. It is implemented in C++ and uses the CMake build environment – in fact CMake was developed for the ITK project. ITK was started in 1999 under a contract by the US National Library of Medicine of the National Institutes of Health to combination of academic and industrial partners.

While ITK can be thought off as a "first-cousin" of VTK, there is one critical difference. ITK is implemented using generic programming principles. It uses templates both for the algorithm implementation and, unlike VTK, the class interfaces themselves. This type of heavily templated C++ code challenges many compilers and it can take much longer to compile. The other difference, is that the memory model depends on "smart pointers" that maintain a reference count to objects. Smart pointers can be allocated on the stack, and when scope is exited, the smart pointers disappear and decrement their reference count to the object that they refer to. There is no need to call `itkFilter->Delete()`, unlike VTK filters.

The use of ITK, especially by beginners, is more challenging than VTK. The use of generic programming techniques assumes a firm grounding in templated programming in general, and the Standard Template Library in particular. The use of templated filters also makes the use of the toolkit from languages other than C++ less elegant than VTK.

The use of templates in the interface has one negative consequence. While the use of templated classes can simplify the filter design – there is no need for multiple switch statements as is the case in VTK templated implementations, it results in the need to explicitly specify image types at compile time. In contrast, in VTK one can allocate a vtkImageData object first, and then dynamically set its type, and potentially even change its type later. In ITK images need to be allocated explicitly with a fixed type at compile time, which is a limitation in using it to develop larger systems.

To elaborate this further, consider the case of allocating an image. In VTK, this is accomplished as:

```
vtkImageData* img=vtkImageData::New();
img->SetScalarTypeToFloat();
```

By contrast, in ITK, the same task requires either:

```
itk::Image< float , 3 >::Pointer img=itk::Image< float , 3 >::New();
```

where 3 is the image dimension and float is the image type, or, making use of the typedef construct to create shorthands for the complex type names, the following:

```
typedef   itk::Image< float , 3 >   ImageType
typename  ImageType::Pointer img=ImageType::New();
```

As you can see, things can get ugly pretty quickly.[1]

Learning to program using ITK, i.e. implementing algorithms by leveraging ITK code, is beyond the scope of this class. However, there are a lot of algorithms in ITK that can be usefully exploited for many tasks. In addition ITK has a nice image I/O framework which supports a large number of image formats. In the rest of this handout, we will focus on using ITK in a VTK-centric environment. This will be accomplished by hiding ITK code inside functions which take VTK images as inputs and return VTK images as outputs. Within such functions, we can convert the images to ITK data structures, do the operation and convert. In this way ITK code is safely packaged in a VTK 'wrapper'.

The techniques we will use to accomplish this type are similar to those described in our discussion on templated VTK filters. We will again use ordinary specially named templated functions, inserted in VTK source which are called from within the member functions. All ITK code will reside primarily within these templated functions.

24.2 The vtkITKMyUtility Class

This is a simple class which implements three static member functions: (i) CurvatureAnisotropicDiffusion – this calls an ITK smoothing filter by the same name, (ii) LoadImage – this can be used to load an image using the ITK IO Factory and (iii) SaveImage – this can be used to save an image. In the case of the last two functions, the file type is automatically determined by the image name! Note that the header below has no traces of anything to do with ITK:

```
#include <vtkObject.h>
class vtkImageData;
class vtkITKMyUtility : public vtkObject {
public:
  static vtkITKMyUtility *New();
  vtkTypeMacro(vtkITKMyUtility,vtkObject);

  static vtkImageData* CurvatureAnisotropicDiffusionFilter(vtkImageData* input,
                double Conductance=1.0,
                double TimeStep=0.15,int NumberOfIterations=8);
  static vtkImageData* LoadImage(char* fname);
  static int           SaveImage(vtkImageData* input,char* filename);
protected:
```

[1] It must be acknowledged that the ability have code that can handle images of arbitrary types and dimensions has its own aesthetic appeal. However, inside the code all the types are of some user defined type which can be dizzying until one gets used to it.

```
  vtkITKMyUtility() {};
  virtual ~vtkITKMyUtility() {};
};
```

Each of the three static member functions, calls special templated utility ordinary functions placed in vtkITKMyUtility.cpp to do the templated operations. Within these ordinary functions we will make use of two convenience classes that come as part of a supplementary distribution called InsightApplications that is also available from the ITK web-page. These classes are VTKImageToImageFilter and ImageToVTKImageFilter. The notation used here is that the work "Image" signifies an ITK image, whereas the word "VTKImage" is used to signify a vtkImageData structure.

