The ITK Software Guide
Book 1: Introduction and Development Guidelines
Fourth Edition
Updated for ITK version 4.7

Hans J. Johnson, Matthew M. McCormick, Luis Ibáñez,
and the *Insight Software Consortium*

February 17, 2015

http://itk.org
Email: community@itk.org

The purpose of computing is Insight, not numbers.

Richard Hamming

ABSTRACT

The Insight Toolkit (ITK) is an open-source software toolkit for performing registration and segmentation. *Segmentation* is the process of identifying and classifying data found in a digitally sampled representation. Typically the sampled representation is an image acquired from such medical instrumentation as CT or MRI scanners. *Registration* is the task of aligning or developing correspondences between data. For example, in the medical environment, a CT scan may be aligned with a MRI scan in order to combine the information contained in both.

ITK is a cross-platform software. It uses a build environment known as CMake to manage platform-specific project generation and compilation process in a platform-independent way. ITK is implemented in C++. ITK's implementation style employs generic programming, which involves the use of templates to generate, at compile-time, code that can be applied *generically* to any class or data-type that supports the operations used by the template. The use of C++ templating means that the code is highly efficient and many issues are discovered at compile-time, rather than at run-time during program execution. It also means that many of ITK's algorithms can be applied to arbitrary spatial dimensions and pixel types.

An automated wrapping system integrated with ITK generates an interface between C++ and a high-level programming language Python. This enables rapid prototyping and faster exploration of ideas by shortening the edit-compile-execute cycle. In addition to automated wrapping, the SimpleITK project provides a streamlined interface to ITK that is available for C++, Python, Java, CSharp, R, Tcl and Ruby.

Developers from around the world can use, debug, maintain, and extend the software because ITK is an open-source project. ITK uses a model of software development known as Extreme Programming. Extreme Programming collapses the usual software development methodology into a simultaneous iterative process of design-implement-test-release. The key features of Extreme Programming are communication and testing. Communication among the members of the ITK community is what helps manage the rapid evolution of the software. Testing is what keeps the software stable. An extensive testing process supported by the system known as CDash measures the quality of ITK code on a daily basis. The ITK Testing Dashboard is updated continuously, reflecting the quality of

the code at any moment.

The most recent version of this document is available online at http://itk.org/ItkSoftwareGuide.pdf. This book is a guide to developing software with ITK; it is the first of two companion books. This book covers building and installation, general architecture and design, as well as the process of contributing in the ITK community. The second book covers detailed design and functionality for reading and writing images, filtering, registration, segmentation, and performing statistical analysis.

CONTRIBUTORS

The Insight Toolkit (ITK) has been created by the efforts of many talented individuals and prestigious organizations. It is also due in great part to the vision of the program established by Dr. Terry Yoo and Dr. Michael Ackerman at the National Library of Medicine.

This book lists a few of these contributors in the following paragraphs. Not all developers of ITK are credited here, so please visit the Web pages at http://itk.org/ITK/project/parti.html for the names of additional contributors, as well as checking the GIT source logs for code contributions.

The following is a brief description of the contributors to this software guide and their contributions.

Luis Ibáñez is principal author of this text. He assisted in the design and layout of the text, implemented the bulk of the LaTeX and CMake build process, and was responsible for the bulk of the content. He also developed most of the example code found in the `Insight/Examples` directory.

Will Schroeder helped design and establish the organization of this text and the `Insight/Examples` directory. He is principal content editor, and has authored several chapters.

Lydia Ng authored the description for the registration framework and its components, the section on the multiresolution framework, and the section on deformable registration methods. She also edited the section on the resampling image filter and the sections on various level set segmentation algorithms.

Joshua Cates authored the iterators chapter and the text and examples describing watershed segmentation. He also co-authored the level-set segmentation material.

Jisung Kim authored the chapter on the statistics framework.

Julien Jomier contributed the chapter on spatial objects and examples on model-based registration using spatial objects.

Karthik Krishnan reconfigured the process for automatically generating images from all the examples. Added a large number of new examples and updated the Filtering and Segmentation chapters

for the second edition.

Stephen Aylward contributed material describing spatial objects and their application.

Tessa Sundaram contributed the section on deformable registration using the finite element method.

YinPeng Jin contributed the examples on hybrid segmentation methods.

Celina Imielinska authored the section describing the principles of hybrid segmentation methods.

Mark Foskey contributed the examples on the AutomaticTopologyMeshSource class.

Mathieu Malaterre contributed the entire section on the description and use of DICOM readers and writers based on the GDCM library. He also contributed an example on the use of the VTKImageIO class.

Gavin Baker contributed the section on how to write composite filters. Also known as minipipeline filters.

Since the software guide is generated in part from the ITK source code itself, many ITK developers have been involved in updating and extending the ITK documentation. These include **David Doria**, **Bradley Lowekamp**, **Mark Foskey**, **Gaëtan Lehmann**, **Andreas Schuh**, **Tom Vercauteren**, **Cory Quammen**, **Daniel Blezek**, **Paul Hughett**, **Matthew McCormick**, **Josh Cates**, **Arnaud Gelas**, **Jim Miller**, **Brad King**, **Gabe Hart**, **Hans Johnson**.

Hans Johnson, **Kent Williams**, **Constantine Zakkaroff**, **Xiaoxiao Liu**, **Ali Ghayoor**, and **Matthew McCormick** updated the documentation for the initial ITK Version 4 release.

Luis Ibáñez and **Sébastien Barré** designed the original Book 1 cover. **Matthew McCormick** and **Brad King** updated the code to produce the Book 1 cover for ITK 4 and VTK 6. **Xiaoxiao Liu**, **Bill Lorensen**, **Luis Ibáñez**,and **Matthew McCormick** created the 3D printed anatomical objects that were photographed by **Sébastien Barré** for the Book 2 cover. **Steve Jordan** designed the layout of the covers.

Lisa Avila, **Hans Johnson**, **Matthew McCormick**, **Sandy McKenzie**, **Christopher Mullins**, **Katie Osterdahl**, and **Michka Popoff** prepared the book for the 4.7 print release.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Introduction

CHAPTER

ONE

WELCOME

Welcome to the *Insight Segmentation and Registration Toolkit (ITK) Software Guide*. This book has been updated for ITK 4.7 and later versions of the Insight Toolkit software.

ITK is an open-source, object-oriented software system for image processing, segmentation, and registration. Although it is large and complex, ITK is designed to be easy to use once you learn about its basic object-oriented and implementation methodology. The purpose of this Software Guide is to help you learn just this, plus to familiarize you with the important algorithms and data representations found throughout the toolkit.

ITK is a large system. As a result, it is not possible to completely document all ITK objects and their methods in this text. Instead, this guide will introduce you to important system concepts and lead you up the learning curve as fast and efficiently as possible. Once you master the basics, take advantage of the many resources available [1], including example materials, which provide cookbook recipes that concisely demonstrate how to achieve a given task, the Doxygen pages, which document the specific algorithm parameters, and the knowledge of the many ITK community members (see Section 1.5 on page 8.)

The Insight Toolkit is an open-source software system. This means that the community surrounding ITK has a great impact on the evolution of the software. The community can make significant contributions to ITK by providing code reviews, bug patches, feature patches, new classes, documentation, and discussions. Please feel free to contribute your ideas through the ITK community mailing list.

1.1 Organization

This software guide is divided into three parts. Part I is a general introduction to ITK, with a description of how to install the Insight Toolkit on your computer. This includes how to build the library from its source code. Part II introduces basic system concepts such as an overview of the

[1] http://www.itk.org/ITK/help/documentation.html

system architecture, and how to build applications in the C++ and Python programming languages. Part II also describes the design of data structures and application of analysis methods within the system. Part III is for the ITK contributor and explains how to create your own classes, extend the system, and be an active participant in the project.

1.2 How to Learn ITK

The key to learning how to use ITK is to become familiar with its palette of objects and the ways to combine them. There are three categories of documentation to help with the learning process: high level guidance material (the Software Guide), "cookbook" demonstrations on how to achieve concrete objectives (the examples), and detailed descriptions of the application programming interface (the Doxygen[2] documentation). These resources are combined in the three recommended stages for learning ITK.

In the first stage, thoroughly read this introduction, which provides an overview of some of the key concepts of the system. It also provides guidance on how to build and install the software. After running your first "hello world" program, you are well on your way to advanced computational image analysis!

The next stage is to execute a few examples and gain familiarity with the available documentation. By running the examples, one can gain confidence in achieving results and is introduced the mechanics of the software system. There are three example resources,

1. the `Examples` directory of the ITK source code repository [3].
2. the Examples pages on the ITK Wiki [4]
3. the Sphinx documented ITK Examples [5]

To gain familiarity with the available documentation, browse the sections available in Part II and Part III of this guide. Also, browse the Doxygen application programming interface (API) documentation for the classes applied in the examples.

Finally, mastery of ITK involves integration of information from multiple sources. the second companion book is a reference to algorithms available, and Part III introduces how to extend them to your needs and participate in the community. Individual examples are a detailed starting point to achieve certain tasks. In practice, the Doxygen documentation becomes a frequent reference as an index of the classes available, their descriptions, and the syntax and descriptions of their methods. When examples and Doxygen documentation are insufficient, the software unit tests thoroughly demonstrate how the code is utilized. Last, but not least, the source code itself is an extremely valuable resource.

[2] http://itk.org/Doxygen/index.html
[3] 1.3
[4] http://itk.org/Wiki/ITK/Examples
[5] http://itk.org/ITKExamples

The code is the most detailed, up-to-date, and definitive description of the software. A great deal of attention and effort is directed to the code's readability, and its value cannot be understated.

The following sections describe how to obtain the software, summarize the software functionality in each directory, and how to locate data.

1.3 Obtaining the Software

There are two different ways to access the ITK source code:

Periodic releases Official releases are available on the ITK web site[6]. They are released twice a year, and announced on the ITK web pages and mailing list. However, they may not provide the latest and greatest features of the toolkit.

Continuous repository checkout Direct access to the Git source code repository[7] provides immediate availability to the latest toolkit additions. But, on any given day the source code may not be stable as compared to the official releases.

This software guide assumes that you are using the current released version of ITK, available on the ITK web site. If you are a new user, we recommend the released version of the software. It is stable, consistent, and better tested than the code available from the Git repository. When working from the repository, please be aware of the ITK quality testing dashboard. The Insight Toolkit is heavily tested using the open-source CDash regression testing system[8]. Before updating the repository, make sure that the dashboard is *green*, indicating stable code. (Learn more about the ITK dashboard and quality assurance process in Section 9.2 on page 208.)

1.3.1 Downloading Packaged Releases

ITK can be downloaded without cost from the following web site:

`http://www.itk.org/ITK/resources/software.html`

On the web page, choose the tarball that better fits your system. The options are `.zip` and `.tar.gz` files. The first type is better suited for Microsoft-Windows, while the second one is the preferred format for UNIX systems.

Once you unzip or untar the file a directory called `InsightToolkit-4.7.0` will be created in your disk and you will be ready to start the configuration process described in Section 2.1.1 on page 12.

[6] `http://itk.org/`
[7] `http://itk.org/ITK.git`
[8] `http://open.cdash.org/index.php?project=Insight`

1.3.2 Downloading From Git

Git is a free and open source distributed version control system. For more information about Git please see Section 9.1 on page 207. (Note: please make sure that you access the software via Git only when the ITK quality dashboard indicates that the code is stable.)

Access ITK via Git using the following commands (under a Git Bash shell):

```
git clone git://itk.org/ITK.git
```

This will trigger the download of the software into a directory named `ITK`. Any time you want to update your version, it will be enough to change into this directory, `ITK`, and type:

```
git pull
```

Once you obtain the software you are ready to configure and compile it (see Section 2.1.1 on page 12). First, however, we recommend reading the following sections that describe the organization of the software and joining the mailing list.

1.3.3 Data

The Insight Toolkit was designed to support the Visible Human Project and its associated data. This data is available from the National Library of Medicine at http://www.nlm.nih.gov/research/visible/visible_human.html.

Another source of data can be obtained from the ITK Web site at either of the following:

http://www.itk.org/ITK/resources/links.html
ftp://public.kitware.com/pub/itk/Data/.

1.4 Software Organization

To begin your ITK odyssey, you will first need to know something about ITK's software organization and directory structure. It is helpful to know enough to navigate through the code base to find examples, code, and documentation.

ITK resources are organized into multiple Git repositories. The ITK library source code are in the `ITK`[9] Git repository. The sphinx Examples are in the `ITKExamples`[10] repository. Fairly complex applications using ITK (and other systems such as VTK, Qt, and FLTK) are available from `InsightApplications`[11] repository. The sources for this guide are in the `ITKSoftwareGuide`[12]

[9] http://itk.org/ITK.git
[10] http://itk.org/ITKExamples.git
[11] http://itk.org/ITKApps.git
[12] http://itk.org/ITKSoftwareGuide.git

repository.

The `ITK` repository contains the following subdirectories:

- `ITK/Modules` — the heart of the software; the location of the majority of the source code.
- `ITK/Documentation` — migration guides and Doxygen infrastructure.
- `ITK/Examples` — a suite of simple, well-documented examples used by this guide, illustrating important ITK concepts.
- `ITK/Testing` — a collection of the MD5 files, which are used to link with the ITK data servers to download test data. This test data is used by tests in `ITK/Modules` to produce the ITK Quality Dashboard using CDash. (see Section 9.2 on page 208.)
- `Insight/Utilities` — the scripts that support source code development. For example, CTest and Doxygen support.
- `Insight/Wrapping` — the wrapping code to build interfaces between the C++ library and various interpreted languages (currently Python is supported).

The source code directory structure—found in `ITK/Modules`—is the most important to understand.

- `ITK/Modules/Core` — core classes, macro definitions, typedefs, and other software constructs central to ITK. The classes in `Core` are the only ones always compiled as part of ITK.
- `ITK/Modules/ThirdParty` — various third-party libraries that are used to implement image file I/O and mathematical algorithms. (Note: ITK's mathematical library is based on the VXL/VNL software package[13].)
- `ITK/Modules/Filtering` — image processing filters.
- `ITK/Modules/IO` — classes that support the reading and writing of images, transforms, and geometry.
- `ITK/Modules/Bridge` — classes used to connect with the other analysis libraries or visualization libraries, such as OpenCV[14] and VTK[15].
- `ITK/Modules/Registration` — classes for registration of images or other data structures to each other.
- `ITK/Modules/Segmentation` — classes for segmentation of images or other data structures.
- `ITK/Modules/Video` — classes for input, output and processing of static and real-time data with temporal components.

[13] http://vxl.sourceforge.net
[14] http://opencv.org
[15] http://www.vtk.org

- ITK/Modules/Compatibility — collects together classes for backwards compatibility with ITK Version 3, and classes that are deprecated – i.e. scheduled for removal from future versions of ITK.
- ITK/Modules/Remote — a group of modules distributed outside of the main ITK source repository (most of them are hosted on github.com) whose source code can be downloaded via CMake when configuring ITK.
- ITK/Modules/External — a directory to place in development or non-publicized modules.
- ITK/Modules/Numerics — a collection of numeric modules, including FEM, Optimization, Statistics, Neural Networks, etc.

The Doxygen documentation is an essential resource when working with ITK, but it is not contained in a separate repository. Each ITK class is implemented with a .h and .cxx/.hxx file (.hxx file for templated classes). All methods found in the .h header files are documented and provide a quick way to find documentation for a particular method. Doxygen uses this header documentation to produce its HTML output.

The extensive Doxygen web pages describe in detail every class and method in the system. It also contains inheritance and collaboration diagrams, listing of event invocations, and data members. heavily hyper-linked to other classes and to the source code. The nightly generated Doxygen documentation is online at http://itk.org/Doxygen/html/. Archived versions for each feature release are also available online; for example, the documentation for the 4.4.0 release are available at http://itk.org/Doxygen44/html/.

The ITKApps contains large, relatively complex examples of ITK usage. See the web pages at http://www.itk.org/ITK/resources/applications.html for a description. Some of these applications require GUI toolkits such as Qt and FLTK or other packages such as VTK (*The Visualization Toolkit* [16]). It is recommend to set the CMake source directory to ITKApps/Superbuild to build the dependent third-party applications.

1.5 The Insight Community and Support

Joining the community mailing list is strongly recommended. This is one of the primary resources for guidance and help regarding the use of the toolkit. You can subscribe to the community list online at

http://www.itk.org/ITK/help/mailing.html

ITK was created from its inception as a collaborative, community effort. Research, teaching, and commercial uses of the toolkit are expected. If you would like to participate in the community, there are a number of possibilities. For details on participation, see Part III of this book.

[16] http://www.vtk.org

- Interaction with other community members is encouraged on the mailing lists by both asking as answering questions. When issues are discovered, patches submitted to the code review system are welcome. Performing code reviews, even by novice members, is encouraged. Improvements and extensions to the documentation are also welcome.
- Research partnerships with members of the Insight Software Consortium are encouraged. Both NIH and NLM will likely provide limited funding over the next few years and will encourage the use of ITK in proposed work.
- For those developing commercial applications with ITK, support and consulting are available from Kitware [17]. Kitware also offers short ITK courses either at a site of your choice or periodically at Kitware offices.
- Educators may wish to use ITK in courses. Materials are being developed for this purpose, e.g., a one-day, conference course and semester-long graduate courses. Check the Wiki[18] for a listing.

1.6 A Brief History of ITK

In 1999 the US National Library of Medicine of the National Institutes of Health awarded six three-year contracts to develop an open-source registration and segmentation toolkit, that eventually came to be known as the Insight Toolkit (ITK) and formed the basis of the Insight Software Consortium. ITK's NIH/NLM Project Manager was Dr. Terry Yoo, who coordinated the six prime contractors composing the Insight consortium. These consortium members included three commercial partners—GE Corporate R&D, Kitware, Inc., and MathSoft (the company name is now Insightful)—and three academic partners—University of North Carolina (UNC), University of Tennessee (UT) (Ross Whitaker subsequently moved to University of Utah), and University of Pennsylvania (UPenn). The Principle Investigators for these partners were, respectively, Bill Lorensen at GE CRD, Will Schroeder at Kitware, Vikram Chalana at Insightful, Stephen Aylward with Luis Ibañez at UNC (Luis is now at Kitware), Ross Whitaker with Josh Cates at UT (both now at Utah), and Dimitri Metaxas at UPenn (now at Rutgers). In addition, several subcontractors rounded out the consortium including Peter Raitu at Brigham & Women's Hospital, Celina Imielinska and Pat Molholt at Columbia University, Jim Gee at UPenn's Grasp Lab, and George Stetten at the University of Pittsburgh.

In 2002 the first official public release of ITK was made available. In addition, the National Library of Medicine awarded thirteen contracts to several organizations to extend ITK's capabilities. The NLM has funded maintenance of the toolkit over the years, and a major funding effort was started in July 2010 that culminated with the release of ITK 4.0.0 in December 2011. If you are interested in potential funding opportunities, we suggest that you contact Dr. Terry Yoo at the National Library of Medicine for more information.

[17] http://www.kitware.com
[18] http://itk.org/Wiki/ITK/Documentation

CHAPTER

TWO

CONFIGURING AND BUILDING ITK

This chapter describes the process for configuring and compiling ITK on your system. Keep in mind that ITK is a toolkit, and as such, once it is installed on your computer it does not provide an application to run. What ITK does provide is a large set of libraries which can be used to create your own applications. Besides the toolkit proper, ITK also includes an extensive set of examples and tests that introduce ITK concepts and show how to use ITK in your own projects.

Some of the examples distributed with ITK depend on third party libraries, some of which may need to be installed separately. For the initial build of ITK, you may want to ignore these extra libraries and just compile the toolkit itself.

ITK has been developed and tested across different combinations of operating systems, compilers, and hardware platforms including Microsoft Windows, Linux on various architectures, Solaris/UNIX, Mac OSX, and Cygwin. Kitware is committed to support the following compilers for building ITK:

- GCC 4.x
- Visual Studio 8 SP 1 (until 2015), 9 (until 2018), 10 (until 2020)
- Intel Compiler Suite 11.x, 12.x (including Mac OS X release)
- Darwin-c++-4.2 PPC (until 2015), x86_64
- Win32-mingw-gcc-4.5
- Clang 3.3 and later

If you are currently using an outdated compiler this may be an excellent excuse for upgrading this old piece of software! Support for different platforms is evident on the ITK quality dashboard (see Section 9.2 on page 208).

2.1 Using CMake for Configuring and Building ITK

The challenge of supporting ITK across platforms has been solved through the use of CMake[1], a cross-platform, open-source build system. CMake controls the software compilation process with simple platform and compiler-independent configuration files. CMake is quite sophisticated—it supports complex environments requiring system introspection, compiler feature testing, and code generation.

CMake generates native Makefiles or workspaces to be used with the corresponding development environment of your choice. For example, on UNIX and Cygwin systems, CMake generates Makefiles; under Microsoft Windows CMake generates Visual Studio workspaces; CMake is also capable of generateing appropriate build files for other development environments, e.g., Eclipse. The information used by CMake is provided in `CMakeLists.txt` files that are present in every directory of the ITK source tree. Along with the specification of project structure and code dependencies these files specify the information that need to be provided to CMake by the user during project configuration stage. Typical configuration options specified by the user include paths to utilities installed on your system and selection of software features to be included.

An ITK build requires only CMake and a C++ compiler. ITK ships with all the third party library dependencies required, and these dependencies are used during compilation unless the use of a system version is requested during CMake configuration.

2.1.1 Preparing CMake

CMake can be downloaded at no cost from

http://www.cmake.org/cmake/resources/software.html

You can download binary versions for most of the popular platforms including Microsoft Windows, Mac OSX, Linux, PowerPC and IRIX. Alternatively you can download the source code and build CMake on your system. Follow the instructions provided on the CMake web page for downloading and installing the software. The minimum version of CMake has been evolving along with the version of ITK. For example, the current version of ITK (4.7) requires the minimum CMake version to be 2.8.8.

CMake provides a terminal-based interface (Figure 2.1) on platforms support the `curses` library. For most platforms CMake also provides a GUI based on the Qt library. Figure 2.1 shows the terminal-based CMake interface for Linux and CMake GUI for Microsoft Windows.

Running CMake to configure and prepare for compilation a new project initially requires two pieces of information: where the source code directory is located, and where the compiled code is to be produced. These are referred to as the *source directory* and the *binary directory* respectively. We

[1] www.cmake.org

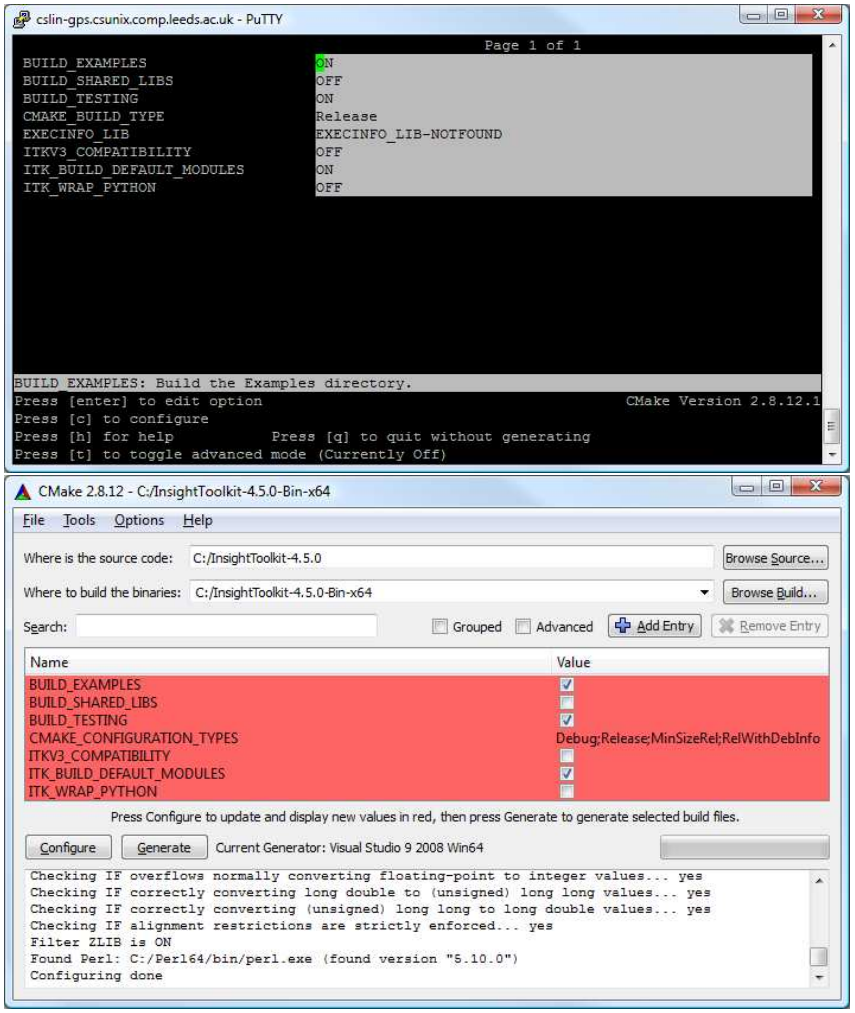

Figure 2.1: CMake user interfaces: at the top is the interface based on the `curses` library supported by UNIX/Linux systems, below is the Microsoft Windows version of the CMake GUI based on the Qt library (CMake GUI is also available on UNIX/Linux systems).

recommend setting the binary directory to be different than the source directory in order to produce an *out-of-source* build.

If you choose to use the terminal-based version of CMake (`ccmake`) the binary directory needs to be created first and then CMake is invoked from the binary directory with the path to the source directory. For example:

```
mkdir ITK-build
cd ITK-build
ccmake ../ITK
```

In the GUI version of CMake (`cmake-gui`) the source and binary directories are specified in the appropriate input fields (Figure 2.1) and the application will request a confirmation to create a new binary directory if it does not exist.