These filters make use of pairs of classes known as importers and exporters. These can export and import an image from/to a naked C-like pointer. VTK has a pair of these classes called vtkImageExport and vtkImageImport. They convert from VTK Images to ITK Images and back.

24.3 Curvature Anisotropic Diffusion Filtering

This is a type of nonlinear image smoothing that tries to smooth uniform areas while preserving sharp discontinuities. It is available in ITK as part of the `itk::CurvatureAnisotropicDiffusionImageFilter` class.[2]

The Filtering Function: The real work is handled by the following doubly-templated function – note that this **is not a member of vtkITKMyUtility** but just an ordinary function:

```
template <class IT,int dimension>
void vtkITKMyUtilityCurvatureAnisotropicDiffusionSmoothImage(vtkImageData* input,
   vtkImageData* output,
   double Conductance,
   double TimeStep,
   int NumberOfIterations)

{
  // Define the parts (types) of the ITK pipeline.
  typedef itk::Image<IT, dimension> ImageType;
  typedef itk::VTKImageToImageFilter<ImageType> VTKImageToImageFilterType;
  typedef itk::ImageToVTKImageFilter<ImageType> ImageToVTKImageFilterType;
  typedef itk::CurvatureAnisotropicDiffusionImageFilter<ImageType, ImageType >  FilterType;

  // From VTK to ITK
  //-----------------------------------------------------------------------
  typename VTKImageToImageFilterType::Pointer importer=VTKImageToImageFilterType::New();
  importer->SetInput(input);
  importer->Update();

  // Create the itk::CurvatureAnisotropicDiffusionImageFilter and connect it
  // -----------------------------------------------------------------------
  typename FilterType::Pointer filter = FilterType::New();
  filter->SetInput(importer->GetOutput());
```

[2] ITK also makes use of C++ namespaces – these are similar to the Tcl namespaces that we looked at earlier.

```
  filter->SetTimeStep(TimeStep);
  filter->SetNumberOfIterations(NumberOfIterations);
  filter->SetConductanceParameter( Conductance );
  filter->Update();

  // Back to VTK
  //-------------------------------------------------------------------------
  typename ImageToVTKImageFilterType::Pointer exporter=ImageToVTKImageFilterType::New();
  exporter->SetInput(filter->GetOutput());
  exporter->Update();

  output->DeepCopy(exporter->GetOutput());
}
```

The code is fairly straightforward, once one gets over the messy type definitions at the top. To avoid writing constantly things like `Image<IT,dimension>` we create shorthands for all the types at the top using the *typedef* operator.

Next we convert the input image which comes as a vtkImageData to an itkImage using a properly templated instance of VTKImageToImageFilter. Following this we are into straight ITK code as lifted from one of the examples that came with ITK. This naturally results in an itkImage which we then convert back to VTK using an instance of ImageToVTKImageFilter.

Finally the result is copied to an out image using a *DeepCopy* operation which ensures that the data will still be around once the pipeline is delete. Crossing over toolkits has overheads – this is one of them.

Note that *we do not delete the ITK objects*. ITK uses smart pointers which essentially *delete themselves*. ITK came into being a few years after VTK and in some respects has some significant improvements. This is one of them.

An Intermediate Function: The function above is called by an intermediate function that takes care of the second template argument, the image dimension. This is fairly straightforward:

```
template <class IT>
void
vtkITKMyUtilityCurvatureAnisotropicDiffusionSmoothImage1(vtkImageData* indata,
       vtkImageData* outdata,double Conductance,double TimeStep,
       int NumberOfIterations,int dimension,IT *)
{
if (dimension==2)
 vtkITKMyUtilityCurvatureAnisotropicDiffusionSmoothImage<IT,2>(indata,outdata,Conductance,
                                                               TimeStep,NumberOfIterations);
else
 vtkITKMyUtilityCurvatureAnisotropicDiffusionSmoothImage<IT,3>(indata,outdata,Conductance,
                                                               TimeStep,NumberOfIterations);
}
```

The Class Member Function: The two functions above are buried inside vtkITKMyUtility.cpp and are inaccessible to the outside world. Outside code access the filtering operation through the following class member

function:

```
vtkImageData*
vtkITKMyUtility::CurvatureAnisotropicDiffusionFilter(vtkImageData* input,
           double Conductance,double TimeStep,int NumberOfIterations)
{
  if (input==NULL)
     return NULL;

  vtkImageData* output=vtkImageData::New();
  output->CopyStructure(input);

  int dimension=3;
  int dim[3]; input->GetDimensions(dim);
  if (dim[2]==1)
    dimension=2;

  switch (input->GetScalarType())
    {
      vtkTemplateMacro7(vtkITKMyUtilityCurvatureAnisotropicDiffusionSmoothImage1, input,output,
                        Conductance,TimeStep,NumberOfIterations,dimension,
                        static_cast<VTK_TT *>(0));
    }
  return output;
}
```

This first creates the output image. Then it checks whether the input image is really a 2D Image. If it is the dimension variable is set to 2 otherwise it stays at 3. Finally, we use the vtkTemplateMacro7 to call the previous ordinary function. The vtkTemplateMacros were discussed in more detail in Chapter 22.

24.4 The LoadImage Function

ITK has a very nice Image Factory IO setup. Here the user only needs to specify an image filename and the appropriate Image Reader class is instantiated depending on the filename (e.g. Analyze, TIF, PNG) and loads the image.

The workhorse function: As before, we will use a combination of an ordinary templated function and a non-templated member function to perform the operation. First the actual method that loads the image:

```
template <class IT>
vtkImageData* vtkITKMyUtilityLoadImage(char* fname)
{
  typedef itk::Image< IT,  3 >    ImageType;
  typedef itk::ImageToVTKImageFilter<ImageType> ImageToVTKImageFilterType;
  typedef itk::ImageFileReader< ImageType  >  ReaderType;
```

```
  typename ReaderType::Pointer reader = ReaderType::New();
  reader->SetFileName(fname);
  reader->Update();

  typename ImageToVTKImageFilterType::Pointer exporter=ImageToVTKImageFilterType::New();
  exporter->SetInput(reader->GetOutput());
  exporter->Update();

  vtkImageData* output=vtkImageData::New();
  output->DeepCopy(exporter->GetOutput());
  return output;
}
```

This is similar to the smoothing filter above, other than for the fact that here we only have an output.

The Interface Function: The outside class accesses the Load Image functionality through the following member function. This has two parts: (i) First we read the image information to identify the image type using a trick posted on the ITK-users mailing list. Then an explicit switch statement is used to call the workhorse function above with the proper template argument:

```
vtkImageData* vtkITKMyUtility::LoadImage(char* filename)
{
  //  Some of this code derives from code posted by Hideaki Hiraki
  //  on the Insight-users mailing list
  itk::ImageIOBase::Pointer imageIO;
  imageIO=itk::ImageIOFactory::CreateImageIO(filename, itk::ImageIOFactory::ReadMode);
  imageIO->SetFileName(filename);
  imageIO->ReadImageInformation();

  switch( imageIO->GetComponentType() ){
  case itk::ImageIOBase::UCHAR:
    return vtkITKMyUtilityLoadImage<unsigned char>(filename);
    break;
  case itk::ImageIOBase::CHAR:
    return vtkITKMyUtilityLoadImage<char>(filename);
    break;
  case itk::ImageIOBase::SHORT:
    return vtkITKMyUtilityLoadImage<short>(filename);
    break;
  case itk::ImageIOBase::FLOAT:
    return vtkITKMyUtilityLoadImage<float>(filename);
    break;
    // ....  Lot's more case statements omitted to save space.
  }

  return NULL;
}
```

The implementation of the Save Image function is similar to the smoothing filter and will not be discussed in any detail here.