CMake runs in an interactive mode which allows iterative selection of options followed by configuration according to the updated options. This iterative process proceeds until no more options remain to be specified. At this point, a generation step produces the appropriate build files for your configuration.

This interactive configuration process can be better understood by imagining the traversal of a path in a decision tree. Every selected option introduces the possibility that new, dependent options may become relevant. These new options are presented by CMake at the top of the options list in its interface. Only when no new options appear after a configuration iteration can you be sure that the necessary decisions have all been made. At this point build files are generated for the current configuration.

2.1.2 Configuring ITK

Start terminal-based CMake interface `ccmake` on Linux and UNIX, or the graphical user interface `cmake-gui` on Microsoft Windows. Remember to run `ccmake` from the binary directory on Linux and UNIX. On Windows, specify the source and binary directories in the GUI, then set and modify the configuration and build option in the interface as necessary.

The examples distributed with the toolkit provide a helpful resource for learning how to use ITK components but are not essential for compiling the toolkit itself. The testing section of the source tree includes a large number of small programs that exercise the capabilities of ITK classes. Enabling the compilation of the examples and unit tests will considerably increase the build time. In order to speed up the build process, you can disable the compilation of the unit tests and examples. This is done by setting the variables `BUILD_TESTING` and `BUILD_EXAMPLES` to `OFF`.

Most CMake variables in ITK have sensible default values. Each time a CMake variable is changed, it is necessary to re-run the configuration step. In the terminal-based version of the interface the configuration step is triggered by hitting the “c” key. In the GUI version this is done by clicking on the “Configure” button.

When no new options appear highlighted in CMake, you can proceed to generate Makefiles, a Visual Studio workspace or other appropriate build files depending on your preferred development environment. This is done in the GUI interface by clicking on the “Generate” button. In the terminal-based version this is done by hitting the “g” key. After the generation process the terminal-based version of CMake will quit silently. The GUI window of CMake can be left open for further refinement of configuration options as described in the next section. With this scenario it is important to generate new build files to reflect the latest configuration changes. In addition, the new build files need to be

reloaded if the project is open in the integrated development environment such as Visual Studio or Eclipse.

2.1.3 Advanced Module Configuration

Following the default configuration introduced in 2.1.2, the majority of the toolkit will be built. The modern modular structure of the toolkit makes it possible to customize the ITK library by choosing which modules to include in the build. ITK was officially modularized in version 4.0.0 released in December of 2011. Developers have been testing and improving the modular structure since then. The toolkit currently contains more than 100 regular/internal modules and many remote modules, while new ITK modules are being developed.

ITK_BUILD_DEFAULT_MODULES is the CMake option to build all default modules in the toolkit, by default this option is ON as shown in Figure 2.1. The default modules include most internal ITK modules except the ones that depend on external third party libraries (such as ITKVtkGlue, ITKBridgeOpenCV, ITKBridgeVXL, etc.) and several modules containing legacy code (ITKReview, ITKDeprecated and ITKv3Compatibility).

Apart from the default mode of selecting the modules for building the ITK library there are two other approaches module selection: the group mode, and the advanced module mode. When ITK_BUILD_DEFAULT_MODULES is set to OFF, the selection of modules to be included in the ITK library can be customized by changing the variables enabling group and advanced module selection.

ITKGroup_{group name} variables for group module selection are visible when ITK_BUILD_DEFAULT_MODULES is OFF. The ITK source code tree is organized in such way that a group of modules characterised by close relationships or similar functionalities stay in one subdirectory. Currently there are 11 groups (excluding the External and Remote groups). The CMake ITKGroup_{group name} options are created for the convenient enabling or disabling of multiple modules at once. The ITKGroup_Core group is selected by default as shown in Figure 2.2. When a group is selected, all modules in the group and their depending modules are enabled. When a group variable is set to OFF, all modules in the group, except the ones that are required by other enabled modules, are disabled.

If you are not sure about which groups to turn on, but you do have a list of specific modules to be included in your ITK library, you can certainly skip the Group options and use the Module_{module name} options only. Whatever modules you select, their dependent modules are automatically enabled. In the advanced mode of the CMake GUI, you can manually toggle the build of the non-default modules via the Module_{module name} variables. In Figure 2.3 all default modules' Module_{module name} variables are shown disabled for toggling since they are enabled via the ITK_BUILD_DEFAULT_MODULES set to ON variable.

However, not all modules will be visible in the CMake GUI at all times due to the various levels of controls in the previous two modes. If some modules are already enabled by other modes, these modules are set as internal variables and are hidden in the CMake GUI. For example, Module_ITKFoo variable is hidden when the module ITKFoo is enabled in either of the following scenarios:

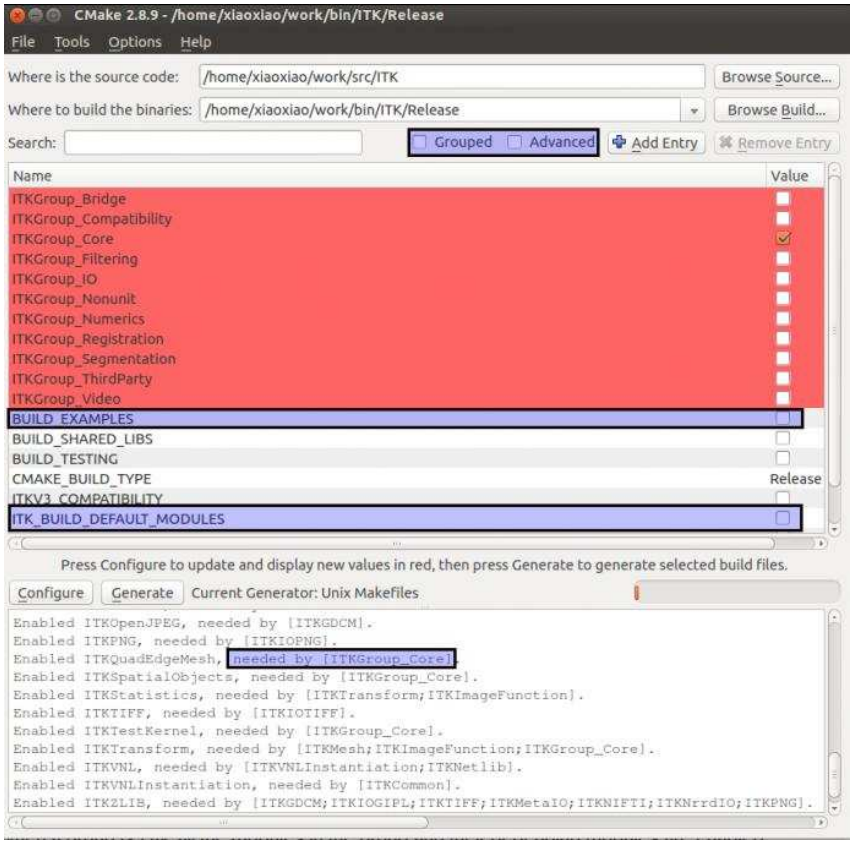

Figure 2.2: CMake GUI shows the ITK Group options.

1. module `ITKBar` is enabled and depends on `ITKFoo`,
2. `ITKFoo` belongs to the group `ITKGroup_FooAndBar` and the group is enabled
3. `ITK_BUILD_DEFAULT_MODULES` is `ON` and `ITKFoo` is a default module.

To find out why a particular module is enabled, check the CMake configuration messages where the information about enabling or disabling the modules is displayed (Figure 2.3); these messages are sorted in alphabetical order by module names.

2.1.4 Compiling ITK

To initiate the build process after generating the build files on Linux or UNIX, simply type `make` in the terminal if the current directory is set to the ITK binary directory. If using Visual Studio,

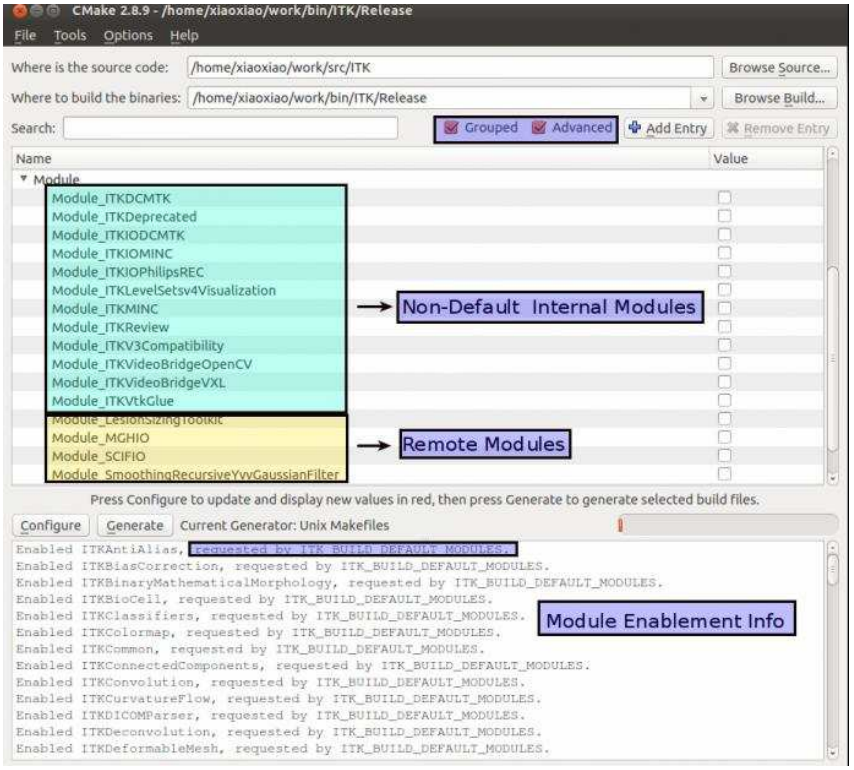

Figure 2.3: CMake GUI for configuring ITK: the advanced mode shows options for non-default ITK Modules.

first load the workspace named `ITK.sln` from the binary directory specified in the CMake GUI and then start the build by selecting "Build Solution" from the "Build" menu or right-clicking on the `ALL_BUILD` target in the Solution Explorer pane and selecting the "Build" context menu item.

The build process can take anywhere from 15 minutes to a couple of hours, depending on the the build configuration and the performance of your system. If testing is enabled as part of the normal build process, about 2400 test programs will be compiled. In this case, you will then need to run `ctest` to verify that all the components of ITK have been correctly built on your system.

2.1.5 Installing ITK on Your System

When the build process is complete an ITK binary distribution package can be generated for installation on your system or on a system with compatible specifications (such as hardware platform and operating system) as well as suitable development environment components (such as C++ compiler and CMake). The default prefix for installation destination directory needs to be specified during CMake configuration process prior to compiling ITK. The installation destination prefix can to be

set through the CMake cache variable CMAKE_INSTALL_PREFIX.

Typically distribution packages are generated to provide a "clean" form of the software which is isolated from the details of the build process (separate from the source and build trees). Due to the intended use of ITK as a toolkit for software development the step of generating ITK binary packages for installing ITK on other systems has limited application and thus it can be treated as optional. However, the step for generating binary distribution packages has a much wide application for distributing software developed with ITK. Further details on configuring and generating binary packages with CMake can be found in the CMake tutorial[2].

2.2 Getting Started With ITK

The simplest way to create a new project with ITK is to create two new directories somewhere in your disk, one to hold the source code and one to hold the binaries and other files that are created in the build process. For this example, create a HelloWorldITK directory to hold the source and a HelloWorldITK-build directory to hold the binaries. The first file to place in the source directory is a CMakeLists.txt file that will be used by CMake to generate a Makefile (if you are using Linux or UNIX) or a Visual Studio workspace (if you are using Microsoft Windows). The second source file to be created is an actual C++ program that will exercise some of the large number of classes available in ITK. The details of these files are described in the following section.

Once both files are in your directory you can run CMake in order to configure your project. Under UNIX/Linux, you can cd to your newly created binary directory and launch the terminal-based version of CMake by entering "ccmake ../HelloWorldITK" in the terminal. Note the "../HelloWorldITK" in the command line to indicate that the CMakeLists.txt file is up one directory and in HelloWorldITK. In CMake GUI which can be used under Microsoft Windows and UNIX/Linux, the source and binary directories will have to be specified prior to the configuration and build file generation process.

Both the terminal-based and GUI versions of CMake will require you to specify the directory where ITK was built in the CMake variable ITK_DIR. The ITK binary directory will contain a file named ITKConfig.cmake generated during ITK configuration process with CMake. From this file, CMake will recover all information required to configure your new ITK project.

After generating the build files, on UNIX/Linux systems the project can be compiled by typing make in the terminal provided the current directory is set to the project's binary directory. In Visual Studio on Microsoft Windows the project can be built by loading the workspace named HelloWorldITK.sln from the binary directory specified in the CMake GUI and selecting "Build Solution" from the "Build" menu or by right-clicking on the ALL_BUILD target in the Solution Explorer pane and selecting the "Build" context menu item.

The resulting executable, which will be called HelloWorld, can be executed on the command line. If on Microsoft Windows, please note that double-clicking on the icon of the executable will quickly

[2] http://www.cmake.org/cmake/help/cmake_tutorial.html

launch a command line window, run the executable and close the window right away, not giving you time to see the output. It is therefore preferable to run the executable from the DOS command line by starting the `cmd.exe` shell first.

2.2.1 Hello World!

This section provides and explains the contents of the two files which need to be created for your new project. These two files can be found in the `ITK/Examples/Installation` directory.

The `CMakeLists.txt` file contains the following lines:

```
project(HelloWorld)

find_package(ITK REQUIRED)
include(${ITK_USE_FILE})

add_executable(HelloWorld HelloWorld.cxx)

target_link_libraries(HelloWorld ${ITK_LIBRARIES})
```

The first line defines the name of your project as it appears in Visual Studio or Eclipse; this line will have no effect with UNIX/Linux Makefiles. The second line loads a CMake file with a predefined strategy for finding ITK. If the strategy for finding ITK fails, CMake will report an error which can be corrected by providing the location of the directory where ITK was compiled or installed on your system. In this case the path to the ITK's binary/installation directory needs to be specified as the value of the `ITK_DIR` CMake variable. The line `include(${USE_ITK_FILE})` loads the `UseITK.cmake` file which contains the configuration information about the specified ITK build. The line starting with `add_executable` call defines as its first argument the name of the executable that will be produced as result of this project. The remaining argument(s) of `add_executable` are the names of the source files to be compiled. Finally, the `target_link_libraries` call specifies which ITK libraries will be linked against this project. Further details on creating and configuring CMake projects can be found in the CMake tutorial[3] and CMake online documentation[4].

The source code for this section can be found in the file `HelloWorld.cxx`.

The following code is an implementation of a small ITK program. It tests including header files and linking with ITK libraries.

[3] http://www.cmake.org/cmake/help/cmake_tutorial.html
[4] http://www.cmake.org/cmake/help/documentation.html

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

int main()
{
  typedef itk::Image< unsigned short, 3 > ImageType;

  ImageType::Pointer image = ImageType::New();

  std::cout << "ITK Hello World !" << std::endl;

  return 0;
}
```

This code instantiates a $3D$ image[5] whose pixels are represented with type `unsigned short`. The image is then constructed and assigned to a `itk::SmartPointer`. Although later in the text we will discuss `SmartPointer`s in detail, for now think of it as a handle on an instance of an object (see section 3.2.4 for more information). The `itk::Image` class will be described in Section 4.1.

By this point you have successfully configured and compiled ITK, and created your first simple program! If you have experienced any difficulties while following the instructions provided in this section, please join the community mailing list (see Section 1.5 on page 8) and post questions there.

[5] Also known as a *volume*.

Part II

Architecture

CHAPTER

THREE

SYSTEM OVERVIEW

The purpose of this chapter is to provide you with an overview of the *Insight Toolkit* system. We recommend that you read this chapter to gain an appreciation for the breadth and area of application of ITK.

3.1 System Organization

The Insight Toolkit consists of several subsystems. A brief description of these subsystems follows. Later sections in this chapter—and in some cases additional chapters—cover these concepts in more detail.

Essential System Concepts. Like any software system, ITK is built around some core design concepts. Some of the more important concepts include generic programming, smart pointers for memory management, object factories for adaptable object instantiation, event management using the command/observer design paradigm, and multithreading support.

Numerics. ITK uses VXL's VNL numerics libraries. These are easy-to-use C++ wrappers around the Netlib Fortran numerical analysis routines [1].

Data Representation and Access. Two principal classes are used to represent data: the `itk::Image` and `itk::Mesh` classes. In addition, various types of iterators and containers are used to hold and traverse the data. Other important but less popular classes are also used to represent data such as `itk::Histogram` and `itk::SpatialObject`.

Data Processing Pipeline. The data representation classes (known as *data objects*) are operated on by *filters* that in turn may be organized into data flow *pipelines*. These pipelines maintain state and therefore execute only when necessary. They also support multithreading, and are streaming capable (i.e., can operate on pieces of data to minimize the memory footprint).

[1] http://www.netlib.org

IO Framework. Associated with the data processing pipeline are *sources*, filters that initiate the pipeline, and *mappers*, filters that terminate the pipeline. The standard examples of sources and mappers are *readers* and *writers* respectively. Readers input data (typically from a file), and writers output data from the pipeline.

Spatial Objects. Geometric shapes are represented in ITK using the spatial object hierarchy. These classes are intended to support modeling of anatomical structures. Using a common basic interface, the spatial objects are capable of representing regions of space in a variety of different ways. For example: mesh structures, image masks, and implicit equations may be used as the underlying representation scheme. Spatial objects are a natural data structure for communicating the results of segmentation methods and for introducing anatomical priors in both segmentation and registration methods.

Registration Framework. A flexible framework for registration supports four different types of registration: image registration, multiresolution registration, PDE-based registration, and FEM (finite element method) registration.

FEM Framework. ITK includes a subsystem for solving general FEM problems, in particular non-rigid registration. The FEM package includes mesh definition (nodes and elements), loads, and boundary conditions.

Level Set Framework. The level set framework is a set of classes for creating filters to solve partial differential equations on images using an iterative, finite difference update scheme. The level set framework consists of finite difference solvers including a sparse level set solver, a generic level set segmentation filter, and several specific subclasses including threshold, Canny, and Laplacian based methods.

Wrapping. ITK uses a unique, powerful system for producing interfaces (i.e., "wrappers") to interpreted languages such as Python. The GCC-XML[2] tool is used to produce an XML description of arbitrarily complex C++ code. An interface generator script is then used to transform the XML description into wrappers using the SWIG[3] package.

3.2 Essential System Concepts

This section describes some of the core concepts and implementation features found in ITK.

3.2.1 Generic Programming

Generic programming is a method of organizing libraries consisting of generic—or reusable—software components [8]. The idea is to make software that is capable of "plugging together" in

[2] http://gccxml.org
[3] http://www.swig.org/

an efficient, adaptable manner. The essential ideas of generic programming are *containers* to hold data, *iterators* to access the data, and *generic algorithms* that use containers and iterators to create efficient, fundamental algorithms such as sorting. Generic programming is implemented in C++ with the *template* programming mechanism and the use of the STL Standard Template Library [1].

C++ templating is a programming technique allowing users to write software in terms of one or more unknown types `T`. To create executable code, the user of the software must specify all types `T` (known as *template instantiation*) and successfully process the code with the compiler. The `T` may be a native type such as `float` or `int`, or `T` may be a user-defined type (e.g., a `class`). At compile-time, the compiler makes sure that the templated types are compatible with the instantiated code and that the types are supported by the necessary methods and operators.

ITK uses the techniques of generic programming in its implementation. The advantage of this approach is that an almost unlimited variety of data types are supported simply by defining the appropriate template types. For example, in ITK it is possible to create images consisting of almost any type of pixel. In addition, the type resolution is performed at compile time, so the compiler can optimize the code to deliver maximal performance. The disadvantage of generic programming is that the analysis performed at compile time increases the time to build an application. Also, the increased complexity may produce difficult to decipher error messages due to even the simplest syntax errors. For those unfamiliar with templated code and generic programming, we recommend the two books cited above.

3.2.2 Include Files and Class Definitions

In ITK, classes are defined by a maximum of two files: a header file (`.h`) and an implementation file (`.cxx`) if defining a non-templated class, and a `.hxx` file if defining a templated class. The header files contain class declarations and formatted comments that are used by the Doxygen documentation system to automatically produce HTML manual pages.

In addition to class headers, there are a few other important header files.

`itkMacro.h` is found in the `Modules/Core/Common/include` directory and defines standard system-wide macros (such as `Set/Get`, constants, and other parameters).

`itkNumericTraits.h` is found in the `Modules/Core/Common/include` directory and defines numeric characteristics for native types such as its maximum and minimum possible values.

3.2.3 Object Factories

Most classes in ITK are instantiated through an *object factory* mechanism. That is, rather than using the standard C++ class constructor and destructor, instances of an ITK class are created with the static class `New()` method. In fact, the constructor and destructor are `protected:` so it is generally not possible to construct an ITK instance on the stack. (Note: this behavior pertains to classes that are derived from `itk::LightObject`. In some cases the need for speed or reduced memory

footprint dictates that a class is not derived from LightObject. In this case instances may be created on the stack. An example of such a class is the `itk::EventObject`.)

The object factory enables users to control run-time instantiation of classes by registering one or more factories with `itk::ObjectFactoryBase`. These registered factories support the method `CreateInstance(classname)` which takes as input the name of a class to create. The factory can choose to create the class based on a number of factors including the computer system configuration and environment variables. For example, a particular application may wish to deploy its own class implemented using specialized image processing hardware (i.e., to realize a performance gain). By using the object factory mechanism, it is possible to replace the creation of a particular ITK filter at run-time with such a custom class. (Of course, the class must provide the exact same API as the one it is replacing.). For this, the user compiles his class (using the same compiler, build options, etc.) and inserts the object code into a shared library or DLL. The library is then placed in a directory referred to by the `ITK_AUTOLOAD_PATH` environment variable. On instantiation, the object factory will locate the library, determine that it can create a class of a particular name with the factory, and use the factory to create the instance. (Note: if the `CreateInstance()` method cannot find a factory that can create the named class, then the instantiation of the class falls back to the usual constructor.)

In practice, object factories are used mainly (and generally transparently) by the ITK input/output (IO) classes. For most users the greatest impact is on the use of the `New()` method to create a class. Generally the `New()` method is declared and implemented via the macro `itkNewMacro()` found in `Modules/Core/Common/include/itkMacro.h`.

3.2.4 Smart Pointers and Memory Management

By their nature, object-oriented systems represent and operate on data through a variety of object types, or classes. When a particular class is instantiated, memory allocation occurs so that the instance can store data attribute values and method pointers (i.e., the vtable). This object may then be referenced by other classes or data structures during normal operation of the program. Typically, during program execution, all references to the instance may disappear at which point the instance must be deleted to recover memory resources. Knowing when to delete an instance, however, is difficult. Deleting the instance too soon results in program crashes; deleting it too late causes memory leaks (or excessive memory consumption). This process of allocating and releasing memory is known as memory management.

In ITK, memory management is implemented through reference counting. This compares to another popular approach—garbage collection—used by many systems, including Java. In reference counting, a count of the number of references to each instance is kept. When the reference goes to zero, the object destroys itself. In garbage collection, a background process sweeps the system identifying instances no longer referenced in the system and deletes them. The problem with garbage collection is that the actual point in time at which memory is deleted is variable. This is unacceptable when an object size may be gigantic (think of a large 3D volume gigabytes in size). Reference counting deletes memory immediately (once all references to an object disappear).

Reference counting is implemented through a `Register()`/`Delete()` member function interface.

All instances of an ITK object have a `Register()` method invoked on them by any other object that references them. The `Register()` method increments the instances' reference count. When the reference to the instance disappears, a `Delete()` method is invoked on the instance that decrements the reference count—this is equivalent to an `UnRegister()` method. When the reference count returns to zero, the instance is destroyed.

This protocol is greatly simplified by using a helper class called a `itk::SmartPointer`. The smart pointer acts like a regular pointer (e.g. supports operators `->` and `*`) but automagically performs a `Register()` when referring to an instance, and an `UnRegister()` when it no longer points to the instance. Unlike most other instances in ITK, SmartPointers can be allocated on the program stack, and are automatically deleted when the scope that the SmartPointer was created in is closed. As a result, you should *rarely if ever call Register() or Delete()* in ITK. For example:

```
MyRegistrationFunction()
  { /* <----- Start of scope */

  // here an interpolator is created and associated to the
  // "interp" SmartPointer.
  InterpolatorType::Pointer interp = InterpolatorType::New();

  } /* <------ End of scope */
```

In this example, reference counted objects are created (with the `New()` method) with a reference count of one. Assignment to the SmartPointer `interp` does not change the reference count. At the end of scope, `interp` is destroyed, the reference count of the actual interpolator object (referred to by `interp`) is decremented, and if it reaches zero, then the interpolator is also destroyed.

Note that in ITK SmartPointers are always used to refer to instances of classes derived from `itk::LightObject`. Method invocations and function calls often return "real" pointers to instances, but they are immediately assigned to a SmartPointer. Raw pointers are used for non-LightObject classes when the need for speed and/or memory demands a smaller, faster class. Raw pointers are preferred for multi-threaded sections of code.

3.2.5 Error Handling and Exceptions

In general, ITK uses exception handling to manage errors during program execution. Exception handling is a standard part of the C++ language and generally takes the form as illustrated below:

```
try
  {
  //...try executing some code here...
  }
catch ( itk::ExceptionObject & exp )
  {
  //...if an exception is thrown catch it here
  }
```

A particular class may throw an exception as demonstrated below (this code snippet is taken from

itk::ByteSwapper:

```
switch ( sizeof(T) )
  {
  //non-error cases go here followed by error case
  default:
    ByteSwapperError e(__FILE__, __LINE__);
    e.SetLocation("SwapBE");
    e.SetDescription("Cannot swap number of bytes requested");
    throw e;
  }
```

Note that itk::ByteSwapperError is a subclass of itk::ExceptionObject. In fact, all ITK exceptions derive from ExceptionObject. In this example a special constructor and C++ preprocessor variables __FILE__ and __LINE__ are used to instantiate the exception object and provide additional information to the user. You can choose to catch a particular exception and hence a specific ITK error, or you can trap *any* ITK exception by catching ExceptionObject.

3.2.6 Event Handling

Event handling in ITK is implemented using the Subject/Observer design pattern [3] (sometimes referred to as the Command/Observer design pattern). In this approach, objects indicate that they are watching for a particular event—invoked by a particular instance—by registering with the instance that they are watching. For example, filters in ITK periodically invoke the itk::ProgressEvent. Objects that have registered their interest in this event are notified when the event occurs. The notification occurs via an invocation of a command (i.e., function callback, method invocation, etc.) that is specified during the registration process. (Note that events in ITK are subclasses of EventObject; look in itkEventObject.h to determine which events are available.)

To recap using an example: various objects in ITK will invoke specific events as they execute (from ProcessObject):

```
this->InvokeEvent( ProgressEvent() );
```

To watch for such an event, registration is required that associates a command (e.g., callback function) with the event: Object::AddObserver() method:

```
unsigned long progressTag =
  filter->AddObserver(ProgressEvent(), itk::Command*);
```

When the event occurs, all registered observers are notified via invocation of the associated Command::Execute() method. Note that several subclasses of Command are available supporting const and non-const member functions as well as C-style functions. (Look in Modules/Core/Common/include/itkCommand.h to find pre-defined subclasses of Command. If nothing suitable is found, derivation is another possibility.)

3.2.7 Multi-Threading

Multithreading is handled in ITK through a high-level design abstraction. This approach provides portable multithreading and hides the complexity of differing thread implementations on the many systems supported by ITK. For example, the class `itk::MultiThreader` provides support for multithreaded execution using `sproc()` on an SGI, or `pthread_create` on any platform supporting POSIX threads.

Multithreading is typically employed by an algorithm during its execution phase. MultiThreader can be used to execute a single method on multiple threads, or to specify a method per thread. For example, in the class `itk::ImageSource` (a superclass for most image processing filters) the `GenerateData()` method uses the following methods:

```
multiThreader->SetNumberOfThreads(int);
multiThreader->SetSingleMethod(ThreadFunctionType, void* data);
multiThreader->SingleMethodExecute();
```

In this example each thread invokes the same method. The multithreaded filter takes care to divide the image into different regions that do not overlap for write operations.

The general philosophy in ITK regarding thread safety is that accessing different instances of a class (and its methods) is a thread-safe operation. Invoking methods on the same instance in different threads is to be avoided.

3.3 Numerics

ITK uses the VNL numerics library to provide resources for numerical programming combining the ease of use of packages like Mathematica and Matlab with the speed of C and the elegance of C++. It provides a C++ interface to the high-quality Fortran routines made available in the public domain by numerical analysis researchers. ITK extends the functionality of VNL by including interface classes between VNL and ITK proper.

The VNL numerics library includes classes for:

Matrices and vectors. Standard matrix and vector support and operations on these types.

Specialized matrix and vector classes. Several special matrix and vector classes with special numerical properties are available. Class `vnl_diagonal_matrix` provides a fast and convenient diagonal matrix, while fixed size matrices and vectors allow ”fast-as-C” computations (see `vnl_matrix_fixed<T,n,m>` and example subclasses `vnl_double_3x3` and `vnl_double_3`).

Matrix decompositions. Classes `vnl_svd<T>`, `vnl_symmetric_eigensystem<T>`, and `vnl_generalized_eigensystem`.

Real polynomials. Class `vnl_real_polynomial` stores the coefficients of a real polynomial, and provides methods of evaluation of the polynomial at any x, while class `vnl_rpoly_roots` provides a root finder.

Optimization. Classes `vnl_levenberg_marquardt`, `vnl_amoeba`, `vnl_conjugate_gradient`, `vnl_lbfgs` allow optimization of user-supplied functions either with or without user-supplied derivatives.

Standardized functions and constants. Class `vnl_math` defines constants (pi, e, eps...) and simple functions (sqr, abs, rnd...). Class `numeric_limits` is from the ISO standard document, and provides a way to access basic limits of a type. For example `numeric_limits<short>::max()` returns the maximum value of a short.

Most VNL routines are implemented as wrappers around the high-quality Fortran routines that have been developed by the numerical analysis community over the last forty years and placed in the public domain. The central repository for these programs is the "netlib" server [4]. The National Institute of Standards and Technology (NIST) provides an excellent search interface to this repository in its *Guide to Available Mathematical Software (GAMS)*[5], both as a decision tree and a text search.

ITK also provides additional numerics functionality. A suite of optimizers, that use VNL under the hood and integrate with the registration framework are available. A large collection of statistics functions—not available from VNL—are also provided in the `Insight/Numerics/Statistics` directory. In addition, a complete finite element (FEM) package is available, primarily to support the deformable registration in ITK.

3.4 Data Representation

There are two principle types of data represented in ITK: images and meshes. This functionality is implemented in the classes `itk::Image` and `itk::Mesh`, both of which are subclasses of `itk::DataObject`. In ITK, data objects are classes that are meant to be passed around the system and may participate in data flow pipelines (see Section 3.5 on page 31 for more information).

`itk::Image` represents an n-dimensional, regular sampling of data. The sampling direction is parallel to direction matrix axes, and the origin of the sampling, inter-pixel spacing, and the number of samples in each direction (i.e., image dimension) can be specified. The sample, or pixel, type in ITK is arbitrary—a template parameter `TPixel` specifies the type upon template instantiation. (The dimensionality of the image must also be specified when the image class is instantiated.) The key is that the pixel type must support certain operations (for example, addition or difference) if the code is to compile in all cases (for example, to be processed by a particular filter that uses these operations). In practice, most applications will use a C++ simple type (e.g., `int`, `float`) or a pre-defined pixel type and will rarely create a new type of pixel class.

[4] http://www.netlib.org/
[5] http://gams.nist.gov

One of the important ITK concepts regarding images is that rectangular, continuous pieces of the image are known as *regions*. Regions are used to specify which part of an image to process, for example in multithreading, or which part to hold in memory. In ITK there are three common types of regions:

1. `LargestPossibleRegion`—the image in its entirety.
2. `BufferedRegion`—the portion of the image retained in memory.
3. `RequestedRegion`—the portion of the region requested by a filter or other class when operating on the image.

The `itk::Mesh` class represents an n-dimensional, unstructured grid. The topology of the mesh is represented by a set of *cells* defined by a type and connectivity list; the connectivity list in turn refers to points. The geometry of the mesh is defined by the n-dimensional points in combination with associated cell interpolation functions. `Mesh` is designed as an adaptive representational structure that changes depending on the operations performed on it. At a minimum, points and cells are required in order to represent a mesh; but it is possible to add additional topological information. For example, links from the points to the cells that use each point can be added; this provides implicit neighborhood information assuming the implied topology is the desired one. It is also possible to specify boundary cells explicitly, to indicate different connectivity from the implied neighborhood relationships, or to store information on the boundaries of cells.

The mesh is defined in terms of three template parameters: 1) a pixel type associated with the points, cells, and cell boundaries; 2) the dimension of the points (which in turn limits the maximum dimension of the cells); and 3) a "mesh traits" template parameter that specifies the types of the containers and identifiers used to access the points, cells, and/or boundaries. By using the mesh traits carefully, it is possible to create meshes better suited for editing, or those better suited for "read-only" operations, allowing a trade-off between representation flexibility, memory, and speed.

Mesh is a subclass of `itk::PointSet`. The PointSet class can be used to represent point clouds or randomly distributed landmarks, etc. The PointSet class has no associated topology.

3.5 Data Processing Pipeline

While data objects (e.g., images and meshes) are used to represent data, *process objects* are classes that operate on data objects and may produce new data objects. Process objects are classed as *sources*, *filter objects*, or *mappers*. Sources (such as readers) produce data, filter objects take in data and process it to produce new data, and mappers accept data for output either to a file or some other system. Sometimes the term *filter* is used broadly to refer to all three types.

The data processing pipeline ties together data objects (e.g., images and meshes) and process objects. The pipeline supports an automatic updating mechanism that causes a filter to execute if and only if its input or its internal state changes. Further, the data pipeline supports *streaming*, the ability

to automatically break data into smaller pieces, process the pieces one by one, and reassemble the processed data into a final result.

Typically data objects and process objects are connected together using the `SetInput()` and `GetOutput()` methods as follows:

```
typedef itk::Image<float,2> FloatImage2DType;

itk::RandomImageSource<FloatImage2DType>::Pointer random;
random = itk::RandomImageSource<FloatImage2DType>::New();
random->SetMin(0.0);
random->SetMax(1.0);

itk::ShrinkImageFilter<FloatImage2DType,FloatImage2DType>::Pointer shrink;
shrink = itk::ShrinkImageFilter<FloatImage2DType,FloatImage2DType>::New();
shrink->SetInput(random->GetOutput());
shrink->SetShrinkFactors(2);

itk::ImageFileWriter<FloatImage2DType>::Pointer writer;
writer = itk::ImageFileWriter<FloatImage2DType>::New();
writer->SetInput (shrink->GetOutput());
writer->SetFileName( "test.raw" );
writer->Update();
```

In this example the source object `itk::RandomImageSource` is connected to the `itk::ShrinkImageFilter`, and the shrink filter is connected to the mapper `itk::ImageFileWriter`. When the `Update()` method is invoked on the writer, the data processing pipeline causes each of these filters in order, culminating in writing the final data to a file on disk.

3.6 Spatial Objects

The ITK spatial object framework supports the philosophy that the task of image segmentation and registration is actually the task of object processing. The image is but one medium for representing objects of interest, and much processing and data analysis can and should occur at the object level and not based on the medium used to represent the object.

ITK spatial objects provide a common interface for accessing the physical location and geometric properties of and the relationship between objects in a scene that is independent of the form used to represent those objects. That is, the internal representation maintained by a spatial object may be a list of points internal to an object, the surface mesh of the object, a continuous or parametric representation of the object's internal points or surfaces, and so forth.

The capabilities provided by the spatial objects framework supports their use in object segmentation, registration, surface/volume rendering, and other display and analysis functions. The spatial object framework extends the concept of a "scene graph" that is common to computer rendering packages so as to support these new functions. With the spatial objects framework you can:

1. Specify a spatial object's parent and children objects. In this way, a liver may contain vessels and those vessels can be organized in a tree structure.
2. Query if a physical point is inside an object or (optionally) any of its children.
3. Request the value and derivatives, at a physical point, of an associated intensity function, as specified by an object or (optionally) its children.
4. Specify the coordinate transformation that maps a parent object's coordinate system into a child object's coordinate system.
5. Compute the bounding box of a spatial object and (optionally) its children.
6. Query the resolution at which the object was originally computed. For example, you can query the resolution (i.e., voxel spacing) of the image used to generate a particular instance of a `itk::BlobSpatialObject`.

Currently implemented types of spatial objects include: Blob, Ellipse, Group, Image, Line, Surface, and Tube. The `itk::Scene` object is used to hold a list of spatial objects that may in turn have children. Each spatial object can be assigned a color property. Each spatial object type has its own capabilities. For example, the `itk::TubeSpatialObject` indicates the point where it is connected with its parent tube.

There are a limited number of spatial objects in ITK, but their number is growing and their potential is huge. Using the nominal spatial object capabilities, methods such as marching cubes or mutual information registration can be applied to objects regardless of their internal representation. By having a common API, the same method can be used to register a parametric representation of a hearth with an individual's CT data or to register two segmentations of a liver.

3.7 Wrapping

While the core of ITK is implemented in C++, Python bindings can be automatically generated and ITK programs can be created using Python. This capability is under active development. This brief description will give an idea of what is possible and where to look for those who are interested in this facility.

The wrapping process in ITK is quite complex due to the use of generic programming (i.e., extensive use of C++ templates). Systems like VTK that use their own wrapping facility are non-templated and customized to the coding methodology found in the system. Even systems like SWIG that are designed for general wrapper generation have difficulty with ITK code because general C++ is difficult to parse. As a result, the ITK wrapper generator uses a combination of tools to produce language bindings.

1. gccxml is a modified version of the GNU compiler gcc that produces an XML description of an input C++ program.

2. The `igenerator.py` script in the ITK source tree processes XML information produced by gccxml and generates standard input files (`*.i` files) to the next tool (SWIG), indicating what is to be wrapped and how to wrap it.

3. SWIG produces the appropriate Python bindings.

To learn more about the wrapping process, please see the `Wrapping` directory. The wrapping process is orchestrated by a number of CMake macros found in the `Wrapping` directory. The result of the wrapping process is a set of shared libraries (.so in Linux or .dlls on Windows) that can be used by interpreted languages.

There is almost a direct translation from C++, with the differences being the particular syntactical requirements of each language. For example, to rescale an image using the Python wrapping:

```
inputImage = sys.argv[1]
outputImage = sys.argv[2]
radiusValue = int(sys.argv[3])

PixelType = itk.UC
Dimension = 2
ImageType = itk.Image[PixelType, Dimension]

ReaderType = itk.ImageFileReader[ImageType]
reader = ReaderType.New()
reader.SetFileName(inputImage)

StructuringElementType = itk.FlatStructuringElement[Dimension]
structuringElement = StructuringElementType.Ball(radiusValue)

DilateFilterType = itk.BinaryDilateImageFilter[
    ImageType, ImageType, StructuringElementType]
dilateFilter = DilateFilterType.New()
dilateFilter.SetInput(reader.GetOutput())
dilateFilter.SetKernel(structuringElement)
```

The same code in C++ would appear as follows:

```
const char * inputImage = argv[1];
const char * outputImage = argv[2];
const unsigned int radiusValue = atoi( argv[3] );

typedef unsigned char PixelType;
const unsigned int Dimension = 2;

typedef itk::Image< PixelType, Dimension >    ImageType;
typedef itk::ImageFileReader< ImageType >     ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( inputImage );

typedef itk::FlatStructuringElement< Dimension >
  StructuringElementType;
StructuringElementType::RadiusType radius;
radius.Fill( radiusValue );
StructuringElementType structuringElement =
  StructuringElementType::Ball( radius );

typedef itk::BinaryDilateImageFilter< ImageType, ImageType,
  StructuringElementType > BinaryDilateImageFilterType;

BinaryDilateImageFilterType::Pointer dilateFilter =
  BinaryDilateImageFilterType::New();
dilateFilter->SetInput( reader->GetOutput() );
dilateFilter->SetKernel( structuringElement );
```

This example demonstrates an important difference between C++ and a wrapped language such as Python. Templated classes must be instantiated prior to wrapping. That is, the template parameters must be specified as part of the wrapping process. In the example above, the `ImageFileReader[ImageType]` indicates that this class, implementing an image source, has been instantiated using an input and output image type of two-dimensional unsigned char values (e.g., `UC`). Typically just a few common types are selected for the wrapping process to avoid an explosion of types and hence, library size. To add a new type requires rerunning the wrapping process to produce new libraries. Some high-level options for these types, such as common pixels types and image dimensions, are specified during CMake configuration. The types of specific classes that should be instantiated, based on these basic options, are defined by the `*.wrap` files in the `wrapping` directory of a module.

The advantage of interpreted languages is that they do not require the lengthy compile/link cycle of a compiled language like C++. Moreover, they typically come with a suite of packages that provide useful functionalities. For example, the Python ecosystem provides a variety of powerful tools for creating sophisticated user interfaces. In the future it is likely that more applications and tests will be implemented in the various interpreted languages supported by ITK. Other languages like Java, Ruby, Tcl could also be wrapped in the future.

3.7.1 Python Setup

In order to access the Python interface of ITK, make sure to compile with the CMake `ITK_WRAP_PYTHON` option. In addition, choose which pixel types and dimensions to build into the wrapped interface. Supported pixel types are represented in the CMake configuration as variables named `ITK_WRAP_<pixel type>`. Supported image dimensions are enumerated in the semicolon-delimited list `ITK_WRAP_DIMS`, the default value of which is `2;3` indicating support for 2- and 3-dimensional images.

After configuration, check to make sure that the values of the following variables are set correctly:

- `PYTHON_INCLUDE_DIR`
- `PYTHON_LIBRARY`
- `PYTHON_EXECUTABLE`

particularly if there are multiple Python installations on the system.

The environment for access to the `itk` Python module is best configured using the Python virtualenv tool, which provides an isolated working copy of Python without interfering with Python installed at the system level. Once the virtualenv package is installed on your Linux system, create the virtual environment within the directory ITK was built in. Create a symbolic link to the `WrapITK.pth` file within the `lib/python2.7/site-packages` directory of Python's virtual install directory.

```
virtualenv --system-site-packages wrapitk-venv
cd wrapitk-venv/lib/python2.7/site-packages
ln -s /path/to/ITK-Wrapped/Wrapping/Generators/Python/WrapITK.pth
cd ../../../../wrapitk-venv/bin
./python /usr/bin/ipython
import itk
```

CHAPTER

FOUR

DATA REPRESENTATION

This chapter introduces the basic classes responsible for representing data in ITK. The most common classes are the `itk::Image`, the `itk::Mesh` and the `itk::PointSet`.

4.1 Image

The `itk::Image` class follows the spirit of Generic Programming, where types are separated from the algorithmic behavior of the class. ITK supports images with any pixel type and any spatial dimension.

4.1.1 Creating an Image

The source code for this section can be found in the file
`Image1.cxx`.

This example illustrates how to manually construct an `itk::Image` class. The following is the minimal code needed to instantiate, declare and create the image class.

First, the header file of the Image class must be included.

```
#include "itkImage.h"
```

Then we must decide with what type to represent the pixels and what the dimension of the image will be. With these two parameters we can instantiate the image class. Here we create a 3D image with `unsigned short` pixel data.

```
  typedef itk::Image< unsigned short, 3 > ImageType;
```

The image can then be created by invoking the `New()` operator from the corresponding image type

and assigning the result to a itk::SmartPointer.

```
ImageType::Pointer image = ImageType::New();
```

In ITK, images exist in combination with one or more *regions*. A region is a subset of the image and indicates a portion of the image that may be processed by other classes in the system. One of the most common regions is the *LargestPossibleRegion*, which defines the image in its entirety. Other important regions found in ITK are the *BufferedRegion*, which is the portion of the image actually maintained in memory, and the *RequestedRegion*, which is the region requested by a filter or other class when operating on the image.

In ITK, manually creating an image requires that the image is instantiated as previously shown, and that regions describing the image are then associated with it.

A region is defined by two classes: the itk::Index and itk::Size classes. The origin of the region within the image is defined by the Index. The extent, or size, of the region is defined by the Size. When an image is created manually, the user is responsible for defining the image size and the index at which the image grid starts. These two parameters make it possible to process selected regions.

The Index is represented by a n-dimensional array where each component is an integer indicating—in topological image coordinates—the initial pixel of the image.

```
ImageType::IndexType start;

start[0] =   0;  // first index on X
start[1] =   0;  // first index on Y
start[2] =   0;  // first index on Z
```

The region size is represented by an array of the same dimension of the image (using the itk::Size class). The components of the array are unsigned integers indicating the extent in pixels of the image along every dimension.

```
ImageType::SizeType  size;

size[0]  = 200;  // size along X
size[1]  = 200;  // size along Y
size[2]  = 200;  // size along Z
```

Having defined the starting index and the image size, these two parameters are used to create an itk::ImageRegion object which basically encapsulates both concepts. The region is initialized with the starting index and size of the image.

```
ImageType::RegionType region;

region.SetSize( size );
region.SetIndex( start );
```

Finally, the region is passed to the Image object in order to define its extent and origin. The

SetRegions method sets the *LargestPossibleRegion*, *BufferedRegion*, and *RequestedRegion* simultaneously. Note that none of the operations performed to this point have allocated memory for the image pixel data. It is necessary to invoke the Allocate() method to do this. Allocate does not require any arguments since all the information needed for memory allocation has already been provided by the region.

```
image->SetRegions( region );
image->Allocate();
```

In practice it is rare to allocate and initialize an image directly. Images are typically read from a source, such a file or data acquisition hardware. The following example illustrates how an image can be read from a file.

4.1.2 Reading an Image from a File

The source code for this section can be found in the file
Image2.cxx.

The first thing required to read an image from a file is to include the header file of the itk::ImageFileReader class.

```
#include "itkImageFileReader.h"
```

Then, the image type should be defined by specifying the type used to represent pixels and the dimensions of the image.

```
typedef unsigned char          PixelType;
const unsigned int            Dimension = 3;

typedef itk::Image< PixelType, Dimension >   ImageType;
```

Using the image type, it is now possible to instantiate the image reader class. The image type is used as a template parameter to define how the data will be represented once it is loaded into memory. This type does not have to correspond exactly to the type stored in the file. However, a conversion based on C-style type casting is used, so the type chosen to represent the data on disk must be sufficient to characterize it accurately. Readers do not apply any transformation to the pixel data other than casting from the pixel type of the file to the pixel type of the ImageFileReader. The following illustrates a typical instantiation of the ImageFileReader type.

```
typedef itk::ImageFileReader< ImageType >  ReaderType;
```

The reader type can now be used to create one reader object. A itk::SmartPointer (defined by the ::Pointer notation) is used to receive the reference to the newly created reader. The New() method is invoked to create an instance of the image reader.

```
ReaderType::Pointer reader = ReaderType::New();
```

The minimal information required by the reader is the filename of the image to be loaded in memory. This is provided through the `SetFileName()` method. The file format here is inferred from the filename extension. The user may also explicitly specify the data format using the `itk::ImageIOBase` class (a list of possibilities can be found in the inheritance diagram of this class.).

```
const char * filename = argv[1];
reader->SetFileName( filename );
```

Reader objects are referred to as pipeline source objects; they respond to pipeline update requests and initiate the data flow in the pipeline. The pipeline update mechanism ensures that the reader only executes when a data request is made to the reader and the reader has not read any data. In the current example we explicitly invoke the `Update()` method because the output of the reader is not connected to other filters. In normal application the reader's output is connected to the input of an image filter and the update invocation on the filter triggers an update of the reader. The following line illustrates how an explicit update is invoked on the reader.

```
reader->Update();
```

Access to the newly read image can be gained by calling the `GetOutput()` method on the reader. This method can also be called before the update request is sent to the reader. The reference to the image will be valid even though the image will be empty until the reader actually executes.

```
ImageType::Pointer image = reader->GetOutput();
```

Any attempt to access image data before the reader executes will yield an image with no pixel data. It is likely that a program crash will result since the image will not have been properly initialized.

4.1.3 Accessing Pixel Data

The source code for this section can be found in the file `Image3.cxx`.

This example illustrates the use of the `SetPixel()` and `GetPixel()` methods. These two methods provide direct access to the pixel data contained in the image. Note that these two methods are relatively slow and should not be used in situations where high-performance access is required. Image iterators are the appropriate mechanism to efficiently access image pixel data. (See Chapter 6 on page 139 for information about image iterators.)

The individual position of a pixel inside the image is identified by a unique index. An index is an array of integers that defines the position of the pixel along each dimension of the image. The `IndexType` is automatically defined by the image and can be accessed using the scope operator `itk::Index`. The length of the array will match the dimensions of the associated image.

The following code illustrates the declaration of an index variable and the assignment of values to each of its components. Please note that no SmartPointer is used to access the `Index`. This is because `Index` is a lightweight object that is not intended to be shared between objects. It is more efficient to produce multiple copies of these small objects than to share them using the SmartPointer mechanism.

The following lines declare an instance of the index type and initialize its content in order to associate it with a pixel position in the image.

```
const ImageType::IndexType pixelIndex = {{27,29,37}}; // Position of {X,Y,Z}
```

Having defined a pixel position with an index, it is then possible to access the content of the pixel in the image. The `GetPixel()` method allows us to get the value of the pixels.

```
ImageType::PixelType   pixelValue = image->GetPixel( pixelIndex );
```

The `SetPixel()` method allows us to set the value of the pixel.