24.5 The CMakeLists.txt File

This is an example of a combined VTK/ITK project. Both VTK and ITK must be found. The two utility classes itkImageToVTKImageFilter and itkVTKImageToImageFilter are implemented in .txx files to signify that this is templated C++ code. Both of these files are marked with a WRAP_EXCLUDE flag to tell the VTK Tcl Wrappers to ignore them, as these wrappers are incapable of handling templated code.

```
PROJECT(VTKITK)
SET(KITBASE VTKITK)
SET(KIT     vtk${KITBASE})

INCLUDE (${CMAKE_ROOT}/Modules/FindVTK.cmake)
FIND_PACKAGE(VTK REQUIRED)
IF (USE_VTK_FILE)
  INCLUDE(${USE_VTK_FILE})
ENDIF(USE_VTK_FILE)

FIND_PACKAGE(ITK REQUIRED)
IF (USE_ITK_FILE)
  INCLUDE(${USE_ITK_FILE})
ENDIF(USE_ITK_FILE)

INCLUDE_DIRECTORIES(${VTKITK_SOURCE_DIR})
SET (LIBRARY_OUTPUT_PATH ${VTKITK_SOURCE_DIR})

SET(LIBRARY_SRCS
itkImageToVTKImageFilter.txx
itkVTKImageToImageFilter.txx
vtkITKMyUtility.cpp)

SET_SOURCE_FILES_PROPERTIES(
itkImageToVTKImageFilter.txx
itkVTKImageToImageFilter.txx
 WRAP_EXCLUDE)

LINK_LIBRARIES(
vtkCommon
vtkCommonTCL
${ITK_LIBRARIES})
ADD_LIBRARY(${KIT} STATIC ${LIBRARY_SRCS})
VTK_WRAP_TCL (${KIT}TCL LIBRARY_TCL_SRCS ${LIBRARY_SRCS})
ADD_LIBRARY (${KIT}TCL SHARED ${LIBRARY_TCL_SRCS} ${LIBRARY_SRCS})
```

The script `script24-2.tcl` is used to exercise the code. This looks an image in the new "NIFTY" format, smooths it and saves it out in analyze format!

```
lappend auto_path [ file dirname [ info script ]]
package require newname
if { $tcl_platform(platform) == "windows" } {
    load debug/vtkVTKITKTCL.dll
} else {
    load libvtkVTKITKTCL.so
}
set util [ vtkITKMyUtility  [ newname::vnewobj ] ]
set inimg [ $util LoadImage  avg152T1_LR_nifti.nii.gz ]
puts stderr "Image Loaded [ $inimg GetDimensions ]"

puts stderr "Calling Curvature Anistropic Diffusion Filter"
set img [ $util CurvatureAnisotropicDiffusionFilter $inimg 1.0 0.05 8 ]

puts stderr "Done [ $img GetDimensions ] on to saving"
$util SaveImage $img  "itksmooth.hdr"
exit
```

24.6 Concluding Remarks

The Insight Toolkit (ITK) is another large object-oriented library that offers a great deal of functionality for medical image analysis. It has among others, some very nice implementations of the Levelset method, Finite Element Code, Registration code etc. Unfortunately, in my opinion, the use of a fully generic programming style requires a level of C++ expertise that makes it difficult to recommend whole-heartedly to a beginner. The methodology presented in this chapter aims to demonstrate how one can take advantage of ITK code by neatly packaging it inside VTK.

Part VII

Appendices

Appendix A

Final Exam

Programming

1. Implement, using C++ and VTK, a reasonably complex medical image analysis algorithm of your choice. This **must** be an algorithm that you **have not previously implemented in C++** – converting code from MATLAB is allowed. The implementation must use VTK data structures for images, surfaces etc. If you need suggestions, I have placed several papers in a sudirectory called `final/papers`; any one of these will be fine.
   (a) The algorithm must be implemented in C++.
   (b) All code must be in classes deriving from VTK classes (e.g. vtkProcessObject)
2. Compile the algorithm into a shared library that is accessible from (i.e. loadable into) Tcl.
3. Implement using Tcl a command-line script for quickly testing the algorithm. This should load a synthetic image of your choice, execute the algorithm and generate some output result.
4. Implement using Tcl/[Incr] Tcl a complete application for interacting with your algorithm. This must include:
   (a) A graphical user interface for parameter setting and executing the algorithm.
   (b) An integrated interactive viewer for displaying the results.
   (c) You may leverage BioImage Suite components if you wish to do so.

Report

The report should have three parts:

1. A brief (1-2 page) description of the selected algorithm.
2. A description of the implementation strategy (e.g. how the algorithm was broken up into different classes etc.), and description of the functions in your code. (This is the heart of the report 4-6 pages, longer if needed).
3. A brief (2 pages) User's Guide for your application. Including snapshots of key GUI elements etc.

The lecture notes for Sessions 20 and Session 21 are typical examples of what is expected for Parts 1 and 2.