```
image->SetPixel(   pixelIndex,   pixelValue+1  );
```

Please note that `GetPixel()` returns the pixel value using copy and not reference semantics. Hence, the method cannot be used to modify image data values.

Remember that both `SetPixel()` and `GetPixel()` are inefficient and should only be used for debugging or for supporting interactions like querying pixel values by clicking with the mouse.

4.1.4 Defining Origin and Spacing

The source code for this section can be found in the file `Image4.cxx`.

Even though ITK can be used to perform general image processing tasks, the primary purpose of the toolkit is the processing of medical image data. In that respect, additional information about the images is considered mandatory. In particular the information associated with the physical spacing between pixels and the position of the image in space with respect to some world coordinate system are extremely important.

Image origin, image voxel directions (i.e. orientation) and spacing are fundamental to many applications. Registration, for example, is performed in physical coordinates. Improperly defined spacing, direction, and origins will result in inconsistent results in such processes. Medical images with no spatial information should not be used for medical diagnosis, image analysis, feature extraction, assisted radiation therapy or image guided surgery. In other words, medical images lacking spatial information are not only useless but also hazardous.

Figure 4.1 illustrates the main geometrical concepts associated with the `itk::Image`. In this figure,

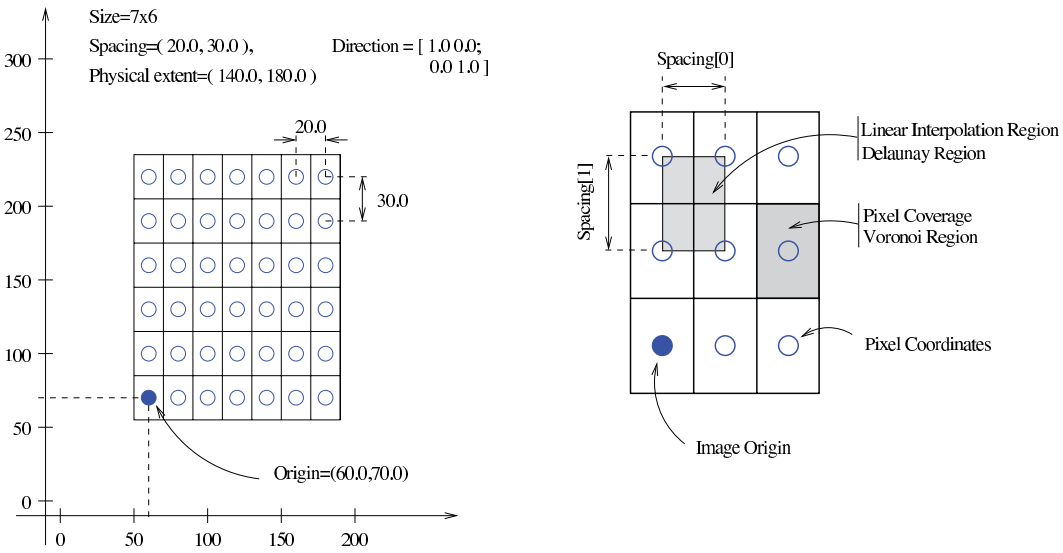

Figure 4.1: Geometrical concepts associated with the ITK image.

circles are used to represent the center of pixels. The value of the pixel is assumed to exist as a Dirac delta function located at the pixel center. Pixel spacing is measured between the pixel centers and can be different along each dimension. The image origin is associated with the coordinates of the first pixel in the image. For this simplified example, the voxel lattice is perfectly aligned with physical space orientation, and the image direction is therefore an identity mapping. If the voxel lattice samples were rotated with respect to physical space, then the image direction would contain a rotation matrix.

A *pixel* is considered to be the rectangular region surrounding the pixel center holding the data value. This can be viewed as the Voronoi region of the image grid, as illustrated in the right side of the figure. Linear interpolation of image values is performed inside the Delaunay region whose corners are pixel centers.

Image spacing is represented in a `FixedArray` whose size matches the dimension of the image. In order to manually set the spacing of the image, an array of the corresponding type must be created. The elements of the array should then be initialized with the spacing between the centers of adjacent pixels. The following code illustrates the methods available in the `itk::Image` class for dealing with spacing and origin.

```
ImageType::SpacingType spacing;

// Units (e.g., mm, inches, etc.) are defined by the application.
spacing[0] = 0.33; // spacing along X
spacing[1] = 0.33; // spacing along Y
spacing[2] = 1.20; // spacing along Z
```

The array can be assigned to the image using the `SetSpacing()` method.

```
image->SetSpacing( spacing );
```

The spacing information can be retrieved from an image by using the `GetSpacing()` method. This method returns a reference to a `FixedArray`. The returned object can then be used to read the contents of the array. Note the use of the `const` keyword to indicate that the array will not be modified.

```
const ImageType::SpacingType& sp = image->GetSpacing();

std::cout << "Spacing = ";
std::cout << sp[0] << ", " << sp[1] << ", " << sp[2] << std::endl;
```

The image origin is managed in a similar way to the spacing. A `Point` of the appropriate dimension must first be allocated. The coordinates of the origin can then be assigned to every component. These coordinates correspond to the position of the first pixel of the image with respect to an arbitrary reference system in physical space. It is the user's responsibility to make sure that multiple images used in the same application are using a consistent reference system. This is extremely important in image registration applications.

The following code illustrates the creation and assignment of a variable suitable for initializing the image origin.

```
// coordinates of the center of the first pixel in N-D
ImageType::PointType newOrigin;
newOrigin.Fill(0.0);
image->SetOrigin( newOrigin );
```

The origin can also be retrieved from an image by using the `GetOrigin()` method. This will return a reference to a `Point`. The reference can be used to read the contents of the array. Note again the use of the `const` keyword to indicate that the array contents will not be modified.

```
const ImageType::PointType & origin = image->GetOrigin();

std::cout << "Origin = ";
std::cout << origin[0] << ", "
          << origin[1] << ", "
          << origin[2] << std::endl;
```

The image direction matrix represents the orientation relationships between the image samples and physical space coordinate systems. The image direction matrix is an orthonormal matrix that describes the possible permutation of image index values and the rotational aspects that are needed to properly reconcile image index organization with physical space axis. The image directions is a *NxN* matrix where *N* is the dimension of the image. An identity image direction indicates that increasing values of the 1st, 2nd, 3rd index element corresponds to increasing values of the 1st, 2nd and 3rd physical space axis respectively, and that the voxel samples are perfectly aligned with the physical space axis.

The following code illustrates the creation and assignment of a variable suitable for initializing the image direction with an identity.

```
// coordinates of the center of the first pixel in N-D
ImageType::DirectionType direction;
direction.SetIdentity();
image->SetDirection( direction );
```

The direction can also be retrieved from an image by using the `GetDirection()` method. This will return a reference to a `Matrix`. The reference can be used to read the contents of the array. Note again the use of the `const` keyword to indicate that the matrix contents can not be modified.

```
const ImageType::DirectionType& direct = image->GetDirection();

std::cout << "Direction = " << std::endl;
std::cout << direct << std::endl;
```

Once the spacing, origin, and direction of the image samples have been initialized, the image will correctly map pixel indices to and from physical space coordinates. The following code illustrates how a point in physical space can be mapped into an image index for the purpose of reading the content of the closest pixel.

First, a `itk::Point` type must be declared. The point type is templated over the type used to represent coordinates and over the dimension of the space. In this particular case, the dimension of the point must match the dimension of the image.

```
typedef itk::Point< double, ImageType::ImageDimension > PointType;
```

The `itk::Point` class, like an `itk::Index`, is a relatively small and simple object. This means that no `itk::SmartPointer` is used here and the objects are simply declared as instances, like any other C++ class. Once the point is declared, its components can be accessed using traditional array notation. In particular, the `[]` operator is available. For efficiency reasons, no bounds checking is performed on the index used to access a particular point component. It is the user's responsibility to make sure that the index is in the range $\{0, Dimension - 1\}$.

```
PointType point;
point[0] = 1.45;    // x coordinate
point[1] = 7.21;    // y coordinate
point[2] = 9.28;    // z coordinate
```

The image will map the point to an index using the values of the current spacing and origin. An index object must be provided to receive the results of the mapping. The index object can be instantiated by using the `IndexType` defined in the image type.

```
ImageType::IndexType pixelIndex;
```

The `TransformPhysicalPointToIndex()` method of the image class will compute the pixel index closest to the point provided. The method checks for this index to be contained inside the current

buffered pixel data. The method returns a boolean indicating whether the resulting index falls inside the buffered region or not. The output index should not be used when the returned value of the method is `false`.

The following lines illustrate the point to index mapping and the subsequent use of the pixel index for accessing pixel data from the image.

```
const bool isInside =
  image->TransformPhysicalPointToIndex( point, pixelIndex );
if ( isInside )
  {
  ImageType::PixelType pixelValue = image->GetPixel( pixelIndex );
  pixelValue += 5;
  image->SetPixel( pixelIndex, pixelValue );
  }
```

Remember that `GetPixel()` and `SetPixel()` are very inefficient methods for accessing pixel data. Image iterators should be used when massive access to pixel data is required.

The following example illustrates the mathematical relationships between image index locations and its corresponding physical point representation for a given Image.

Let us imagine that a graphical user interface exists where the end user manually selects the voxel index location of the left eye in a volume with a mouse interface. We need to convert that index location to a physical location so that laser guided surgery can be accurately performed. The `TransformIndexToPhysicalPoint` method can be used for this.

```
const ImageType::IndexType LeftEyeIndex = GetIndexFromMouseClick();
ImageType::PointType LeftEyePoint;
image->TransformIndexToPhysicalPoint(LeftEyeIndex,LeftEyePoint);
```

For a given index I_{3X1}, the physical location P_{3X1} is calculated as following:

$$P_{3X1} = O_{3X1} + D_{3X3} * diag(S_{3X1})_{3x3} * I_{3X1} \tag{4.1}$$

where D is an orthonormal direction cosines matrix and S is the image spacing diagonal matrix.

In matlab syntax the conversions are:

```
% Non-identity Spacing and Direction
spacing=diag( [0.9375, 0.9375, 1.5] );
direction=[0.998189, 0.0569345, -0.0194113;
0.0194429, -7.38061e-08, 0.999811;
0.0569237, -0.998378, -0.00110704];
point = origin + direction * spacing * LeftEyeIndex
```

A corresponding mathematical expansion of the C/C++ code is:

```
typedef itk::Matrix<double, Dimension, Dimension> MatrixType;
MatrixType SpacingMatrix;
SpacingMatrix.Fill( 0.0F );

const ImageType::SpacingType & ImageSpacing = image->GetSpacing();
SpacingMatrix( 0,0 ) = ImageSpacing[0];
SpacingMatrix( 1,1 ) = ImageSpacing[1];
SpacingMatrix( 2,2 ) = ImageSpacing[2];

const ImageType::DirectionType & ImageDirectionCosines =
  image->GetDirection();
const ImageType::PointType &ImageOrigin = image->GetOrigin();

typedef itk::Vector< double, Dimension > VectorType;
VectorType LeftEyeIndexVector;
LeftEyeIndexVector[0]= LeftEyeIndex[0];
LeftEyeIndexVector[1]= LeftEyeIndex[1];
LeftEyeIndexVector[2]= LeftEyeIndex[2];

ImageType::PointType LeftEyePointByHand =
   ImageOrigin + ImageDirectionCosines * SpacingMatrix * LeftEyeIndexVector;
```

4.1.5 RGB Images

The term RGB (Red, Green, Blue) stands for a color representation commonly used in digital imaging. RGB is a representation of the human physiological capability to analyze visual light using three spectral-selective sensors [7, 9]. The human retina possess different types of light sensitive cells. Three of them, known as *cones*, are sensitive to color [5] and their regions of sensitivity loosely match regions of the spectrum that will be perceived as red, green and blue respectively. The *rods* on the other hand provide no color discrimination and favor high resolution and high sensitivity[1]. A fifth type of receptors, the *ganglion cells*, also known as circadian[2] receptors are sensitive to the lighting conditions that differentiate day from night. These receptors evolved as a mechanism for synchronizing the physiology with the time of the day. Cellular controls for circadian rythms are present in every cell of an organism and are known to be exquisitively precise [6].

The RGB space has been constructed as a representation of a physiological response to light by the three types of *cones* in the human eye. RGB is not a Vector space. For example, negative numbers are not appropriate in a color space because they will be the equivalent of "negative stimulation" on the human eye. In the context of colorimetry, negative color values are used as an artificial construct for color comparison in the sense that

$$ColorA = ColorB - ColorC \tag{4.2}$$

is just a way of saying that we can produce *ColorB* by combining *ColorA* and *ColorC*. However,

[1] The human eye is capable of perceiving a single isolated photon.

[2] The term *Circadian* refers to the cycle of day and night, that is, events that are repeated with 24 hours intervals.

we must be aware that (at least in emitted light) it is not possible to *substract light*. So when we mention Equation 4.2 we actually mean

$$ColorB = ColorA + ColorC \tag{4.3}$$

On the other hand, when dealing with printed color and with paint, as opposed to emitted light like in computer screens, the physical behavior of color allows for subtraction. This is because strictly speaking the objects that we see as red are those that absorb all light frequencies except those in the red section of the spectrum [9].

The concept of addition and subtraction of colors has to be carefully interpreted. In fact, RGB has a different definition regarding whether we are talking about the channels associated to the three color sensors of the human eye, or to the three phosphors found in most computer monitors or to the color inks that are used for printing reproduction. Color spaces are usually non linear and do not even from a group. For example, not all visible colors can be represented in RGB space [9].

ITK introduces the `itk::RGBPixel` type as a support for representing the values of an RGB color space. As such, the RGBPixel class embodies a different concept from the one of an `itk::Vector` in space. For this reason, the RGBPixel lack many of the operators that may be naively expected from it. In particular, there are no defined operations for subtraction or addition.

When you anticipate to perform the operation of "Mean" on a RGB type you are assuming that in the color space provides the action of finding a color in the middle of two colors, can be found by using a linear operation between their numerical representation. This is unfortunately not the case in color spaces due to the fact that they are based on a human physiological response [7].

If you decide to interpret RGB images as simply three independent channels then you should rather use the `itk::Vector` type as pixel type. In this way, you will have access to the set of operations that are defined in Vector spaces. The current implementation of the RGBPixel in ITK presumes that RGB color images are intended to be used in applications where a formal interpretation of color is desired, therefore only the operations that are valid in a color space are available in the RGBPixel class.

The following example illustrates how RGB images can be represented in ITK.

The source code for this section can be found in the file
`RGBImage.cxx`.

Thanks to the flexibility offered by the Generic Programming style on which ITK is based, it is possible to instantiate images of arbitrary pixel type. The following example illustrates how a color image with RGB pixels can be defined.

A class intended to support the RGB pixel type is available in ITK. You could also define your own pixel class and use it to instantiate a custom image type. In order to use the `itk::RGBPixel` class, it is necessary to include its header file.

```
#include "itkRGBPixel.h"
```

The RGB pixel class is templated over a type used to represent each one of the red, green and blue pixel components. A typical instantiation of the templated class is as follows.

```
typedef itk::RGBPixel< unsigned char >    PixelType;
```

The type is then used as the pixel template parameter of the image.

```
typedef itk::Image< PixelType, 3 >   ImageType;
```

The image type can be used to instantiate other filter, for example, an `itk::ImageFileReader` object that will read the image from a file.

```
typedef itk::ImageFileReader< ImageType >  ReaderType;
```

Access to the color components of the pixels can now be performed using the methods provided by the RGBPixel class.

```
PixelType onePixel = image->GetPixel( pixelIndex );

PixelType::ValueType red   = onePixel.GetRed();
PixelType::ValueType green = onePixel.GetGreen();
PixelType::ValueType blue  = onePixel.GetBlue();
```

The subindex notation can also be used since the `itk::RGBPixel` inherits the `[]` operator from the `itk::FixedArray` class.

```
red   = onePixel[0];  // extract Red   component
green = onePixel[1];  // extract Green component
blue  = onePixel[2];  // extract Blue  component

std::cout << "Pixel values:" << std::endl;
std::cout << "Red = "
          << itk::NumericTraits<PixelType::ValueType>::PrintType(red)
          << std::endl;
std::cout << "Green = "
          << itk::NumericTraits<PixelType::ValueType>::PrintType(green)
          << std::endl;
std::cout << "Blue = "
          << itk::NumericTraits<PixelType::ValueType>::PrintType(blue)
          << std::endl;
```

4.1.6 Vector Images

The source code for this section can be found in the file `VectorImage.cxx`.

Many image processing tasks require images of non-scalar pixel type. A typical example is an image of vectors. This is the image type required to represent the gradient of a scalar image. The following

code illustrates how to instantiate and use an image whose pixels are of vector type.

For convenience we use the itk::Vector class to define the pixel type. The Vector class is intended to represent a geometrical vector in space. It is not intended to be used as an array container like the std::vector in STL. If you are interested in containers, the itk::VectorContainer class may provide the functionality you want.

The first step is to include the header file of the Vector class.

```
#include "itkVector.h"
```

The Vector class is templated over the type used to represent the coordinate in space and over the dimension of the space. In this example, we want the vector dimension to match the image dimension, but this is by no means a requirement. We could have defined a four-dimensional image with three-dimensional vectors as pixels.

```
  typedef itk::Vector< float, 3 >       PixelType;
  typedef itk::Image< PixelType, 3 >    ImageType;
```

The Vector class inherits the operator [] from the itk::FixedArray class. This makes it possible to access the Vector's components using index notation.

```
  ImageType::PixelType   pixelValue;
  pixelValue[0] =  1.345;   // x component
  pixelValue[1] =  6.841;   // y component
  pixelValue[2] =  3.295;   // x component
```

We can now store this vector in one of the image pixels by defining an index and invoking the SetPixel() method.

```
  image->SetPixel(   pixelIndex,   pixelValue  );
```

4.1.7 Importing Image Data from a Buffer

The source code for this section can be found in the file
Image5.cxx.

This example illustrates how to import data into the itk::Image class. This is particularly useful for interfacing with other software systems. Many systems use a contiguous block of memory as a buffer for image pixel data. The current example assumes this is the case and feeds the buffer into an itk::ImportImageFilter, thereby producing an image as output.

Here we create a synthetic image with a centered sphere in a locally allocated buffer and pass this block of memory to the ImportImageFilter. This example is set up so that on execution, the user must provide the name of an output file as a command-line argument.

First, the header file of the itk::ImportImageFilter class must be included.

```
#include "itkImage.h"
#include "itkImportImageFilter.h"
```

Next, we select the data type used to represent the image pixels. We assume that the external block of memory uses the same data type to represent the pixels.

```
typedef unsigned char   PixelType;
const unsigned int Dimension = 3;

typedef itk::Image< PixelType, Dimension > ImageType;
```

The type of the `ImportImageFilter` is instantiated in the following line.

```
typedef itk::ImportImageFilter< PixelType, Dimension >   ImportFilterType;
```

A filter object created using the `New()` method is then assigned to a `SmartPointer`.

```
ImportFilterType::Pointer importFilter = ImportFilterType::New();
```

This filter requires the user to specify the size of the image to be produced as output. The `SetRegion()` method is used to this end. The image size should exactly match the number of pixels available in the locally allocated buffer.

```
ImportFilterType::SizeType  size;

size[0]  = 200;  // size along X
size[1]  = 200;  // size along Y
size[2]  = 200;  // size along Z

ImportFilterType::IndexType start;
start.Fill( 0 );

ImportFilterType::RegionType region;
region.SetIndex( start );
region.SetSize(  size  );

importFilter->SetRegion( region );
```

The origin of the output image is specified with the `SetOrigin()` method.

```
const itk::SpacePrecisionType origin[ Dimension ] = { 0.0, 0.0, 0.0 };
importFilter->SetOrigin( origin );
```

The spacing of the image is passed with the `SetSpacing()` method.

```
// spacing isotropic volumes to 1.0
const itk::SpacePrecisionType  spacing[ Dimension ] =  { 1.0, 1.0, 1.0 };
importFilter->SetSpacing( spacing );
```

Next we allocate the memory block containing the pixel data to be passed to the

ImportImageFilter. Note that we use exactly the same size that was specified with the SetRegion() method. In a practical application, you may get this buffer from some other library using a different data structure to represent the images.

```
const unsigned int numberOfPixels =  size[0] * size[1] * size[2];
PixelType * localBuffer = new PixelType[ numberOfPixels ];
```

Here we fill up the buffer with a binary sphere. We use simple for() loops here, similar to those found in the C or FORTRAN programming languages. Note that ITK does not use for() loops in its internal code to access pixels. All pixel access tasks are instead performed using an itk::ImageIterator that supports the management of n-dimensional images.

```
const double radius2 = radius * radius;
PixelType * it = localBuffer;

for(unsigned int z=0; z < size[2]; z++)
  {
  const double dz = static_cast<double>( z )
    - static_cast<double>(size[2])/2.0;
  for(unsigned int y=0; y < size[1]; y++)
    {
    const double dy = static_cast<double>( y )
      - static_cast<double>(size[1])/2.0;
    for(unsigned int x=0; x < size[0]; x++)
      {
      const double dx = static_cast<double>( x )
        - static_cast<double>(size[0])/2.0;
      const double d2 = dx*dx + dy*dy + dz*dz;
      *it++ = ( d2 < radius2 ) ? 255 : 0;
      }
    }
  }
```

The buffer is passed to the ImportImageFilter with the SetImportPointer() method. Note that the last argument of this method specifies who will be responsible for deleting the memory block once it is no longer in use. A false value indicates that the ImportImageFilter will not try to delete the buffer when its destructor is called. A true value, on the other hand, will allow the filter to delete the memory block upon destruction of the import filter.

For the ImportImageFilter to appropriately delete the memory block, the memory must be allocated with the C++ new() operator. Memory allocated with other memory allocation mechanisms, such as C malloc or calloc, will not be deleted properly by the ImportImageFilter. In other words, it is the application programmer's responsibility to ensure that ImportImageFilter is only given permission to delete the C++ new operator-allocated memory.

```
const bool importImageFilterWillOwnTheBuffer = true;
importFilter->SetImportPointer( localBuffer, numberOfPixels,
                                importImageFilterWillOwnTheBuffer );
```

Finally, we can connect the output of this filter to a pipeline. For simplicity we just use a writer here,

but it could be any other filter.

```
typedef itk::ImageFileWriter< ImageType > WriterType;
WriterType::Pointer writer = WriterType::New();

writer->SetFileName( argv[1] );
writer->SetInput(  importFilter->GetOutput()  );
```

Note that we do not call `delete` on the buffer since we pass `true` as the last argument of `SetImportPointer()`. Now the buffer is owned by the `ImportImageFilter`.

4.2 PointSet

4.2.1 Creating a PointSet

The source code for this section can be found in the file `PointSet1.cxx`.

The `itk::PointSet` is a basic class intended to represent geometry in the form of a set of points in n-dimensional space. It is the base class for the `itk::Mesh` providing the methods necessary to manipulate sets of point. Points can have values associated with them. The type of such values is defined by a template parameter of the `itk::PointSet` class (i.e., `TPixelType`. Two basic interaction styles of PointSets are available in ITK. These styles are referred to as *static* and *dynamic*. The first style is used when the number of points in the set is known in advance and is not expected to change as a consequence of the manipulations performed on the set. The dynamic style, on the other hand, is intended to support insertion and removal of points in an efficient manner. Distinguishing between the two styles is meant to facilitate the fine tuning of a `PointSet`'s behavior while optimizing performance and memory management.

In order to use the PointSet class, its header file should be included.

```
#include "itkPointSet.h"
```

Then we must decide what type of value to associate with the points. This is generally called the `PixelType` in order to make the terminology consistent with the `itk::Image`. The PointSet is also templated over the dimension of the space in which the points are represented. The following declaration illustrates a typical instantiation of the PointSet class.

```
typedef itk::PointSet< unsigned short, 3 > PointSetType;
```

A `PointSet` object is created by invoking the `New()` method on its type. The resulting object must be assigned to a `SmartPointer`. The PointSet is then reference-counted and can be shared by multiple objects. The memory allocated for the PointSet will be released when the number of references to the object is reduced to zero. This simply means that the user does not need to be concerned with

invoking the `Delete()` method on this class. In fact, the `Delete()` method should **never** be called directly within any of the reference-counted ITK classes.

```
PointSetType::Pointer  pointsSet = PointSetType::New();
```

Following the principles of Generic Programming, the `PointSet` class has a set of associated defined types to ensure that interacting objects can be declared with compatible types. This set of type definitions is commonly known as a set of *traits*. Among them we can find the `PointType` type, for example. This is the type used by the point set to represent points in space. The following declaration takes the point type as defined in the `PointSet` traits and renames it to be conveniently used in the global namespace.

```
typedef PointSetType::PointType     PointType;
```

The `PointType` can now be used to declare point objects to be inserted in the `PointSet`. Points are fairly small objects, so it is inconvenient to manage them with reference counting and smart pointers. They are simply instantiated as typical C++ classes. The Point class inherits the `[]` operator from the `itk::Array` class. This makes it possible to access its components using index notation. For efficiency's sake no bounds checking is performed during index access. It is the user's responsibility to ensure that the index used is in the range $\{0, Dimension - 1\}$. Each of the components in the point is associated with space coordinates. The following code illustrates how to instantiate a point and initialize its components.

```
PointType p0;
p0[0] = -1.0;     //  x coordinate
p0[1] = -1.0;     //  y coordinate
p0[2] =  0.0;     //  z coordinate
```

Points are inserted in the PointSet by using the `SetPoint()` method. This method requires the user to provide a unique identifier for the point. The identifier is typically an unsigned integer that will enumerate the points as they are being inserted. The following code shows how three points are inserted into the PointSet.

```
pointsSet->SetPoint( 0, p0 );
pointsSet->SetPoint( 1, p1 );
pointsSet->SetPoint( 2, p2 );
```

It is possible to query the PointSet in order to determine how many points have been inserted into it. This is done with the `GetNumberOfPoints()` method as illustrated below.

```
const unsigned int numberOfPoints = pointsSet->GetNumberOfPoints();
std::cout << numberOfPoints << std::endl;
```

Points can be read from the PointSet by using the `GetPoint()` method and the integer identifier. The point is stored in a pointer provided by the user. If the identifier provided does not match an existing point, the method will return `false` and the contents of the point will be invalid. The following code illustrates point access using defensive programming.

```
PointType pp;
bool pointExists =  pointsSet->GetPoint( 1, & pp );

if( pointExists )
  {
  std::cout << "Point is = " << pp << std::endl;
  }
```

`GetPoint()` and `SetPoint()` are not the most efficient methods to access points in the PointSet. It is preferable to get direct access to the internal point container defined by the *traits* and use iterators to walk sequentially over the list of points (as shown in the following example).

4.2.2 Getting Access to Points

The source code for this section can be found in the file `PointSet2.cxx`.

The `itk::PointSet` class uses an internal container to manage the storage of `itk::Point`s. It is more efficient, in general, to manage points by using the access methods provided directly on the points container. The following example illustrates how to interact with the point container and how to use point iterators.

The type is defined by the *traits* of the PointSet class. The following line conveniently takes the PointsContainer type from the PointSet traits and declare it in the global namespace.

```
typedef PointSetType::PointsContainer      PointsContainer;
```

The actual type of the PointsContainer depends on what style of PointSet is being used. The dynamic PointSet use the `itk::MapContainer` while the static PointSet uses the `itk::VectorContainer`. The vector and map containers are basically ITK wrappers around the STL classes `std::map` and `std::vector`. By default, the PointSet uses a static style, hence the default type of point container is an VectorContainer. Both the map and vector container are templated over the type of the elements they contain. In this case they are templated over PointType. Containers are reference counted object. They are then created with the `New()` method and assigned to a `itk::SmartPointer` after creation. The following line creates a point container compatible with the type of the PointSet from which the trait has been taken.

```
PointsContainer::Pointer points = PointsContainer::New();
```

Points can now be defined using the `PointType` trait from the PointSet.

```
typedef PointSetType::PointType   PointType;
PointType p0;
PointType p1;
p0[0] = -1.0; p0[1] = 0.0; p0[2] = 0.0; // Point 0 = {-1,0,0 }
p1[0] =  1.0; p1[1] = 0.0; p1[2] = 0.0; // Point 1 = { 1,0,0 }
```

The created points can be inserted in the PointsContainer using the generic method `InsertElement()` which requires an identifier to be provided for each point.

```
unsigned int pointId = 0;
points->InsertElement( pointId++ , p0 );
points->InsertElement( pointId++ , p1 );
```

Finally the PointsContainer can be assigned to the PointSet. This will substitute any previously existing PointsContainer on the PointSet. The assignment is done using the `SetPoints()` method.

```
pointSet->SetPoints( points );
```

The PointsContainer object can be obtained from the PointSet using the `GetPoints()` method. This method returns a pointer to the actual container owned by the PointSet which is then assigned to a SmartPointer.

```
PointsContainer::Pointer  points2 = pointSet->GetPoints();
```

The most efficient way to sequentially visit the points is to use the iterators provided by PointsContainer. The `Iterator` type belongs to the traits of the PointsContainer classes. It behaves pretty much like the STL iterators.[3] The Points iterator is not a reference counted class, so it is created directly from the traits without using SmartPointers.

```
typedef PointsContainer::Iterator     PointsIterator;
```

The subsequent use of the iterator follows what you may expect from a STL iterator. The iterator to the first point is obtained from the container with the `Begin()` method and assigned to another iterator.

```
PointsIterator  pointIterator = points->Begin();
```

The ++ operator on the iterator can be used to advance from one point to the next. The actual value of the Point to which the iterator is pointing can be obtained with the `Value()` method. The loop for walking through all the points can be controlled by comparing the current iterator with the iterator returned by the `End()` method of the PointsContainer. The following lines illustrate the typical loop for walking through the points.

```
PointsIterator end = points->End();
while( pointIterator != end )
  {
  PointType p = pointIterator.Value();   // access the point
  std::cout << p << std::endl;           // print the point
  ++pointIterator;                       // advance to next point
  }
```

[3] If you dig deep enough into the code, you will discover that these iterators are actually ITK wrappers around STL iterators.

Note that as in STL, the iterator returned by the `End()` method is not a valid iterator. This is called a past-end iterator in order to indicate that it is the value resulting from advancing one step after visiting the last element in the container.

The number of elements stored in a container can be queried with the `Size()` method. In the case of the PointSet, the following two lines of code are equivalent, both of them returning the number of points in the PointSet.

```
std::cout << pointSet->GetNumberOfPoints() << std::endl;
std::cout << pointSet->GetPoints()->Size() << std::endl;
```

4.2.3 Getting Access to Data in Points

The source code for this section can be found in the file `PointSet3.cxx`.

The `itk::PointSet` class was designed to interact with the Image class. For this reason it was found convenient to allow the points in the set to hold values that could be computed from images. The value associated with the point is referred as `PixelType` in order to make it consistent with image terminology. Users can define the type as they please thanks to the flexibility offered by the Generic Programming approach used in the toolkit. The `PixelType` is the first template parameter of the PointSet.

The following code defines a particular type for a pixel type and instantiates a PointSet class with it.

```
typedef unsigned short               PixelType;
typedef itk::PointSet< PixelType, 3 > PointSetType;
```

Data can be inserted into the PointSet using the `SetPointData()` method. This method requires the user to provide an identifier. The data in question will be associated to the point holding the same identifier. It is the user's responsibility to verify the appropriate matching between inserted data and inserted points. The following line illustrates the use of the `SetPointData()` method.

```
unsigned int dataId =  0;
PixelType value     = 79;
pointSet->SetPointData( dataId++, value );
```

Data associated with points can be read from the PointSet using the `GetPointData()` method. This method requires the user to provide the identifier to the point and a valid pointer to a location where the pixel data can be safely written. In case the identifier does not match any existing identifier on the PointSet the method will return `false` and the pixel value returned will be invalid. It is the user's responsibility to check the returned boolean value before attempting to use it.

```
const bool found = pointSet->GetPointData( dataId, & value );
if( found )
  {
  std::cout << "Pixel value = " << value << std::endl;
  }
```

The `SetPointData()` and `GetPointData()` methods are not the most efficient way to get access to point data. It is far more efficient to use the Iterators provided by the `PointDataContainer`.

Data associated with points is internally stored in `PointDataContainers`. In the same way as with points, the actual container type used depend on whether the style of the PointSet is static or dynamic. Static point sets will use an `itk::VectorContainer` while dynamic point sets will use an `itk::MapContainer`. The type of the data container is defined as one of the traits in the PointSet. The following declaration illustrates how the type can be taken from the traits and used to conveniently declare a similar type on the global namespace.

```
typedef PointSetType::PointDataContainer      PointDataContainer;
```

Using the type it is now possible to create an instance of the data container. This is a standard reference counted object, henceforth it uses the `New()` method for creation and assigns the newly created object to a SmartPointer.

```
PointDataContainer::Pointer pointData = PointDataContainer::New();
```

Pixel data can be inserted in the container with the method `InsertElement()`. This method requires an identified to be provided for each point data.

```
unsigned int pointId = 0;

PixelType value0 = 34;
PixelType value1 = 67;

pointData->InsertElement( pointId++ , value0 );
pointData->InsertElement( pointId++ , value1 );
```

Finally the PointDataContainer can be assigned to the PointSet. This will substitute any previously existing PointDataContainer on the PointSet. The assignment is done using the `SetPointData()` method.

```
pointSet->SetPointData( pointData );
```

The PointDataContainer can be obtained from the PointSet using the `GetPointData()` method. This method returns a pointer (assigned to a SmartPointer) to the actual container owned by the PointSet.

```
PointDataContainer::Pointer  pointData2 = pointSet->GetPointData();
```

The most efficient way to sequentially visit the data associated with points is to use the iterators provided by `PointDataContainer`. The `Iterator` type belongs to the traits of the PointsContainer classes. The iterator is not a reference counted class, so it is just created directly from the traits without using SmartPointers.

```
typedef PointDataContainer::Iterator     PointDataIterator;
```

The subsequent use of the iterator follows what you may expect from a STL iterator. The iterator to the first point is obtained from the container with the `Begin()` method and assigned to another iterator.

```
PointDataIterator  pointDataIterator = pointData2->Begin();
```

The ++ operator on the iterator can be used to advance from one data point to the next. The actual value of the PixelType to which the iterator is pointing can be obtained with the `Value()` method. The loop for walking through all the point data can be controlled by comparing the current iterator with the iterator returned by the `End()` method of the PointsContainer. The following lines illustrate the typical loop for walking through the point data.

```
PointDataIterator end = pointData2->End();
while( pointDataIterator != end )
  {
  PixelType p = pointDataIterator.Value();  // access the pixel data
  std::cout << p << std::endl;              // print the pixel data
  ++pointDataIterator;                      // advance to next pixel/point
  }
```

Note that as in STL, the iterator returned by the `End()` method is not a valid iterator. This is called a *past-end* iterator in order to indicate that it is the value resulting from advancing one step after visiting the last element in the container.

4.2.4 RGB as Pixel Type

The source code for this section can be found in the file `RGBPointSet.cxx`.

The following example illustrates how a point set can be parameterized to manage a particular pixel type. In this case, pixels of RGB type are used. The first step is then to include the header files of the `itk::RGBPixel` and `itk::PointSet` classes.

```
#include "itkRGBPixel.h"
#include "itkPointSet.h"
```

Then, the pixel type can be defined by selecting the type to be used to represent each one of the RGB components.

```
typedef itk::RGBPixel< float >    PixelType;
```

The newly defined pixel type is now used to instantiate the PointSet type and subsequently create a point set object.

```
typedef itk::PointSet< PixelType, 3 > PointSetType;
PointSetType::Pointer  pointSet = PointSetType::New();
```

The following code is generating a sphere and assigning RGB values to the points. The components of the RGB values in this example are computed to represent the position of the points.

```
PointSetType::PixelType   pixel;
PointSetType::PointType   point;
unsigned int pointId =  0;
const double radius = 3.0;

for(unsigned int i=0; i<360; i++)
  {
  const double angle = i * vnl_math::pi / 180.0;
  point[0] = radius * std::sin( angle );
  point[1] = radius * std::cos( angle );
  point[2] = 1.0;
  pixel.SetRed(    point[0] * 2.0 );
  pixel.SetGreen(  point[1] * 2.0 );
  pixel.SetBlue(   point[2] * 2.0 );
  pointSet->SetPoint( pointId, point );
  pointSet->SetPointData( pointId, pixel );
  pointId++;
  }
```

All the points on the PointSet are visited using the following code.

```
typedef  PointSetType::PointsContainer::ConstIterator     PointIterator;
PointIterator pointIterator = pointSet->GetPoints()->Begin();
PointIterator pointEnd      = pointSet->GetPoints()->End();
while( pointIterator != pointEnd )
  {
  point = pointIterator.Value();
  std::cout << point << std::endl;
  ++pointIterator;
  }
```

Note that here the `ConstIterator` was used instead of the `Iterator` since the pixel values are not expected to be modified. ITK supports const-correctness at the API level.

All the pixel values on the PointSet are visited using the following code.

```
typedef  PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData()->Begin();
PointDataIterator pixelEnd      = pointSet->GetPointData()->End();
while( pixelIterator != pixelEnd )
  {
  pixel = pixelIterator.Value();
  std::cout << pixel << std::endl;
  ++pixelIterator;
  }
```

Again, please note the use of the `ConstIterator` instead of the `Iterator`.

4.2.5 Vectors as Pixel Type

The source code for this section can be found in the file `PointSetWithVectors.cxx`.

This example illustrates how a point set can be parameterized to manage a particular pixel type. It is quite common to associate vector values with points for producing geometric representations. The following code shows how vector values can be used as the pixel type on the PointSet class. The `itk::Vector` class is used here as the pixel type. This class is appropriate for representing the relative position between two points. It could then be used to manage displacements, for example.

In order to use the vector class it is necessary to include its header file along with the header of the point set.

```
#include "itkVector.h"
#include "itkPointSet.h"
```

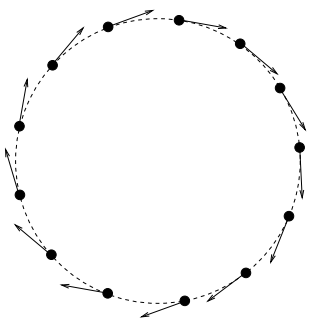

Figure 4.2: Vectors as PixelType.

The Vector class is templated over the type used to represent the spatial coordinates and over the space dimension. Since the PixelType is independent of the PointType, we are free to select any dimension for the vectors to be used as pixel type. However, for the sake of producing an interesting example, we will use vectors that represent displacements of the points in the PointSet. Those vectors are then selected to be of the same dimension as the PointSet.

```
const unsigned int Dimension = 3;
typedef itk::Vector< float, Dimension >    PixelType;
```

Then we use the PixelType (which are actually Vectors) to instantiate the PointSet type and subsequently create a PointSet object.

```
typedef itk::PointSet< PixelType, Dimension > PointSetType;
PointSetType::Pointer  pointSet = PointSetType::New();
```

The following code is generating a sphere and assigning vector values to the points. The components of the vectors in this example are computed to represent the tangents to the circle as shown in Figure 4.2.

```
PointSetType::PixelType   tangent;
PointSetType::PointType   point;

unsigned int pointId =  0;
const double radius = 300.0;

for(unsigned int i=0; i<360; i++)
  {
  const double angle = i * vnl_math::pi / 180.0;
  point[0] = radius * std::sin( angle );
  point[1] = radius * std::cos( angle );
  point[2] = 1.0;   // flat on the Z plane
  tangent[0] =  std::cos(angle);
  tangent[1] = -std::sin(angle);
  tangent[2] = 0.0;  // flat on the Z plane
  pointSet->SetPoint( pointId, point );
  pointSet->SetPointData( pointId, tangent );
  pointId++;
  }
```

We can now visit all the points and use the vector on the pixel values to apply a displacement on the points. This is along the spirit of what a deformable model could do at each one of its iterations.

```
typedef  PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData()->Begin();
PointDataIterator pixelEnd      = pointSet->GetPointData()->End();

typedef  PointSetType::PointsContainer::Iterator     PointIterator;
PointIterator pointIterator = pointSet->GetPoints()->Begin();
PointIterator pointEnd      = pointSet->GetPoints()->End();

while( pixelIterator != pixelEnd  && pointIterator != pointEnd )
  {
  pointIterator.Value() = pointIterator.Value() + pixelIterator.Value();
  ++pixelIterator;
  ++pointIterator;
  }
```

Note that the `ConstIterator` was used here instead of the normal `Iterator` since the pixel values are only intended to be read and not modified. ITK supports const-correctness at the API level.

The `itk::Vector` class has overloaded the + operator with the `itk::Point`. In other words,

vectors can be added to points in order to produce new points. This property is exploited in the center of the loop in order to update the points positions with a single statement.

We can finally visit all the points and print out the new values

```
pointIterator = pointSet->GetPoints()->Begin();
pointEnd      = pointSet->GetPoints()->End();
while( pointIterator != pointEnd )
  {
  std::cout << pointIterator.Value() << std::endl;
  ++pointIterator;
  }
```

Note that `itk::Vector` is not the appropriate class for representing normals to surfaces and gradients of functions. This is due to the way vectors behave under affine transforms. ITK has a specific class for representing normals and function gradients. This is the `itk::CovariantVector` class.

4.2.6 Normals as Pixel Type

The source code for this section can be found in the file
`PointSetWithCovariantVectors.cxx`.

It is common to represent geometric object by using points on their surfaces and normals associated with those points. This structure can be easily instantiated with the `itk::PointSet` class.

The natural class for representing normals to surfaces and gradients of functions is the `itk::CovariantVector`. A covariant vector differs from a vector in the way they behave under affine transforms, in particular under anisotropic scaling. If a covariant vector represents the gradient of a function, the transformed covariant vector will still be the valid gradient of the transformed function, a property which would not hold with a regular vector.

The following code shows how vector values can be used as pixel type on the PointSet class. The CovariantVector class is used here as the pixel type. The example illustrates how a deformable model could move under the influence of the gradient of potential function.

In order to use the CovariantVector class it is necessary to include its header file along with the header of the point set.

```
#include "itkCovariantVector.h"
#include "itkPointSet.h"
```

The CovariantVector class is templated over the type used to represent the spatial coordinates and over the space dimension. Since the PixelType is independent of the PointType, we are free to select any dimension for the covariant vectors to be used as pixel type. However, we want to illustrate here the spirit of a deformable model. It is then required for the vectors representing gradients to be of the same dimension as the points in space.

```
const unsigned int Dimension = 3;
typedef itk::CovariantVector< float, Dimension >    PixelType;
```

Then we use the PixelType (which are actually CovariantVectors) to instantiate the PointSet type and subsequently create a PointSet object.

```
typedef itk::PointSet< PixelType, Dimension > PointSetType;
PointSetType::Pointer  pointSet = PointSetType::New();
```

The following code generates a sphere and assigns gradient values to the points. The components of the CovariantVectors in this example are computed to represent the normals to the circle.

```
PointSetType::PixelType   gradient;
PointSetType::PointType   point;

unsigned int pointId =  0;
const double radius = 300.0;

for(unsigned int i=0; i<360; i++)
  {
  const double angle = i * std::atan(1.0) / 45.0;
  point[0] = radius * std::sin( angle );
  point[1] = radius * std::cos( angle );
  point[2] = 1.0;   // flat on the Z plane
  gradient[0] =  std::sin(angle);
  gradient[1] =  std::cos(angle);
  gradient[2] = 0.0;  // flat on the Z plane
  pointSet->SetPoint( pointId, point );
  pointSet->SetPointData( pointId, gradient );
  pointId++;
  }
```

We can now visit all the points and use the vector on the pixel values to apply a deformation on the points by following the gradient of the function. This is along the spirit of what a deformable model could do at each one of its iterations. To be more formal we should use the function gradients as forces and multiply them by local stress tensors in order to obtain local deformations. The resulting deformations would finally be used to apply displacements on the points. However, to shorten the example, we will ignore this complexity for the moment.

```
typedef  PointSetType::PointDataContainer::ConstIterator PointDataIterator;
PointDataIterator pixelIterator = pointSet->GetPointData()->Begin();
PointDataIterator pixelEnd      = pointSet->GetPointData()->End();

typedef  PointSetType::PointsContainer::Iterator     PointIterator;
PointIterator pointIterator = pointSet->GetPoints()->Begin();
PointIterator pointEnd      = pointSet->GetPoints()->End();

while( pixelIterator != pixelEnd  && pointIterator != pointEnd )
  {
  point    = pointIterator.Value();
  gradient = pixelIterator.Value();
  for(unsigned int i=0; i<Dimension; i++)
    {
    point[i] += gradient[i];
    }
  pointIterator.Value() = point;
  ++pixelIterator;
  ++pointIterator;
  }
```

The CovariantVector class does not overload the + operator with the `itk::Point`. In other words, CovariantVectors can not be added to points in order to get new points. Further, since we are ignoring physics in the example, we are also forced to do the illegal addition manually between the components of the gradient and the coordinates of the points.

Note that the absence of some basic operators on the ITK geometry classes is completely intentional with the aim of preventing the incorrect use of the mathematical concepts they represent.

4.3 Mesh

4.3.1 Creating a Mesh

The source code for this section can be found in the file
`Mesh1.cxx`.

The `itk::Mesh` class is intended to represent shapes in space. It derives from the `itk::PointSet` class and hence inherits all the functionality related to points and access to the pixel-data associated with the points. The mesh class is also n-dimensional which allows a great flexibility in its use.

In practice a Mesh class can be seen as a PointSet to which cells (also known as elements) of many different dimensions and shapes have been added. Cells in the mesh are defined in terms of the existing points using their point-identifiers.

In the same way as for the PointSet, two basic styles of Meshes are available in ITK. They are referred to as *static* and *dynamic*. The first one is used when the number of points in the set can be known in advance and it is not expected to change as a consequence of the manipulations performed on the set. The dynamic style, on the other hand, is intended to support insertion and removal of

points in an efficient manner. The reason for making the distinction between the two styles is to facilitate fine tuning its behavior with the aim of optimizing performance and memory management. In the case of the Mesh, the dynamic/static aspect is extended to the management of cells.

In order to use the Mesh class, its header file should be included.

```
#include "itkMesh.h"
```

Then, the type associated with the points must be selected and used for instantiating the Mesh type.

```
  typedef   float   PixelType;
```

The Mesh type extensively uses the capabilities provided by Generic Programming. In particular the Mesh class is parameterized over the PixelType and the dimension of the space. PixelType is the type of the value associated with every point just as is done with the PointSet. The following line illustrates a typical instantiation of the Mesh.

```
  const unsigned int Dimension = 3;
  typedef itk::Mesh< PixelType, Dimension >   MeshType;
```

Meshes are expected to take large amounts of memory. For this reason they are reference counted objects and are managed using SmartPointers. The following line illustrates how a mesh is created by invoking the `New()` method of the MeshType and the resulting object is assigned to a `itk::SmartPointer`.

```
  MeshType::Pointer  mesh = MeshType::New();
```

The management of points in the Mesh is exactly the same as in the PointSet. The type point associated with the mesh can be obtained through the `PointType` trait. The following code shows the creation of points compatible with the mesh type defined above and the assignment of values to its coordinates.

```
  MeshType::PointType p0;
  MeshType::PointType p1;
  MeshType::PointType p2;
  MeshType::PointType p3;

  p0[0]= -1.0; p0[1]= -1.0; p0[2]= 0.0; // first  point ( -1, -1, 0 )
  p1[0]=  1.0; p1[1]= -1.0; p1[2]= 0.0; // second point (  1, -1, 0 )
  p2[0]=  1.0; p2[1]=  1.0; p2[2]= 0.0; // third  point (  1,  1, 0 )
  p3[0]= -1.0; p3[1]=  1.0; p3[2]= 0.0; // fourth point ( -1,  1, 0 )
```

The points can now be inserted in the Mesh using the `SetPoint()` method. Note that points are copied into the mesh structure. This means that the local instances of the points can now be modified without affecting the Mesh content.

```
mesh->SetPoint( 0, p0 );
mesh->SetPoint( 1, p1 );
mesh->SetPoint( 2, p2 );
mesh->SetPoint( 3, p3 );
```

The current number of points in the Mesh can be queried with the `GetNumberOfPoints()` method.

```
std::cout << "Points = " << mesh->GetNumberOfPoints() << std::endl;
```

The points can now be efficiently accessed using the Iterator to the PointsContainer as it was done in the previous section for the PointSet. First, the point iterator type is extracted through the mesh traits.

```
typedef MeshType::PointsContainer::Iterator     PointsIterator;
```

A point iterator is initialized to the first point with the `Begin()` method of the PointsContainer.

```
PointsIterator  pointIterator = mesh->GetPoints()->Begin();
```

The ++ operator on the iterator is now used to advance from one point to the next. The actual value of the Point to which the iterator is pointing can be obtained with the `Value()` method. The loop for walking through all the points is controlled by comparing the current iterator with the iterator returned by the `End()` method of the PointsContainer. The following lines illustrate the typical loop for walking through the points.

```
PointsIterator end = mesh->GetPoints()->End();
while( pointIterator != end )
  {
  MeshType::PointType p = pointIterator.Value();  // access the point
  std::cout << p << std::endl;                    // print the point
  ++pointIterator;                                // advance to next point
  }
```

4.3.2 Inserting Cells

The source code for this section can be found in the file
`Mesh2.cxx`.

A `itk::Mesh` can contain a variety of cell types. Typical cells are the `itk::LineCell`, `itk::TriangleCell`, `itk::QuadrilateralCell` and `itk::TetrahedronCell`. Additional flexibility is provided for managing cells at the price of a bit more of complexity than in the case of point management.

The following code creates a polygonal line in order to illustrate the simplest case of cell management in a Mesh. The only cell type used here is the LineCell. The header file of this class has to be included.

```
#include "itkLineCell.h"
```

In order to be consistent with the Mesh, cell types have to be configured with a number of custom types taken from the mesh traits. The set of traits relevant to cells are packaged by the Mesh class into the `CellType` trait. This trait needs to be passed to the actual cell types at the moment of their instantiation. The following line shows how to extract the Cell traits from the Mesh type.

```
  typedef MeshType::CellType                CellType;
```

The LineCell type can now be instantiated using the traits taken from the Mesh.

```
  typedef itk::LineCell< CellType >         LineType;
```

The main difference in the way cells and points are managed by the Mesh is that points are stored by copy on the PointsContainer while cells are stored in the CellsContainer using pointers. The reason for using pointers is that cells use C++ polymorphism on the mesh. This means that the mesh is only aware of having pointers to a generic cell which is the base class of all the specific cell types. This architecture makes it possible to combine different cell types in the same mesh. Points, on the other hand, are of a single type and have a small memory footprint, which makes it efficient to copy them directly into the container.

Managing cells by pointers add another level of complexity to the Mesh since it is now necessary to establish a protocol to make clear who is responsible for allocating and releasing the cells' memory. This protocol is implemented in the form of a specific type of pointer called the `CellAutoPointer`. This pointer, based on the `itk::AutoPointer`, differs in many respects from the SmartPointer. The CellAutoPointer has an internal pointer to the actual object and a boolean flag that indicates if the CellAutoPointer is responsible for releasing the cell memory whenever the time comes for its own destruction. It is said that a `CellAutoPointer` *owns* the cell when it is responsible for its destruction. Many CellAutoPointer can point to the same cell but at any given time, only **one** CellAutoPointer can own the cell.

The `CellAutoPointer` trait is defined in the MeshType and can be extracted as illustrated in the following line.

```
  typedef CellType::CellAutoPointer         CellAutoPointer;
```

Note that the CellAutoPointer is pointing to a generic cell type. It is not aware of the actual type of the cell, which can be for example LineCell, TriangleCell or TetrahedronCell. This fact will influence the way in which we access cells later on.

At this point we can actually create a mesh and insert some points on it.

```
MeshType::Pointer  mesh = MeshType::New();

MeshType::PointType p0;
MeshType::PointType p1;
MeshType::PointType p2;

p0[0] = -1.0; p0[1] = 0.0; p0[2] = 0.0;
p1[0] =  1.0; p1[1] = 0.0; p1[2] = 0.0;
p2[0] =  1.0; p2[1] = 1.0; p2[2] = 0.0;

mesh->SetPoint( 0, p0 );
mesh->SetPoint( 1, p1 );
mesh->SetPoint( 2, p2 );
```

The following code creates two CellAutoPointers and initializes them with newly created cell objects. The actual cell type created in this case is LineCell. Note that cells are created with the normal `new` C++ operator. The CellAutoPointer takes ownership of the received pointer by using the method `TakeOwnership()`. Even though this may seem verbose, it is necessary in order to make it explicit from the code that the responsibility of memory release is assumed by the AutoPointer.

```
CellAutoPointer line0;
CellAutoPointer line1;

line0.TakeOwnership(  new LineType  );
line1.TakeOwnership(  new LineType  );
```

The LineCells should now be associated with points in the mesh. This is done using the identifiers assigned to points when they were inserted in the mesh. Every cell type has a specific number of points that must be associated with it.[4] For example a LineCell requires two points, a TriangleCell requires three and a TetrahedronCell requires four. Cells use an internal numbering system for points. It is simply an index in the range $\{0, NumberOfPoints - 1\}$. The association of points and cells is done by the `SetPointId()` method which requires the user to provide the internal index of the point in the cell and the corresponding PointIdentifier in the Mesh. The internal cell index is the first parameter of `SetPointId()` while the mesh point-identifier is the second.