Appendix B

Code License

All example code from this book is made available under the following BSD-style open source license.

Bibliography

[1] Cygwin: a Linux-like environment for Windows. http://www.cygwin.com/.

[2] Brainsuite 2: a magnetic resonance (MR) image analysis tool designed for identifying tissue types and surfaces in MR images of the human head. http://brainsuite.usc.edu/.

[3] Metakit: an efficient embedded database library. http://www.equi4.com/metakit.html.

[4] P. J. Besl and N. D. Mackay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[5] BrainLAB, Heimstetten, Germany. *http://www.brainlab.com/*.

[6] H. Chui, L. Win, R. T. Schultz, J. S. Duncan, and A. Rangarajan. A unified non-rigid feature registration method for brain mapping. *Medical Image Analysis*, 7(2):113–130, 2003.

[7] M. DiStasio, K. Vives, and X. Papademetris. The BioImage Suite Datatree Tool: Enabling flexible realtime surgical visualizations. In *ISC/NA-MIC Workshop on Open Science at MICCAI 2006*, 2006. http://hdl.handle.net/1926/208.

[8] Doxygen. http://www.doxygen.org/.

[9] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, N.Y., 1973.

[10] TCL Developer Exchange. http://www.tcl.tk/.

[11] S. Geman and D. Geman. Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

[12] L. Ibanez and W. Schroeder. *The ITK Software Guide: The Insight Segmentation and Registration Toolkit*. Kitware, Inc., Albany, NY, `www.itk.org`, 2003.

[13] [incr Tcl]. http://incrtcl.sourceforge.net/itcl/.

[14] GNU General Public License. http://www.gnu.org/copyleft/gpl.htm.

[15] Matlab. http://www.mathworks.com/products/matlab/.

[16] OpenGL. http://www.opengl.org/.

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

[18] X. Papademetris. Programming for medical image analysis using VTK, http://noodle.med.yale.edu/papad/seminar/.

[19] X. Papademetris, M. Jackowski, N. Rajeevan, R.T. Constable, and L.H Staib. BioImage Suite: An integrated medical image analysis suite, Section of Bioimaging Sciences, Dept. of Diagnostic Radiology, Yale School of Medicine, `http://www.bioimagesuite.org`.

[20] X. Papademetris, M. Jackowski, N. Rajeevan, M. DiStasio, H. Okuda, R. T. Constable, and L. H. Staib. BioImage Suite: An integrated medical image analysis suite: An update. In *ISC/NA-MIC Workshop on Open Science at MICCAI 2006*, 2006. http://hdl.handle.net/1926/209.

[21] X. Papademetris, K. P. Vives, M. DiStasio, L. H. Staib, M. Neff, S. Flossman, N. Frielinghaus, H. Zaveri, E. J. Novotny, H. Blumenfeld, R. T. Constable, H. P. Hetherington, R. B. Duckrow, S. S. Spencer, D. D. Spencer, and J. S. Duncan. Development of a research interface for image guided intervention: Initial application to epilepsy neurosurgery. In *International Symposium on Biomedical Imaging ISBI*, pages 490–493, 2006.

[22] Apache HTTPD Server Project. http://httpd.apache.org/.

[23] The Netlib repository. http://www.netlib.org.

[24] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., Albany, NY, `www.vtk.org`, 2003.

[25] VWware Server. http://www.vmware.com/products/server/.

[26] D. Shreiner, M. Woo, J. Neidera, and T. Davis. *OpenGL: Programming Guide: The official guide to learning OpenGL, Version 1.4*. Addison-Wesley Publishing, fourth edition, 2004.

[27] C. Smith. *[Incr-tcl/tk] from the Ground Up*. McGraw-Hill, 2000.

[28] B. Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, 1991.

[29] Subversion: A version control system. http://subversion.tigris.org/.

[30] 3D Slicer: Medical Visualization and Processing Environment for Research. http://www.slicer.org.

[31] B. Welch, K. Jones, and J. Hobbs. *Practical Programming in Tcl and Tk: 4th Edition*. Prentice Hall, 2003.

[32] Y. Zhang, M. Brady, and S. Smith. Segmentation of brain MR images through a hidden markov random field model and the expectation maximization algorithm. *IEEE Trans. Med. Imag.*, 20(1):45–57, 2001.