```
line0->SetPointId( 0, 0 ); // line between points 0 and 1
line0->SetPointId( 1, 1 );

line1->SetPointId( 0, 1 ); // line between points 1 and 2
line1->SetPointId( 1, 2 );
```

Cells are inserted in the mesh using the `SetCell()` method. It requires an identifier and the AutoPointer to the cell. The Mesh will take ownership of the cell to which the AutoPointer is pointing. This is done internally by the `SetCell()` method. In this way, the destruction of the CellAutoPointer will not induce the destruction of the associated cell.

```
mesh->SetCell( 0, line0 );
mesh->SetCell( 1, line1 );
```

[4] Some cell types like polygons have a variable number of points associated with them.

After serving as an argument of the `SetCell()` method, a CellAutoPointer no longer holds ownership of the cell. It is important not to use this same CellAutoPointer again as argument to `SetCell()` without first securing ownership of another cell.

The number of Cells currently inserted in the mesh can be queried with the `GetNumberOfCells()` method.

```
std::cout << "Cells  = " << mesh->GetNumberOfCells()  << std::endl;
```

In a way analogous to points, cells can be accessed using Iterators to the CellsContainer in the mesh. The trait for the cell iterator can be extracted from the mesh and used to define a local type.

```
typedef MeshType::CellsContainer::Iterator  CellIterator;
```

Then the iterators to the first and past-end cell in the mesh can be obtained respectively with the `Begin()` and `End()` methods of the CellsContainer. The CellsContainer of the mesh is returned by the `GetCells()` method.

```
CellIterator  cellIterator = mesh->GetCells()->Begin();
CellIterator  end          = mesh->GetCells()->End();
```

Finally a standard loop is used to iterate over all the cells. Note the use of the `Value()` method used to get the actual pointer to the cell from the CellIterator. Note also that the values returned are pointers to the generic CellType. These pointers have to be down-casted in order to be used as actual LineCell types. Safe down-casting is performed with the `dynamic_cast` operator which will throw an exception if the conversion cannot be safely performed.

```
while( cellIterator != end )
  {
  MeshType::CellType * cellptr = cellIterator.Value();
  LineType * line = dynamic_cast<LineType *>( cellptr );
  if(line == ITK_NULLPTR)
    {
    continue;
    }
  std::cout << line->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

4.3.3 Managing Data in Cells

The source code for this section can be found in the file
`Mesh3.cxx`.

In the same way that custom data can be associated with points in the mesh, it is also possible to associate custom data with cells. The type of the data associated with the cells can be different from the data type associated with points. By default, however, these two types are the same. The

following example illustrates how to access data associated with cells. The approach is analogous to the one used to access point data.

Consider the example of a mesh containing lines on which values are associated with each line. The mesh and cell header files should be included first.

```
#include "itkMesh.h"
#include "itkLineCell.h"
```

Then the PixelType is defined and the mesh type is instantiated with it.

```
typedef float                           PixelType;
typedef itk::Mesh< PixelType, 2 >       MeshType;
```

The `itk::LineCell` type can now be instantiated using the traits taken from the Mesh.

```
typedef MeshType::CellType              CellType;
typedef itk::LineCell< CellType >       LineType;
```

Let's now create a Mesh and insert some points into it. Note that the dimension of the points matches the dimension of the Mesh. Here we insert a sequence of points that look like a plot of the log() function. We add the `vnl_math::eps` value in oder to avoid numerical errors when the point id is zero. The value of `vnl_math::eps` is the difference between 1.0 and the least value greater than 1.0 that is representable in this computer.

```
MeshType::Pointer  mesh = MeshType::New();

typedef MeshType::PointType PointType;
PointType point;

const unsigned int numberOfPoints = 10;
for(unsigned int id=0; id<numberOfPoints; id++)
  {
  point[0] = static_cast<PointType::ValueType>( id ); // x
  point[1] = std::log( static_cast<double>( id ) + vnl_math::eps );    // y
  mesh->SetPoint( id, point );
  }
```

A set of line cells is created and associated with the existing points by using point identifiers. In this simple case, the point identifiers can be deduced from cell identifiers since the line cells are ordered in the same way.

```
CellType::CellAutoPointer line;
const unsigned int numberOfCells = numberOfPoints-1;
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  line.TakeOwnership(  new LineType  );
  line->SetPointId( 0, cellId   ); // first point
  line->SetPointId( 1, cellId+1 ); // second point
  mesh->SetCell( cellId, line );   // insert the cell
  }
```

Data associated with cells is inserted in the itk::Mesh by using the SetCellData() method. It requires the user to provide an identifier and the value to be inserted. The identifier should match one of the inserted cells. In this simple example, the square of the cell identifier is used as cell data. Note the use of static_cast to PixelType in the assignment.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  mesh->SetCellData( cellId, static_cast<PixelType>( cellId * cellId ) );
  }
```

Cell data can be read from the Mesh with the GetCellData() method. It requires the user to provide the identifier of the cell for which the data is to be retrieved. The user should provide also a valid pointer to a location where the data can be copied.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  PixelType value = static_cast<PixelType>(0.0);
  mesh->GetCellData( cellId, &value );
  std::cout << "Cell " << cellId << " = " << value << std::endl;
  }
```

Neither SetCellData() or GetCellData() are efficient ways to access cell data. More efficient access to cell data can be achieved by using the Iterators built into the CellDataContainer.

```
typedef MeshType::CellDataContainer::ConstIterator CellDataIterator;
```

Note that the ConstIterator is used here because the data is only going to be read. This approach is exactly the same already illustrated for getting access to point data. The iterator to the first cell data item can be obtained with the Begin() method of the CellDataContainer. The past-end iterator is returned by the End() method. The cell data container itself can be obtained from the mesh with the method GetCellData().

```
CellDataIterator cellDataIterator  = mesh->GetCellData()->Begin();
CellDataIterator end               = mesh->GetCellData()->End();
```

Finally a standard loop is used to iterate over all the cell data entries. Note the use of the Value() method used to get the actual value of the data entry. PixelType elements are copied into the local variable cellValue.

```
while( cellDataIterator != end )
  {
  PixelType cellValue = cellDataIterator.Value();
  std::cout << cellValue << std::endl;
  ++cellDataIterator;
  }
```

4.3.4 Customizing the Mesh

The source code for this section can be found in the file
`MeshTraits.cxx`.

This section illustrates the full power of Generic Programming. This is sometimes perceived as *too much of a good thing*!

The toolkit has been designed to offer flexibility while keeping the complexity of the code to a moderate level. This is achieved in the Mesh by hiding most of its parameters and defining reasonable defaults for them.

The generic concept of a mesh integrates many different elements. It is possible in principle to use independent types for every one of such elements. The mechanism used in generic programming for specifying the many different types involved in a concept is called *traits*. They are basically the list of all types that interact with the current class.

The `itk::Mesh` is templated over three parameters. So far only two of them have been discussed, namely the `PixelType` and the `Dimension`. The third parameter is a class providing the set of traits required by the mesh. When the third parameter is omitted a default class is used. This default class is the `itk::DefaultStaticMeshTraits`. If you want to customize the types used by the mesh, the way to proceed is to modify the default traits and provide them as the third parameter of the Mesh class instantiation.

There are two ways of achieving this. The first is to use the existing DefaultStaticMeshTraits class. This class is itself templated over six parameters. Customizing those parameters could provide enough flexibility to define a very specific kind of mesh. The second way is to write a traits class from scratch, in which case the easiest way to proceed is to copy the DefaultStaticMeshTraits into another file and edit its content. Only the first approach is illustrated here. The second is discouraged unless you are familiar with Generic Programming, feel comfortable with C++ templates and have access to an abundant supply of (Columbian) coffee.

The first step in customizing the mesh is to include the header file of the Mesh and its static traits.

```
#include "itkMesh.h"
#include "itkDefaultStaticMeshTraits.h"
```

Then the MeshTraits class is instantiated by selecting the types of each one of its six template arguments. They are in order

PixelType. The type associated with every point.

PointDimension. The dimension of the space in which the mesh is embedded.

MaxTopologicalDimension. The highest dimension of the mesh cells.

CoordRepType. The type used to represent space coordinates.

InterpolationWeightType. The type used to represent interpolation weights.

CellPixelType. The type associated with every cell.

Let's define types and values for each one of those elements. For example the following code will use points in 3D space as nodes of the Mesh. The maximum dimension of the cells will be two which means that this is a 2D manifold better know as a *surface*. The data type associated with points is defined to be a four-dimensional vector. This type could represent values of membership for a four-classes segmentation method. The value selected for the cells are 4×3 matrices which could have for example the derivative of the membership values with respect to coordinates in space. Finally a `double` type is selected for representing space coordinates on the mesh points and also for the weight used for interpolating values.

```
const unsigned int PointDimension = 3;
const unsigned int MaxTopologicalDimension = 2;

typedef itk::Vector<double,4>                PixelType;
typedef itk::Matrix<double,4,3>              CellDataType;

typedef double CoordinateType;
typedef double InterpolationWeightType;

typedef itk::DefaultStaticMeshTraits<
          PixelType, PointDimension, MaxTopologicalDimension,
          CoordinateType, InterpolationWeightType, CellDataType > MeshTraits;

typedef itk::Mesh< PixelType, PointDimension, MeshTraits > MeshType;
```

The `itk::LineCell` type can now be instantiated using the traits taken from the Mesh.

```
typedef MeshType::CellType               CellType;
typedef itk::LineCell< CellType >        LineType;
```

Let's now create an Mesh and insert some points on it. Note that the dimension of the points matches the dimension of the Mesh. Here we insert a sequence of points that look like a plot of the $log()$ function.

```
MeshType::Pointer  mesh = MeshType::New();

typedef MeshType::PointType PointType;
PointType point;

const unsigned int numberOfPoints = 10;
for(unsigned int id=0; id<numberOfPoints; id++)
  {
  point[0] = 1.565;   // Initialize points here
  point[1] = 3.647;   // with arbitrary values
  point[2] = 4.129;
  mesh->SetPoint( id, point );
  }
```

A set of line cells is created and associated with the existing points by using point identifiers. In this simple case, the point identifiers can be deduced from cell identifiers since the line cells are ordered

in the same way. Note that in the code above, the values assigned to point components are arbitrary. In a more realistic example, those values would be computed from another source.

```
CellType::CellAutoPointer line;
const unsigned int numberOfCells = numberOfPoints-1;
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  line.TakeOwnership(  new LineType  );
  line->SetPointId( 0, cellId   ); // first point
  line->SetPointId( 1, cellId+1 ); // second point
  mesh->SetCell( cellId, line );   // insert the cell
  }
```

Data associated with cells is inserted in the Mesh by using the `SetCellData()` method. It requires the user to provide an identifier and the value to be inserted. The identifier should match one of the inserted cells. In this example, we simply store a CellDataType dummy variable named `value`.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  CellDataType value;
  mesh->SetCellData( cellId, value );
  }
```

Cell data can be read from the Mesh with the `GetCellData()` method. It requires the user to provide the identifier of the cell for which the data is to be retrieved. The user should provide also a valid pointer to a location where the data can be copied.

```
for(unsigned int cellId=0; cellId<numberOfCells; cellId++)
  {
  CellDataType value;
  mesh->GetCellData( cellId, &value );
  std::cout << "Cell " << cellId << " = " << value << std::endl;
  }
```

Neither `SetCellData()` or `GetCellData()` are efficient ways to access cell data. Efficient access to cell data can be achieved by using the Iterators built into the CellDataContainer.

```
typedef MeshType::CellDataContainer::ConstIterator CellDataIterator;
```

Note that the `ConstIterator` is used here because the data is only going to be read. This approach is exactly the same already illustrated for getting access to point data. The iterator to the first cell data item can be obtained with the `Begin()` method of the CellDataContainer. The past-end iterator is returned by the `End()` method. The cell data container itself can be obtained from the mesh with the method `GetCellData()`.

```
CellDataIterator cellDataIterator  = mesh->GetCellData()->Begin();
CellDataIterator end               = mesh->GetCellData()->End();
```

Finally a standard loop is used to iterate over all the cell data entries. Note the use of the `Value()` method used to get the actual value of the data entry. `PixelType` elements are returned by copy.

```
while( cellDataIterator != end )
  {
  CellDataType cellValue = cellDataIterator.Value();
  std::cout << cellValue << std::endl;
  ++cellDataIterator;
  }
```

4.3.5 Topology and the K-Complex

The source code for this section can be found in the file
`MeshKComplex.cxx`.

The `itk::Mesh` class supports the representation of formal topologies. In particular the concept of *K-Complex* can be correctly represented in the Mesh. An informal definition of K-Complex may be as follows: a K-Complex is a topological structure in which for every cell of dimension N, its boundary faces which are cells of dimension $N-1$ also belong to the structure.

This section illustrates how to instantiate a K-Complex structure using the mesh. The example structure is composed of one tetrahedron, its four triangle faces, its six line edges and its four vertices.

The header files of all the cell types involved should be loaded along with the header file of the mesh class.

```
#include "itkMesh.h"
#include "itkLineCell.h"
#include "itkTetrahedronCell.h"
```

Then the PixelType is defined and the mesh type is instantiated with it. Note that the dimension of the space is three in this case.

```
typedef float                           PixelType;
typedef itk::Mesh< PixelType, 3 >       MeshType;
```

The cell type can now be instantiated using the traits taken from the Mesh.

```
typedef MeshType::CellType                CellType;
typedef itk::VertexCell< CellType >       VertexType;
typedef itk::LineCell< CellType >         LineType;
typedef itk::TriangleCell< CellType >     TriangleType;
typedef itk::TetrahedronCell< CellType >  TetrahedronType;
```

The mesh is created and the points associated with the vertices are inserted. Note that there is an important distinction between the points in the mesh and the `itk::VertexCell` concept. A VertexCell is a cell of dimension zero. Its main difference as compared to a point is that the cell can be aware of neighborhood relationships with other cells. Points are not aware of the existence of cells. In fact, from the pure topological point of view, the coordinates of points in the mesh are completely irrelevant. They may as well be absent from the mesh structure altogether. VertexCells on the other hand are necessary to represent the full set of neighborhood relationships on the K-

Complex.

The geometrical coordinates of the nodes of a regular tetrahedron can be obtained by taking every other node from a regular cube.

```
MeshType::Pointer  mesh = MeshType::New();

MeshType::PointType   point0;
MeshType::PointType   point1;
MeshType::PointType   point2;
MeshType::PointType   point3;

point0[0] = -1; point0[1] = -1; point0[2] = -1;
point1[0] =  1; point1[1] =  1; point1[2] = -1;
point2[0] =  1; point2[1] = -1; point2[2] =  1;
point3[0] = -1; point3[1] =  1; point3[2] =  1;

mesh->SetPoint( 0, point0 );
mesh->SetPoint( 1, point1 );
mesh->SetPoint( 2, point2 );
mesh->SetPoint( 3, point3 );
```

We proceed now to create the cells, associate them with the points and insert them on the mesh. Starting with the tetrahedron we write the following code.

```
CellType::CellAutoPointer cellpointer;

cellpointer.TakeOwnership( new TetrahedronType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
cellpointer->SetPointId( 2, 2 );
cellpointer->SetPointId( 3, 3 );
mesh->SetCell( 0, cellpointer );
```

Four triangular faces are created and associated with the mesh now. The first triangle connects points 0,1,2.

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
cellpointer->SetPointId( 2, 2 );
mesh->SetCell( 1, cellpointer );
```

The second triangle connects points 0, 2, 3 .

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 2 );
cellpointer->SetPointId( 2, 3 );
mesh->SetCell( 2, cellpointer );
```

The third triangle connects points 0, 3, 1 .

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 3 );
cellpointer->SetPointId( 2, 1 );
mesh->SetCell( 3, cellpointer );
```

The fourth triangle connects points 3, 2, 1 .

```
cellpointer.TakeOwnership( new TriangleType );
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 2 );
cellpointer->SetPointId( 2, 1 );
mesh->SetCell( 4, cellpointer );
```

Note how the `CellAutoPointer` is reused every time. Reminder: the `itk::AutoPointer` loses ownership of the cell when it is passed as an argument of the `SetCell()` method. The AutoPointer is attached to a new cell by using the `TakeOwnership()` method.

The construction of the K-Complex continues now with the creation of the six lines on the tetrahedron edges.

```
cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
mesh->SetCell( 5, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 1 );
cellpointer->SetPointId( 1, 2 );
mesh->SetCell( 6, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 2 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 7, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 1 );
cellpointer->SetPointId( 1, 3 );
mesh->SetCell( 8, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 2 );
mesh->SetCell( 9, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 3 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 10, cellpointer );
```

Finally the zero dimensional cells represented by the `itk::VertexCell` are created and inserted in the mesh.

```
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 0 );
mesh->SetCell( 11, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 1 );
mesh->SetCell( 12, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 2 );
mesh->SetCell( 13, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 3 );
mesh->SetCell( 14, cellpointer );
```

At this point the Mesh contains four points and fifteen cells enumerated from 0 to 14. The points can be visited using PointContainer iterators.

```
typedef MeshType::PointsContainer::ConstIterator  PointIterator;
PointIterator pointIterator = mesh->GetPoints()->Begin();
PointIterator pointEnd      = mesh->GetPoints()->End();

while( pointIterator != pointEnd )
  {
  std::cout << pointIterator.Value() << std::endl;
  ++pointIterator;
  }
```

The cells can be visited using CellsContainer iterators.

```
typedef MeshType::CellsContainer::ConstIterator  CellIterator;

CellIterator cellIterator = mesh->GetCells()->Begin();
CellIterator cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  std::cout << cell->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

Note that cells are stored as pointer to a generic cell type that is the base class of all the specific cell classes. This means that at this level we can only have access to the virtual methods defined in the `CellType`.

The point identifiers to which the cells have been associated can be visited using iterators defined in the `CellType` trait. The following code illustrates the use of the PointIdIterators. The `PointIdsBegin()` method returns the iterator to the first point-identifier in the cell. The `PointIdsEnd()` method returns the iterator to the past-end point-identifier in the cell.

```
typedef CellType::PointIdIterator     PointIdIterator;

PointIdIterator pointIditer = cell->PointIdsBegin();
PointIdIterator pointIdend  = cell->PointIdsEnd();

while( pointIditer != pointIdend )
  {
  std::cout << *pointIditer << std::endl;
  ++pointIditer;
  }
```

Note that the point-identifier is obtained from the iterator using the more traditional `*iterator` notation instead the `Value()` notation used by cell-iterators.

Up to here, the topology of the K-Complex is not completely defined since we have only introduced the cells. ITK allows the user to define explicitly the neighborhood relationships between cells. It is clear that a clever exploration of the point identifiers could have allowed a user to figure out the neighborhood relationships. For example, two triangle cells sharing the same two point identifiers will probably be neighbor cells. Some of the drawbacks on this implicit discovery of neighborhood relationships is that it takes computing time and that some applications may not accept the same assumptions. A specific case is surgery simulation. This application typically simulates bistoury cuts in a mesh representing an organ. A small cut in the surface may be made by specifying that two triangles are not considered to be neighbors any more.

Neighborhood relationships are represented in the mesh by the notion of *BoundaryFeature*. Every cell has an internal list of cell-identifiers pointing to other cells that are considered to be its neighbors. Boundary features are classified by dimension. For example, a line will have two boundary features of dimension zero corresponding to its two vertices. A tetrahedron will have boundary features of dimension zero, one and two, corresponding to its four vertices, six edges and four triangular faces. It is up to the user to specify the connections between the cells.

Let's take in our current example the tetrahedron cell that was associated with the cell-identifier `0` and assign to it the four vertices as boundaries of dimension zero. This is done by invoking the `SetBoundaryAssignment()` method on the Mesh class.

```
MeshType::CellIdentifier cellId = 0;  // the tetrahedron

int dimension = 0;                    // vertices

MeshType::CellFeatureIdentifier featureId = 0;

mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 11 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 12 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 13 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 14 );
```

The `featureId` is simply a number associated with the sequence of the boundary cells of the same dimension in a specific cell. For example, the zero-dimensional features of a tetrahedron are its four vertices. Then the zero-dimensional feature-Ids for this cell will range from zero to three. The one-dimensional features of the tetrahedron are its six edges, hence its one-dimensional feature-Ids will

range from zero to five. The two-dimensional features of the tetrahedron are its four triangular faces. The two-dimensional feature ids will then range from zero to three. The following table summarizes the use on indices for boundary assignments.

Dimension	CellType	FeatureId range	Cell Ids
0	VertexCell	[0:3]	{11,12,13,14}
1	LineCell	[0:5]	{5,6,7,8,9,10}
2	TriangleCell	[0:3]	{1,2,3,4}

In the code example above, the values of featureId range from zero to three. The cell identifiers of the triangle cells in this example are the numbers {1,2,3,4}, while the cell identifiers of the vertex cells are the numbers {11,12,13,14}.

Let's now assign one-dimensional boundary features of the tetrahedron. Those are the line cells with identifiers {5,6,7,8,9,10}. Note that the feature identifier is reinitialized to zero since the count is independent for each dimension.

```
cellId    = 0;  // still the tetrahedron
dimension = 1;  // one-dimensional features = edges
featureId = 0;  // reinitialize the count

mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  5 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  6 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  7 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  8 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  9 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 10 );
```

Finally we assign the two-dimensional boundary features of the tetrahedron. These are the four triangular cells with identifiers {1,2,3,4}. The featureId is reset to zero since feature-Ids are independent on each dimension.

```
cellId    = 0;  // still the tetrahedron
dimension = 2;  // two-dimensional features = triangles
featureId = 0;  // reinitialize the count

mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  1 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  2 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  3 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  4 );
```

At this point we can query the tetrahedron cell for information about its boundary features. For example, the number of boundary features of each dimension can be obtained with the method `GetNumberOfBoundaryFeatures()`.

```
cellId = 0; // still the tetrahedron

MeshType::CellFeatureCount n0;  // number of zero-dimensional features
MeshType::CellFeatureCount n1;  // number of  one-dimensional features
MeshType::CellFeatureCount n2;  // number of  two-dimensional features

n0 = mesh->GetNumberOfCellBoundaryFeatures( 0, cellId );
n1 = mesh->GetNumberOfCellBoundaryFeatures( 1, cellId );
n2 = mesh->GetNumberOfCellBoundaryFeatures( 2, cellId );
```

The boundary assignments can be recovered with the method `GetBoundaryAssigment()`. For example, the zero-dimensional features of the tetrahedron can be obtained with the following code.

```
dimension = 0;
for(unsigned int b0=0; b0 < n0; b0++)
  {
  MeshType::CellIdentifier id;
  bool found = mesh->GetBoundaryAssignment( dimension, cellId, b0, &id );
  if( found ) std::cout << id << std::endl;
  }
```

The following code illustrates how to set the edge boundaries for one of the triangular faces.

```
cellId     =  2;    // one of the triangles
dimension  =  1;    // boundary edges
featureId  =  0;    // start the count of features

mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  7 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++,  9 );
mesh->SetBoundaryAssignment( dimension, cellId, featureId++, 10 );
```

4.3.6 Representing a PolyLine

The source code for this section can be found in the file `MeshPolyLine.cxx`.

This section illustrates how to represent a classical *PolyLine* structure using the `itk::Mesh`

A PolyLine only involves zero and one dimensional cells, which are represented by the `itk::VertexCell` and the `itk::LineCell`.

```
#include "itkMesh.h"
#include "itkLineCell.h"
```

Then the PixelType is defined and the mesh type is instantiated with it. Note that the dimension of the space is two in this case.

```
typedef float                           PixelType;
typedef itk::Mesh< PixelType, 2 >       MeshType;
```

The cell type can now be instantiated using the traits taken from the Mesh.

```
typedef MeshType::CellType                CellType;
typedef itk::VertexCell< CellType >       VertexType;
typedef itk::LineCell< CellType >         LineType;
```

The mesh is created and the points associated with the vertices are inserted. Note that there is an important distinction between the points in the mesh and the `itk::VertexCell` concept. A VertexCell is a cell of dimension zero. Its main difference as compared to a point is that the cell can be aware of neighborhood relationships with other cells. Points are not aware of the existence of cells. In fact, from the pure topological point of view, the coordinates of points in the mesh are completely irrelevant. They may as well be absent from the mesh structure altogether. VertexCells on the other hand are necessary to represent the full set of neighborhood relationships on the Polyline.

In this example we create a polyline connecting the four vertices of a square by using three of the square sides.

```
MeshType::Pointer  mesh = MeshType::New();

MeshType::PointType   point0;
MeshType::PointType   point1;
MeshType::PointType   point2;
MeshType::PointType   point3;

point0[0] = -1; point0[1] = -1;
point1[0] =  1; point1[1] = -1;
point2[0] =  1; point2[1] =  1;
point3[0] = -1; point3[1] =  1;

mesh->SetPoint( 0, point0 );
mesh->SetPoint( 1, point1 );
mesh->SetPoint( 2, point2 );
mesh->SetPoint( 3, point3 );
```

We proceed now to create the cells, associate them with the points and insert them on the mesh.

```
CellType::CellAutoPointer cellpointer;

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 0 );
cellpointer->SetPointId( 1, 1 );
mesh->SetCell( 0, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 1 );
cellpointer->SetPointId( 1, 2 );
mesh->SetCell( 1, cellpointer );

cellpointer.TakeOwnership( new LineType );
cellpointer->SetPointId( 0, 2 );
cellpointer->SetPointId( 1, 0 );
mesh->SetCell( 2, cellpointer );
```

Finally the zero dimensional cells represented by the itk::VertexCell are created and inserted in the mesh.

```
cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 0 );
mesh->SetCell( 3, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 1 );
mesh->SetCell( 4, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 2 );
mesh->SetCell( 5, cellpointer );

cellpointer.TakeOwnership( new VertexType );
cellpointer->SetPointId( 0, 3 );
mesh->SetCell( 6, cellpointer );
```

At this point the Mesh contains four points and three cells. The points can be visited using PointContainer iterators.

```
typedef MeshType::PointsContainer::ConstIterator  PointIterator;
PointIterator pointIterator = mesh->GetPoints()->Begin();
PointIterator pointEnd      = mesh->GetPoints()->End();

while( pointIterator != pointEnd )
  {
  std::cout << pointIterator.Value() << std::endl;
  ++pointIterator;
  }
```

The cells can be visited using CellsContainer iterators.

```
typedef MeshType::CellsContainer::ConstIterator  CellIterator;

CellIterator cellIterator = mesh->GetCells()->Begin();
CellIterator cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  std::cout << cell->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

Note that cells are stored as pointer to a generic cell type that is the base class of all the specific cell classes. This means that at this level we can only have access to the virtual methods defined in the `CellType`.

The point identifiers to which the cells have been associated can be visited using iterators defined in the `CellType` trait. The following code illustrates the use of the PointIdIterator. The `PointIdsBegin()` method returns the iterator to the first point-identifier in the cell. The

PointIdsEnd() method returns the iterator to the past-end point-identifier in the cell.

```
typedef CellType::PointIdIterator     PointIdIterator;

PointIdIterator pointIditer = cell->PointIdsBegin();
PointIdIterator pointIdend  = cell->PointIdsEnd();

while( pointIditer != pointIdend )
  {
  std::cout << *pointIditer << std::endl;
  ++pointIditer;
  }
```

Note that the point-identifier is obtained from the iterator using the more traditional *iterator notation instead the Value() notation used by cell-iterators.

4.3.7 Simplifying Mesh Creation

The source code for this section can be found in the file AutomaticMesh.cxx.

The itk::Mesh class is extremely general and flexible, but there is some cost to convenience. If convenience is exactly what you need, then it is possible to get it, in exchange for some of that flexibility, by means of the itk::AutomaticTopologyMeshSource class. This class automatically generates an explicit K-Complex, based on the cells you add. It explicitly includes all boundary information, so that the resulting mesh can be easily traversed. It merges all shared edges, vertices, and faces, so no geometric feature appears more than once.

This section shows how you can use the AutomaticTopologyMeshSource to instantiate a mesh representing a K-Complex. We will first generate the same tetrahedron from Section 4.3.5, after which we will add a hollow one to illustrate some additional features of the mesh source.

The header files of all the cell types involved should be loaded along with the header file of the mesh class.

```
#include "itkTriangleCell.h"
#include "itkAutomaticTopologyMeshSource.h"
```

We then define the necessary types and instantiate the mesh source. Two new types are IdentifierType and IdentifierArrayType. Every cell in a mesh has an identifier, whose type is determined by the mesh traits. AutomaticTopologyMeshSource requires that the identifier type of all vertices and cells be unsigned long, which is already the default. However, if you created a new mesh traits class to use string tags as identifiers, the resulting mesh would not be compatible with itk::AutomaticTopologyMeshSource. An IdentifierArrayType is simply an itk::Array of IdentifierType objects.

```
typedef float                              PixelType;
typedef itk::Mesh< PixelType, 3 >          MeshType;

typedef MeshType::PointType                PointType;

typedef itk::AutomaticTopologyMeshSource< MeshType >   MeshSourceType;
typedef MeshSourceType::IdentifierArrayType            IdentifierArrayType;

MeshSourceType::Pointer meshSource;

meshSource = MeshSourceType::New();
```

Now let us generate the tetrahedron. The following line of code generates all the vertices, edges, and faces, along with the tetrahedral solid, and adds them to the mesh along with the connectivity information.

```
meshSource->AddTetrahedron(
  meshSource->AddPoint( -1, -1, -1 ),
  meshSource->AddPoint(  1,  1, -1 ),
  meshSource->AddPoint(  1, -1,  1 ),
  meshSource->AddPoint( -1,  1,  1 )
);
```

The function `AutomaticTopologyMeshSource::AddTetrahedron()` takes point identifiers as parameters; the identifiers must correspond to points that have already been added. `AutomaticTopologyMeshSource::AddPoint()` returns the appropriate identifier type for the point being added. It first checks to see if the point is already in the mesh. If so, it returns the ID of the point in the mesh, and if not, it generates a new unique ID, adds the point with that ID, and returns the ID.

Actually, `AddTetrahedron()` behaves in the same way. If the tetrahedron has already been added, it leaves the mesh unchanged and returns the ID that the tetrahedron already has. If not, it adds the tetrahedron (and all its faces, edges, and vertices), and generates a new ID, which it returns.

It is also possible to add all the points first, and then add a number of cells using the point IDs directly. This approach corresponds with the way the data is stored in many file formats for 3D polygonal models.

First we add the points (in this case the vertices of a larger tetrahedron). This example also illustrates that `AddPoint()` can take a single `PointType` as a parameter if desired, rather than a sequence of floats. Another possibility (not illustrated) is to pass in a C-style array.

```
PointType p;
IdentifierArrayType idArray( 4 );

p[ 0 ] = -2;
p[ 1 ] = -2;
p[ 2 ] = -2;
idArray[ 0 ] = meshSource->AddPoint( p );

p[ 0 ] =  2;
p[ 1 ] =  2;
p[ 2 ] = -2;
idArray[ 1 ] = meshSource->AddPoint( p );

p[ 0 ] =  2;
p[ 1 ] = -2;
p[ 2 ] =  2;
idArray[ 2 ] = meshSource->AddPoint( p );

p[ 0 ] = -2;
p[ 1 ] =  2;
p[ 2 ] =  2;
idArray[ 3 ] = meshSource->AddPoint( p );
```

Now we add the cells. This time we are just going to create the boundary of a tetrahedron, so we must add each face separately.

```
meshSource->AddTriangle( idArray[0], idArray[1], idArray[2] );
meshSource->AddTriangle( idArray[1], idArray[2], idArray[3] );
meshSource->AddTriangle( idArray[2], idArray[3], idArray[0] );
meshSource->AddTriangle( idArray[3], idArray[0], idArray[1] );
```

Actually, we could have called, e.g., `AddTriangle( 4, 5, 6 )`, since IDs are assigned sequentially starting at zero, and `idArray[0]` contains the ID for the fifth point added. But you should only do this if you are confident that you know what the IDs are. If you add the same point twice and don't realize it, your count will differ from that of the mesh source.

You may be wondering what happens if you call, say, `AddEdge(0, 1)` followed by `AddEdge(1, 0)`. The answer is that they do count as the same edge, and so only one edge is added. The order of the vertices determines an orientation, and the first orientation specified is the one that is kept.

Once you have built the mesh you want, you can access it by calling `GetOutput()`. Here we send it to `cout`, which prints some summary data for the mesh.

In contrast to the case with typical filters, `GetOutput()` does not trigger an update process. The mesh is always maintained in a valid state as cells are added, and can be accessed at any time. It would, however, be a mistake to modify the mesh by some other means until AutomaticTopologyMeshSource is done with it, since the mesh source would then have an inaccurate record of which points and cells are currently in the mesh.

4.3.8 Iterating Through Cells

The source code for this section can be found in the file `MeshCellsIteration.cxx`.

Cells are stored in the `itk::Mesh` as pointers to a generic cell `itk::CellInterface`. This implies that only the virtual methods defined on this base cell class can be invoked. In order to use methods that are specific to each cell type it is necessary to down-cast the pointer to the actual type of the cell. This can be done safely by taking advantage of the `GetType()` method that allows to identify the actual type of a cell.

Let's start by assuming a mesh defined with one tetrahedron and all its boundary faces. That is, four triangles, six edges and four vertices.

The cells can be visited using CellsContainer iterators . The iterator `Value()` corresponds to a raw pointer to the `CellType` base class.

```
typedef MeshType::CellsContainer::ConstIterator  CellIterator;

CellIterator cellIterator = mesh->GetCells()->Begin();
CellIterator cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  std::cout << cell->GetNumberOfPoints() << std::endl;
  ++cellIterator;
  }
```

In order to perform down-casting in a safe manner, the cell type can be queried first using the `GetType()` method. Codes for the cell types have been defined with an `enum` type on the `itkCellInterface.h` header file. These codes are :

- VERTEX_CELL
- LINE_CELL
- TRIANGLE_CELL
- QUADRILATERAL_CELL
- POLYGON_CELL
- TETRAHEDRON_CELL
- HEXAHEDRON_CELL
- QUADRATIC_EDGE_CELL
- QUADRATIC_TRIANGLE_CELL

The method `GetType()` returns one of these codes. It is then possible to test the type of the cell before down-casting its pointer to the actual type. For example, the following code visits all the cells in the mesh and tests which ones are actually of type `LINE_CELL`. Only those cells are down-casted to `LineType` cells and a method specific for the `LineType` is invoked.

```
cellIterator = mesh->GetCells()->Begin();
cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  if( cell->GetType() == CellType::LINE_CELL )
    {
    LineType * line = static_cast<LineType *>( cell );
    std::cout << "dimension = " << line->GetDimension();
    std::cout << " # points = " << line->GetNumberOfPoints();
    std::cout << std::endl;
    }
  ++cellIterator;
  }
```

In order to perform different actions on different cell types a `switch` statement can be used with cases for every cell type. The following code illustrates an iteration over the cells and the invocation of different methods on each cell type.

```
cellIterator = mesh->GetCells()->Begin();
cellEnd      = mesh->GetCells()->End();

while( cellIterator != cellEnd )
  {
  CellType * cell = cellIterator.Value();
  switch( cell->GetType() )
    {
    case CellType::VERTEX_CELL:
      {
      std::cout << "VertexCell : " << std::endl;
      VertexType * line = dynamic_cast<VertexType *>( cell );
      std::cout << "dimension = " << line->GetDimension()      << std::endl;
      std::cout << "# points  = " << line->GetNumberOfPoints() << std::endl;
      break;
      }
    case CellType::LINE_CELL:
      {
      std::cout << "LineCell : " << std::endl;
      LineType * line = dynamic_cast<LineType *>( cell );
      std::cout << "dimension = " << line->GetDimension()      << std::endl;
      std::cout << "# points  = " << line->GetNumberOfPoints() << std::endl;
      break;
      }
    case CellType::TRIANGLE_CELL:
      {
      std::cout << "TriangleCell : " << std::endl;
      TriangleType * line = dynamic_cast<TriangleType *>( cell );
      std::cout << "dimension = " << line->GetDimension()      << std::endl;
      std::cout << "# points  = " << line->GetNumberOfPoints() << std::endl;
      break;
      }
    default:
      {
      std::cout << "Cell with more than three points" << std::endl;
      std::cout << "dimension = " << cell->GetDimension()      << std::endl;
      std::cout << "# points  = " << cell->GetNumberOfPoints() << std::endl;
      break;
      }
    }
  ++cellIterator;
  }
```

4.3.9 Visiting Cells

The source code for this section can be found in the file `MeshCellVisitor.cxx`.

In order to facilitate access to particular cell types, a convenience mechanism has been built-in on the `itk::Mesh`. This mechanism is based on the *Visitor Pattern* presented in [3]. The visitor pattern is designed to facilitate the process of walking through an heterogeneous list of objects sharing a

common base class.

The first requirement for using the `CellVisitor` mechanism it to include the `CellInterfaceVisitor` header file.

```
#include "itkCellInterfaceVisitor.h"
```

The typical mesh types are now declared.

```
  typedef float                            PixelType;
  typedef itk::Mesh< PixelType, 3 >        MeshType;

  typedef MeshType::CellType               CellType;

  typedef itk::VertexCell< CellType >      VertexType;
  typedef itk::LineCell< CellType >        LineType;
  typedef itk::TriangleCell< CellType >    TriangleType;
  typedef itk::TetrahedronCell< CellType > TetrahedronType;
```

Then, a custom CellVisitor class should be declared. In this particular example, the visitor class is intended to act only on `TriangleType` cells. The only requirement on the declaration of the visitor class is that it must provide a method named `Visit()`. This method expects as arguments a cell identifier and a pointer to the *specific* cell type for which this visitor is intended. Nothing prevents a visitor class from providing `Visit()` methods for several different cell types. The multiple methods will be differentiated by the natural C++ mechanism of function overload. The following code illustrates a minimal cell visitor class.

```
class CustomTriangleVisitor
{
public:
  typedef itk::TriangleCell<CellType>      TriangleType;
  void Visit(unsigned long cellId, TriangleType * t )
    {
    std::cout << "Cell # " << cellId << " is a TriangleType ";
    std::cout << t->GetNumberOfPoints() << std::endl;
    }
  CustomTriangleVisitor() {}
  virtual ~CustomTriangleVisitor() {}
};
```

This newly defined class will now be used to instantiate a cell visitor. In this particular example we create a class `CustomTriangleVisitor` which will be invoked each time a triangle cell is found while the mesh iterates over the cells.

```
  typedef itk::CellInterfaceVisitorImplementation<
                            PixelType,
                            MeshType::CellTraits,
                            TriangleType,
                            CustomTriangleVisitor
                                    > TriangleVisitorInterfaceType;
```

Note that the actual `CellInterfaceVisitorImplementation` is templated over the PixelType, the CellTraits, the CellType to be visited and the Visitor class that defines with will be done with the cell.

A visitor implementation class can now be created using the normal invocation to its `New()` method and assigning the result to a `itk::SmartPointer`.

```
TriangleVisitorInterfaceType::Pointer  triangleVisitor =
                                TriangleVisitorInterfaceType::New();
```

Many different visitors can be configured in this way. The set of all visitors can be registered with the MultiVisitor class provided for the mesh. An instance of the MultiVisitor class will walk through the cells and delegate action to every registered visitor when the appropriate cell type is encountered.

```
typedef CellType::MultiVisitor CellMultiVisitorType;
CellMultiVisitorType::Pointer multiVisitor = CellMultiVisitorType::New();
```

The visitor is registered with the Mesh using the `AddVisitor()` method.

```
multiVisitor->AddVisitor( triangleVisitor );
```

Finally, the iteration over the cells is triggered by calling the method `Accept()` on the `itk::Mesh`.

```
mesh->Accept( multiVisitor );
```

The `Accept()` method will iterate over all the cells and for each one will invite the MultiVisitor to attempt an action on the cell. If no visitor is interested on the current cell type the cell is just ignored and skipped.

MultiVisitors make it possible to add behavior to the cells without having to create new methods on the cell types or creating a complex visitor class that knows about every CellType.

4.3.10 More on Visiting Cells

The source code for this section can be found in the file `MeshCellVisitor2.cxx`.

The following section illustrates a realistic example of the use of Cell visitors on the `itk::Mesh`. A set of different visitors is defined here, each visitor associated with a particular type of cell. All the visitors are registered with a MultiVisitor class which is passed to the mesh.

The first step is to include the `CellInterfaceVisitor` header file.

```
#include "itkCellInterfaceVisitor.h"
```

The typical mesh types are now declared.

```
  typedef float                            PixelType;
  typedef itk::Mesh< PixelType, 3 >        MeshType;

  typedef MeshType::CellType               CellType;

  typedef itk::VertexCell< CellType >      VertexType;
  typedef itk::LineCell< CellType >        LineType;
  typedef itk::TriangleCell< CellType >    TriangleType;
  typedef itk::TetrahedronCell< CellType > TetrahedronType;
```

Then, custom CellVisitor classes should be declared. The only requirement on the declaration of each visitor class is to provide a method named `Visit()`. This method expects as arguments a cell identifier and a pointer to the *specific* cell type for which this visitor is intended.

The following Vertex visitor simply prints out the identifier of the point with which the cell is associated. Note that the cell uses the method `GetPointId()` without any arguments. This method is only defined on the VertexCell.

```
class CustomVertexVisitor
{
public:
  void Visit(unsigned long cellId, VertexType * t )
    {
    std::cout << "cell " << cellId << " is a Vertex " << std::endl;
    std::cout << "    associated with point id = ";
    std::cout << t->GetPointId() << std::endl;
    }
  virtual ~CustomVertexVisitor() {}
};
```

The following Line visitor computes the length of the line. Note that this visitor is slightly more complicated since it needs to get access to the actual mesh in order to get point coordinates from the point identifiers returned by the line cell. This is done by holding a pointer to the mesh and querying the mesh each time point coordinates are required. The mesh pointer is set up in this case with the `SetMesh()` method.

```
class CustomLineVisitor
{
public:
  CustomLineVisitor():m_Mesh( 0 ) {}
  virtual ~CustomLineVisitor() {}

  void SetMesh( MeshType * mesh ) { m_Mesh = mesh; }

  void Visit(unsigned long cellId, LineType * t )
    {
    std::cout << "cell " << cellId << " is a Line " << std::endl;
    LineType::PointIdIterator pit = t->PointIdsBegin();
    MeshType::PointType p0;
    MeshType::PointType p1;
    m_Mesh->GetPoint( *pit++, &p0 );
    m_Mesh->GetPoint( *pit++, &p1 );
    const double length = p0.EuclideanDistanceTo( p1 );
    std::cout << " length = " << length << std::endl;
    }

private:
  MeshType::Pointer m_Mesh;
};
```

The Triangle visitor below prints out the identifiers of its points. Note the use of the `PointIdIterator` and the `PointIdsBegin()` and `PointIdsEnd()` methods.

```
class CustomTriangleVisitor
{
public:
  void Visit(unsigned long cellId, TriangleType * t )
    {
    std::cout << "cell " << cellId << " is a Triangle " << std::endl;
    LineType::PointIdIterator pit = t->PointIdsBegin();
    LineType::PointIdIterator end = t->PointIdsEnd();
    while( pit != end )
      {
      std::cout << "  point id = " << *pit << std::endl;
      ++pit;
      }
    }
  virtual ~CustomTriangleVisitor() {}
};
```

The TetrahedronVisitor below simply returns the number of faces on this figure. Note that `GetNumberOfFaces()` is a method exclusive of 3D cells.

```
class CustomTetrahedronVisitor
{
public:
  void Visit(unsigned long cellId, TetrahedronType * t )
    {
    std::cout << "cell " << cellId << " is a Tetrahedron " << std::endl;
    std::cout << "  number of faces = ";
    std::cout << t->GetNumberOfFaces() << std::endl;
    }
  virtual ~CustomTetrahedronVisitor() {}
};
```

With the cell visitors we proceed now to instantiate CellVisitor implementations. The visitor classes defined above are used as template arguments of the cell visitor implementation.

```
  typedef itk::CellInterfaceVisitorImplementation<
      PixelType, MeshType::CellTraits, VertexType,
      CustomVertexVisitor > VertexVisitorInterfaceType;

  typedef itk::CellInterfaceVisitorImplementation<
      PixelType, MeshType::CellTraits, LineType,
      CustomLineVisitor > LineVisitorInterfaceType;

  typedef itk::CellInterfaceVisitorImplementation<
      PixelType, MeshType::CellTraits, TriangleType,
      CustomTriangleVisitor > TriangleVisitorInterfaceType;

  typedef itk::CellInterfaceVisitorImplementation<
      PixelType, MeshType::CellTraits, TetrahedronType,
      CustomTetrahedronVisitor > TetrahedronVisitorInterfaceType;
```

Note that the actual `CellInterfaceVisitorImplementation` is templated over the PixelType, the CellTraits, the CellType to be visited and the Visitor class defining what to do with the cell.

A visitor implementation class can now be created using the normal invocation to its `New()` method and assigning the result to a `itk::SmartPointer`.

```
  VertexVisitorInterfaceType::Pointer  vertexVisitor =
                                VertexVisitorInterfaceType::New();

  LineVisitorInterfaceType::Pointer  lineVisitor =
                                LineVisitorInterfaceType::New();

  TriangleVisitorInterfaceType::Pointer  triangleVisitor =
                                TriangleVisitorInterfaceType::New();

  TetrahedronVisitorInterfaceType::Pointer  tetrahedronVisitor =
                                TetrahedronVisitorInterfaceType::New();
```

Remember that the LineVisitor requires the pointer to the mesh object since it needs to get access to actual point coordinates. This is done by invoking the `SetMesh()` method defined above.

```
  lineVisitor->SetMesh( mesh );
```

Looking carefully you will notice that the SetMesh() method is declared in CustomLineVisitor but we are invoking it on LineVisitorInterfaceType. This is possible thanks to the way in which the VisitorInterfaceImplementation is defined. This class derives from the visitor type provided by the user as the fourth template parameter. LineVisitorInterfaceType is then a derived class of CustomLineVisitor.

The set of visitors should now be registered with the MultiVisitor class that will walk through the cells and delegate action to every registered visitor when the appropriate cell type is encountered. The following lines create a MultiVisitor object.

```
typedef CellType::MultiVisitor CellMultiVisitorType;
CellMultiVisitorType::Pointer multiVisitor = CellMultiVisitorType::New();
```

Every visitor implementation is registered with the Mesh using the AddVisitor() method.

```
multiVisitor->AddVisitor( vertexVisitor      );
multiVisitor->AddVisitor( lineVisitor        );
multiVisitor->AddVisitor( triangleVisitor    );
multiVisitor->AddVisitor( tetrahedronVisitor );
```

Finally, the iteration over the cells is triggered by calling the method Accept() on the Mesh class.

```
mesh->Accept( multiVisitor );
```

The Accept() method will iterate over all the cells and for each one will invite the MultiVisitor to attempt an action on the cell. If no visitor is interested on the current cell type, the cell is just ignored and skipped.

4.4 Path

4.4.1 Creating a PolyLineParametricPath

The source code for this section can be found in the file PolyLineParametricPath1.cxx.

This example illustrates how to use the itk::PolyLineParametricPath. This class will typically be used for representing in a concise way the output of an image segmentation algorithm in 2D. The PolyLineParametricPath however could also be used for representing any open or close curve in N-Dimensions as a linear piece-wise approximation.

First, the header file of the PolyLineParametricPath class must be included.

```
#include "itkPolyLineParametricPath.h"
```

The path is instantiated over the dimension of the image. In this example the image and path are two-dimensional.

```
const unsigned int Dimension = 2;

typedef itk::Image< unsigned char, Dimension > ImageType;

typedef itk::PolyLineParametricPath< Dimension > PathType;
```

```
ImageType::ConstPointer image = reader->GetOutput();
PathType::Pointer path = PathType::New();
path->Initialize();

typedef PathType::ContinuousIndexType    ContinuousIndexType;
ContinuousIndexType cindex;

typedef ImageType::PointType             ImagePointType;
ImagePointType origin = image->GetOrigin();

ImageType::SpacingType spacing = image->GetSpacing();
ImageType::SizeType    size    = image->GetBufferedRegion().GetSize();

ImagePointType point;

point[0] = origin[0] + spacing[0] * size[0];
point[1] = origin[1] + spacing[1] * size[1];

image->TransformPhysicalPointToContinuousIndex( origin, cindex );
path->AddVertex( cindex );
image->TransformPhysicalPointToContinuousIndex( point, cindex );
path->AddVertex( cindex );
```

4.5 Containers

The source code for this section can be found in the file `TreeContainer.cxx`.

This example shows how to use the `itk::TreeContainer` and the associated TreeIterators. The `itk::TreeContainer` implements the notion of tree and is templated over the type of node so it can virtually handle any objects. Each node is supposed to have only one parent so no cycle is present in the tree. No checking is done to ensure a cycle-free tree.

Let's begin by including the appropriate header file.

```
#include "itkTreeContainer.h"
#include "itkTreeContainer.h"
#include "itkChildTreeIterator.h"
#include "itkLeafTreeIterator.h"
#include "itkLevelOrderTreeIterator.h"
#include "itkInOrderTreeIterator.h"
#include "itkPostOrderTreeIterator.h"
#include "itkRootTreeIterator.h"
#include "itkTreeIteratorClone.h"
```

First, we create a tree of integers. The TreeContainer is templated over the type of nodes.

```
  typedef int                          NodeType;
  typedef itk::TreeContainer<NodeType> TreeType;
  TreeType::Pointer tree = TreeType::New();
```

Next we set the value of the root node using `SetRoot()`.

```
  tree->SetRoot(0);
```

Then we use the `Add()` function to add nodes to the tree The first argument is the value of the new node and the second argument is the value of the parent node. If two nodes have the same values then the first one is picked. In this particular case it is better to use an iterator to fill the tree.

```
  tree->Add(1,0);
  tree->Add(2,0);
  tree->Add(3,0);
  tree->Add(4,2);
  tree->Add(5,2);
  tree->Add(6,5);
  tree->Add(7,1);
```

We define an `itk::LevelOrderTreeIterator` to parse the tree in level order. This particular iterator takes three arguments. The first one is the actual tree to be parsed, the second one is the maximum depth level and the third one is the starting node. The `GetNode()` function return a node given its value. Once again the first node that corresponds to the value is returned.

```
  itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10,tree->GetNode(2));
  levelIt.GoToBegin();
  while(!levelIt.IsAtEnd())
    {
    std::cout << levelIt.Get()
              << " ("<< levelIt.GetLevel()
              << ")" << std::endl;
    ++levelIt;
    }
  std::cout << std::endl;
```

The TreeIterators have useful functions to test the property of the current pointed node. Among these functions: `IsLeaf` returns true if the current node is a leaf, `IsRoot` returns true if the node is a root, `HasParent` returns true if the node has a parent and `CountChildren` returns the number of

children for this particular node.

```
levelIt.IsLeaf();
levelIt.IsRoot();
levelIt.HasParent();
levelIt.CountChildren();
```

The itk::ChildTreeIterator provides another way to iterate through a tree by listing all the children of a node.

```
itk::ChildTreeIterator<TreeType> childIt(tree);
childIt.GoToBegin();
while(!childIt.IsAtEnd())
  {
  std::cout << childIt.Get() << std::endl;
  ++childIt;
  }
std::cout << std::endl;
```

The GetType() function returns the type of iterator used. The list of enumerated types is as follow: PREORDER, INORDER, POSTORDER, LEVELORDER, CHILD, ROOT and LEAF.

```
if(childIt.GetType() != itk::TreeIteratorBase<TreeType>::CHILD)
  {
  std::cout << "[FAILURE]" << std::endl;
  return EXIT_FAILURE;
  }
```

Every TreeIterator has a Clone() function which returns a copy of the current iterator. Note that the user should delete the created iterator by hand.

```
childIt.GoToParent();
itk::TreeIteratorBase<TreeType>* childItClone = childIt.Clone();
delete childItClone;
```

The itk::LeafTreeIterator iterates through the leaves of the tree.

```
itk::LeafTreeIterator<TreeType> leafIt(tree);
leafIt.GoToBegin();
while(!leafIt.IsAtEnd())
  {
  std::cout << leafIt.Get() << std::endl;
  ++leafIt;
  }
std::cout << std::endl;
```

The itk::InOrderTreeIterator iterates through the tree in the order from left to right.

```
itk::InOrderTreeIterator<TreeType> InOrderIt(tree);
InOrderIt.GoToBegin();
while(!InOrderIt.IsAtEnd())
  {
  std::cout << InOrderIt.Get() << std::endl;
  ++InOrderIt;
  }
std::cout << std::endl;
```

The itk::PreOrderTreeIterator iterates through the tree from left to right but do a depth first search.

```
itk::PreOrderTreeIterator<TreeType> PreOrderIt(tree);
PreOrderIt.GoToBegin();
while(!PreOrderIt.IsAtEnd())
  {
  std::cout << PreOrderIt.Get() << std::endl;
  ++PreOrderIt;
  }
std::cout << std::endl;
```

The itk::PostOrderTreeIterator iterates through the tree from left to right but goes from the leaves to the root in the search.

```
itk::PostOrderTreeIterator<TreeType> PostOrderIt(tree);
PostOrderIt.GoToBegin();
while(!PostOrderIt.IsAtEnd())
  {
  std::cout << PostOrderIt.Get() << std::endl;
  ++PostOrderIt;
  }
std::cout << std::endl;
```

The itk::RootTreeIterator goes from one node to the root. The second arguments is the starting node. Here we go from the leaf node (value = 6) up to the root.

```
itk::RootTreeIterator<TreeType> RootIt(tree,tree->GetNode(6));
RootIt.GoToBegin();
while(!RootIt.IsAtEnd())
  {
  std::cout << RootIt.Get() << std::endl;
  ++RootIt;
  }
std::cout << std::endl;
```

All the nodes of the tree can be removed by using the `Clear()` function.

```
tree->Clear();
```

We show how to use a TreeIterator to form a tree by creating nodes. The `Add()` function is used to add a node and put a value on it. The `GoToChild()` is used to jump to a node.

```
itk::PreOrderTreeIterator<TreeType> PreOrderIt2(tree);
PreOrderIt2.Add(0);
PreOrderIt2.Add(1);
PreOrderIt2.Add(2);
PreOrderIt2.Add(3);
PreOrderIt2.GoToChild(2);
PreOrderIt2.Add(4);
PreOrderIt2.Add(5);
```

The itk::TreeIteratorClone can be used to have a generic copy of an iterator.

```
typedef itk::TreeIteratorBase<TreeType>      IteratorType;
typedef itk::TreeIteratorClone<IteratorType> IteratorCloneType;
itk::PreOrderTreeIterator<TreeType> anIterator(tree);
IteratorCloneType aClone = anIterator;
```

CHAPTER

FIVE

SPATIAL OBJECTS

This chapter introduces the basic classes that describe `itk::SpatialObject`s.

5.1 Introduction

We promote the philosophy that many of the goals of medical image processing are more effectively addressed if we consider them in the broader context of object processing. ITK's Spatial Object class hierarchy provides a consistent API for querying, manipulating, and interconnecting objects in physical space. Via this API, methods can be coded to be invariant to the data structure used to store the objects being processed. By abstracting the representations of objects to support their representation by data structures other than images, a broad range of medical image analysis research is supported; key examples are described in the following.

Model-to-image registration. A mathematical instance of an object can be registered with an image to localize the instance of that object in the image. Using SpatialObjects, mutual information, cross-correlation, and boundary-to-image metrics can be applied without modification to perform spatial object-to-image registration.

Model-to-model registration. Iterative closest point, landmark, and surface distance minimization methods can be used with any ITK transform, to rigidly and non-rigidly register image, FEM, and Fourier descriptor-based representations of objects as SpatialObjects.

Atlas formation. Collections of images or SpatialObjects can be integrated to represent expected object characteristics and their common modes of variation. Labels can be associated with the objects of an atlas.

Storing segmentation results from one or multiple scans. Results of segmentations are best stored in physical/world coordinates so that they can be combined and compared with other segmentations from other images taken at other resolutions. Segmentation results from hand drawn contours, pixel labelings, or model-to-image registrations are treated consistently.

Capturing functional and logical relationships between objects. SpatialObjects can have parent and children objects. Queries made of an object (such as to determine if a point is inside of the object) can be made to integrate the responses from the children object. Transformations applied to a parent can also be propagated to the children. Thus, for example, when a liver model is moved, its vessels move with it.

Conversion to and from images. Basic functions are provided to render any SpatialObject (or collection of SpatialObjects) into an image.

IO. SpatialObject reading and writing to disk is independent of the SpatialObject class hierarchy. Meta object IO (through `itk::MetaImageIO`) methods are provided, and others are easily defined.

Tubes, blobs, images, surfaces. Are a few of the many SpatialObject data containers and types provided. New types can be added, generally by only defining one or two member functions in a derived class.

In the remainder of this chapter several examples are used to demonstrate the many spatial objects found in ITK and how they can be organized into hierarchies using `itk::SceneSpatialObject`. Further the examples illustrate how to use SpatialObject transformations to control and calculate the position of objects in space.

5.2 Hierarchy

Spatial objects can be combined to form a hierarchy as a tree. By design, a SpatialObject can have one parent and only one. Moreover, each transform is stored within each object, therefore the hierarchy cannot be described as a Directed Acyclic Graph (DAG) but effectively as a tree. The user is responsible for maintaining the tree structure, no checking is done to ensure a cycle-free tree.

The source code for this section can be found in the file `SpatialObjectHierarchy.cxx`.

This example describes how `itk::SpatialObject` can form a hierarchy. This first example also shows how to create and manipulate spatial objects.

```
#include "itkSpatialObject.h"
```

First, we create two spatial objects and give them the names `First Object` and `Second Object`, respectively.

```
typedef itk::SpatialObject<3> SpatialObjectType;

SpatialObjectType::Pointer object1 = SpatialObjectType ::New();
object1->GetProperty()->SetName("First Object");

SpatialObjectType::Pointer object2 = SpatialObjectType ::New();
object2->GetProperty()->SetName("Second Object");
```

We then add the second object to the first one by using the `AddSpatialObject()` method. As a result `object2` becomes a child of object1.

```
object1->AddSpatialObject(object2);
```

We can query if an object has a parent by using the HasParent() method. If it has one, the `GetParent()` method returns a constant pointer to the parent. In our case, if we ask the parent's name of the object2 we should obtain: `First Object`.

```
if(object2->HasParent())
  {
  std::cout << "Name of the parent of the object2: ";
  std::cout << object2->GetParent()->GetProperty()->GetName() << std::endl;
  }
```

To access the list of children of the object, the `GetChildren()` method returns a pointer to the (STL) list of children.

```
SpatialObjectType::ChildrenListType * childrenList = object1->GetChildren();
std::cout << "object1 has " << childrenList->size() << " child" << std::endl;

SpatialObjectType::ChildrenListType::const_iterator it
                                                    = childrenList->begin();
while(it != childrenList->end())
  {
  std::cout << "Name of the child of the object 1: ";
  std::cout << (*it)->GetProperty()->GetName() << std::endl;
  it++;
  }
```

Do NOT forget to delete the list of children since the `GetChildren()` function creates an internal list.

```
delete childrenList;
```

An object can also be removed by using the `RemoveSpatialObject()` method.

```
object1->RemoveSpatialObject(object2);
```

We can query the number of children an object has with the `GetNumberOfChildren()` method.

```
  std::cout << "Number of children for object1: ";
  std::cout << object1->GetNumberOfChildren() << std::endl;
```

The `Clear()` method erases all the information regarding the object as well as the data. This method is usually overloaded by derived classes.

```
  object1->Clear();
```

The output of this first example looks like the following:

```
Name of the parent of the object2: First Object
object1 has 1 child
Name of the child of the object 1: Second Object
Number of children for object1: 0
```

5.3 SpatialObject Tree Container

The source code for this section can be found in the file `SpatialObjectTreeContainer.cxx`.

This example describes how to use the `itk::SpatialObjectTreeContainer` to form a hierarchy of SpatialObjects. First we include the appropriate header file.

```
#include "itkSpatialObjectTreeContainer.h"
```

Next we define the type of node and the type of tree we plan to use. Both are templated over the dimensionality of the space. Let's create a 2-dimensional tree.

```
  typedef itk::GroupSpatialObject< 2 >         NodeType;
  typedef itk::SpatialObjectTreeContainer< 2 > TreeType;
```

Then, we can create three nodes and set their corresponding identification numbers (using `SetId`).

```
  NodeType::Pointer object0 = NodeType::New();
  object0->SetId(0);
  NodeType::Pointer object1 = NodeType::New();
  object1->SetId(1);
  NodeType::Pointer object2 = NodeType::New();
  object2->SetId(2);
```

The hierarchy is formed using the `AddSpatialObject()` function.

```
  object0->AddSpatialObject(object1);
  object1->AddSpatialObject(object2);
```

After instantiation of the tree we set its root using the `SetRoot()` function.

```
TreeType::Pointer tree = TreeType::New();
tree->SetRoot(object0.GetPointer());
```

The tree iterators described in a previous section of this guide can be used to parse the hierarchy. For example, via an `itk::LevelOrderTreeIterator` templated over the type of tree, we can parse the hierarchy of SpatialObjects. We set the maximum level to 10 which is enough in this case since our hierarchy is only 2 deep.

```
itk::LevelOrderTreeIterator<TreeType> levelIt(tree,10);
levelIt.GoToBegin();
while(!levelIt.IsAtEnd())
  {
  std::cout << levelIt.Get()->GetId() << " ("<< levelIt.GetLevel()
    << ")" << std::endl;
  ++levelIt;
  }
```

Tree iterators can also be used to add spatial objects to the hierarchy. Here we show how to use the `itk::PreOrderTreeIterator` to add a fourth object to the tree.

```
NodeType::Pointer object4 = NodeType::New();
itk::PreOrderTreeIterator<TreeType> preIt( tree );
preIt.Add(object4.GetPointer());
```

5.4 Transformations

The source code for this section can be found in the file `SpatialObjectTransforms.cxx`.

This example describes the different transformations associated with a spatial object.

Figure 5.1 shows our set of transformations.

Like the first example, we create two spatial objects and give them the names `First Object` and `Second Object`, respectively.

```
typedef itk::SpatialObject<2>             SpatialObjectType;
typedef SpatialObjectType::TransformType  TransformType;

SpatialObjectType::Pointer object1 = SpatialObjectType ::New();
object1->GetProperty()->SetName("First Object");

SpatialObjectType::Pointer object2 = SpatialObjectType ::New();
object2->GetProperty()->SetName("Second Object");
object1->AddSpatialObject(object2);
```

Instances of `itk::SpatialObject` maintain three transformations internally that can be used to compute the position and orientation of data and objects. These transformations are: an IndexToObjectTransform, an ObjectToParentTransform, and an ObjectToWorldTransform. As a convenience

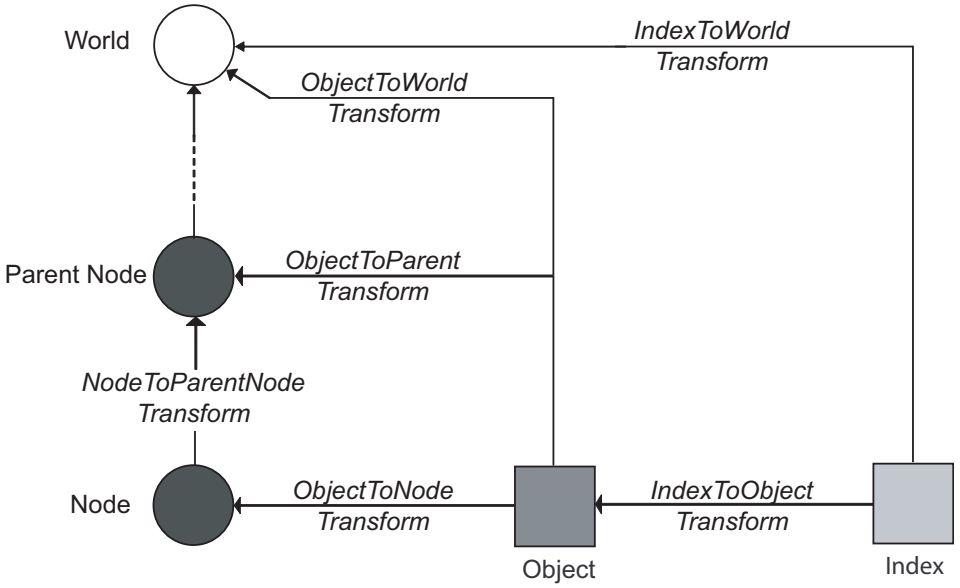

Figure 5.1: Set of transformations associated with a Spatial Object

to the user, the global transformation IndexToWorldTransform and its inverse, WorldToIndexTransform, are also maintained by the class. Methods are provided by SpatialObject to access and manipulate these transforms.

The two main transformations, IndexToObjectTransform and ObjectToParentTransform, are applied successively. ObjectToParentTransform is applied to children.

The IndexToObjectTransform transforms points from the internal data coordinate system of the object (typically the indices of the image from which the object was defined) to "physical" space (which accounts for the spacing, orientation, and offset of the indices).

The ObjectToParentTransform transforms points from the object-specific "physical" space to the "physical" space of its parent object. As one can see from the figure 5.1, the ObjectToParentTransform is composed of two transforms: ObjectToNodeTransform and NodeToParentNodeTransform. The ObjectToNodeTransform is not applied to the children, but the NodeToParentNodeTransform is. Therefore, if one sets the ObjectToParentTransform, the NodeToParentNodeTransform is actually set.

The ObjectToWorldTransform maps points from the reference system of the SpatialObject into the global coordinate system. This is useful when the position of the object is known only in the global coordinate frame. Note that by setting this transform, the ObjectToParent transform is recomputed.

These transformations use the `itk::FixedCenterOfRotationAffineTransform`. They are created in the constructor of the spatial `itk::SpatialObject`.

First we define an index scaling factor of 2 for the object2. This is done by setting the Scale of the IndexToObjectTransform.

```
double scale[2];
scale[0]=2;
scale[1]=2;
object2->GetIndexToObjectTransform()->SetScale(scale);
```

Next, we apply an offset on the ObjectToParentTransform of the child object Therefore, object2 is now translated by a vector [4,3] regarding to its parent.

```
TransformType::OffsetType Object2ToObject1Offset;
Object2ToObject1Offset[0] = 4;
Object2ToObject1Offset[1] = 3;
object2->GetObjectToParentTransform()->SetOffset(Object2ToObject1Offset);
```

To realize the previous operations on the transformations, we should invoke the `ComputeObjectToWorldTransform()` that recomputes all dependent transformations.

```
object2->ComputeObjectToWorldTransform();
```

We can now display the ObjectToWorldTransform for both objects. One should notice that the FixedCenterOfRotationAffineTransform derives from `itk::AffineTransform` and therefore the only valid members of the transformation are a Matrix and an Offset. For instance, when we invoke the `Scale()` method the internal Matrix is recomputed to reflect this change.

The FixedCenterOfRotationAffineTransform performs the following computation

$$X' = R \cdot (S \cdot X - C) + C + V \tag{5.1}$$

Where R is the rotation matrix, S is a scaling factor, C is the center of rotation and V is a translation vector or offset. Therefore the affine matrix M and the affine offset T are defined as:

$$M = R \cdot S \tag{5.2}$$

$$T = C + V - R \cdot C \tag{5.3}$$

This means that `GetScale()` and `GetOffset()` as well as the `GetMatrix()` might not be set to the expected value, especially if the transformation results from a composition with another transformation since the composition is done using the Matrix and the Offset of the affine transformation.

Next, we show the two affine transformations corresponding to the two objects.

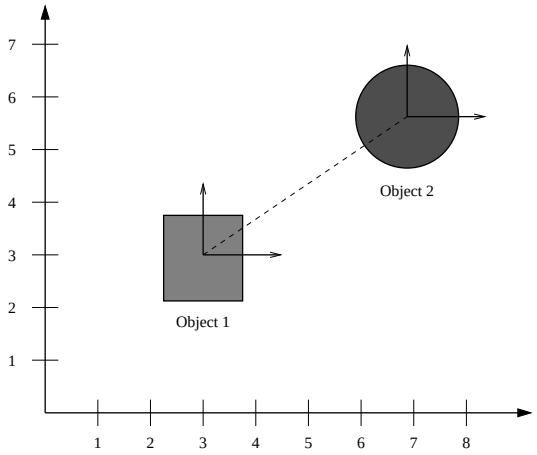

Figure 5.2: Physical positions of the two objects in the world frame (shapes are merely for illustration purposes).

```
std::cout << "object2 IndexToObject Matrix: " << std::endl;
std::cout << object2->GetIndexToObjectTransform()->GetMatrix() << std::endl;
std::cout << "object2 IndexToObject Offset: ";
std::cout << object2->GetIndexToObjectTransform()->GetOffset() << std::endl;
std::cout << "object2 IndexToWorld Matrix: " << std::endl;
std::cout << object2->GetIndexToWorldTransform()->GetMatrix() << std::endl;
std::cout << "object2 IndexToWorld Offset: ";
std::cout << object2->GetIndexToWorldTransform()->GetOffset() << std::endl;
```

Then, we decide to translate the first object which is the parent of the second by a vector [3,3]. This is still done by setting the offset of the ObjectToParentTransform. This can also be done by setting the ObjectToWorldTransform because the first object does not have any parent and therefore is attached to the world coordinate frame.

```
TransformType::OffsetType Object1ToWorldOffset;
Object1ToWorldOffset[0] = 3;
Object1ToWorldOffset[1] = 3;
object1->GetObjectToParentTransform()->SetOffset(Object1ToWorldOffset);
```

Next we invoke `ComputeObjectToWorldTransform()` on the modified object. This will propagate the transformation through all its children.

```
object1->ComputeObjectToWorldTransform();
```

Figure 5.2 shows our set of transformations.

Finally, we display the resulting affine transformations.

```
  std::cout << "object1 IndexToWorld Matrix: " << std::endl;
  std::cout << object1->GetIndexToWorldTransform()->GetMatrix() << std::endl;
  std::cout << "object1 IndexToWorld Offset: ";
  std::cout << object1->GetIndexToWorldTransform()->GetOffset() << std::endl;
  std::cout << "object2 IndexToWorld Matrix: " << std::endl;
  std::cout << object2->GetIndexToWorldTransform()->GetMatrix() << std::endl;
  std::cout << "object2 IndexToWorld Offset: ";
  std::cout << object2->GetIndexToWorldTransform()->GetOffset() << std::endl;
```

The output of this second example looks like the following:

```
object2 IndexToObject Matrix:
2 0
0 2
object2 IndexToObject Offset: 0  0
object2 IndexToWorld Matrix:
2 0
0 2
object2 IndexToWorld Offset: 4  3
object1 IndexToWorld Matrix:
1 0
0 1
object1 IndexToWorld Offset: 3  3
object2 IndexToWorld Matrix:
2 0
0 2
object2 IndexToWorld Offset: 7  6
```

5.5 Types of Spatial Objects

This section describes in detail the variety of spatial objects implemented in ITK.

5.5.1 ArrowSpatialObject

The source code for this section can be found in the file `ArrowSpatialObject.cxx`.

This example shows how to create a `itk::ArrowSpatialObject`. Let's begin by including the appropriate header file.

```
#include "itkArrowSpatialObject.h"
```

The `itk::ArrowSpatialObject`, like many SpatialObjects, is templated over the dimensionality of the object.

```
typedef itk::ArrowSpatialObject<3>   ArrowType;
ArrowType::Pointer myArrow = ArrowType::New();
```

The length of the arrow in the local coordinate frame is done using the SetLength() function. By default the length is set to 1.

```
myArrow->SetLength(2);
```

The direction of the arrow can be set using the SetDirection() function. The SetDirection() function modifies the ObjectToParentTransform (not the IndexToObjectTransform). By default the direction is set along the X axis (first direction).

```
ArrowType::VectorType direction;
direction.Fill(0);
direction[1] = 1.0;
myArrow->SetDirection(direction);
```

5.5.2 BlobSpatialObject

The source code for this section can be found in the file `BlobSpatialObject.cxx`.

`itk::BlobSpatialObject` defines an N-dimensional blob. Like other SpatialObjects this class derives from `itk::itkSpatialObject`. A blob is defined as a list of points which compose the object.

Let's start by including the appropriate header file.

```
#include "itkBlobSpatialObject.h"
```

BlobSpatialObject is templated over the dimension of the space. A BlobSpatialObject contains a list of SpatialObjectPoints. Basically, a SpatialObjectPoint has a position and a color.

```
#include "itkSpatialObjectPoint.h"
```

First we declare some type definitions.

```
typedef itk::BlobSpatialObject<3>    BlobType;
typedef BlobType::Pointer            BlobPointer;
typedef itk::SpatialObjectPoint<3>   BlobPointType;
```

Then, we create a list of points and we set the position of each point in the local coordinate system using the `SetPosition()` method. We also set the color of each point to be red.

```
BlobType::PointListType list;

 for( unsigned int i=0; i<4; i++)
   {
   BlobPointType p;
   p.SetPosition(i,i+1,i+2);
   p.SetRed(1);
   p.SetGreen(0);
   p.SetBlue(0);
   p.SetAlpha(1.0);
   list.push_back(p);
   }
```

Next, we create the blob and set its name using the `SetName()` function. We also set its Identification number with `SetId()` and we add the list of points previously created.

```
BlobPointer blob = BlobType::New();
blob->GetProperty()->SetName("My Blob");
blob->SetId(1);
blob->SetPoints(list);
```

The `GetPoints()` method returns a reference to the internal list of points of the object.

```
 BlobType::PointListType pointList = blob->GetPoints();
 std::cout << "The blob contains " << pointList.size();
 std::cout << " points" << std::endl;
```

Then we can access the points using standard STL iterators and `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point.

```
BlobType::PointListType::const_iterator it = blob->GetPoints().begin();
while(it != blob->GetPoints().end())
  {
  std::cout << "Position = " << (*it).GetPosition() << std::endl;
  std::cout << "Color = " << (*it).GetColor() << std::endl;
  it++;
  }
```

5.5.3 CylinderSpatialObject

The source code for this section can be found in the file `CylinderSpatialObject.cxx`.

This example shows how to create a `itk::CylinderSpatialObject`. Let's begin by including the appropriate header file.

```
#include "itkCylinderSpatialObject.h"
```

An `itk::CylinderSpatialObject` exists only in 3D, therefore, it is not templated.

```
typedef itk::CylinderSpatialObject   CylinderType;
```

We create a cylinder using the standard smart pointers.

```
CylinderType::Pointer myCylinder = CylinderType::New();
```

The radius of the cylinder is set using the `SetRadius()` function. By default the radius is set to 1.

```
double radius = 3.0;
myCylinder->SetRadius(radius);
```

The height of the cylinder is set using the `SetHeight()` function. By default the cylinder is defined along the X axis (first dimension).

```
double height = 12.0;
myCylinder->SetHeight(height);
```

Like any other `itk::SpatialObject`s, the `IsInside()` function can be used to query if a point is inside or outside the cylinder.

```
itk::Point<double,3> insidePoint;
insidePoint[0]=1;
insidePoint[1]=2;
insidePoint[2]=0;
std::cout << "Is my point "<< insidePoint << " inside the cylinder? : "
  << myCylinder->IsInside(insidePoint) << std::endl;
```

We can print the cylinder information using the `Print()` function.

```
myCylinder->Print(std::cout);
```

5.5.4 EllipseSpatialObject

The source code for this section can be found in the file `EllipseSpatialObject.cxx`.

`itk::EllipseSpatialObject` defines an n-Dimensional ellipse. Like other spatial objects this class derives from `itk::SpatialObject`. Let's start by including the appropriate header file.

```
#include "itkEllipseSpatialObject.h"
```

Like most of the SpatialObjects, the `itk::EllipseSpatialObject` is templated over the dimension of the space. In this example we create a 3-dimensional ellipse.

```
typedef itk::EllipseSpatialObject<3>   EllipseType;
EllipseType::Pointer myEllipse = EllipseType::New();
```

Then we set a radius for each dimension. By default the radius is set to 1.

```
EllipseType::ArrayType radius;
for(unsigned int i = 0; i<3; i++)
  {
  radius[i] = i;
  }

myEllipse->SetRadius(radius);
```

Or if we have the same radius in each dimension we can do

```
myEllipse->SetRadius(2.0);
```

We can then display the current radius by using the `GetRadius()` function:

```
EllipseType::ArrayType myCurrentRadius = myEllipse->GetRadius();
std::cout << "Current radius is " << myCurrentRadius << std::endl;
```

Like other SpatialObjects, we can query the object if a point is inside the object by using the IsInside(itk::Point) function. This function expects the point to be in world coordinates.

```
itk::Point<double,3> insidePoint;
insidePoint.Fill(1.0);
if(myEllipse->IsInside(insidePoint))
  {
  std::cout << "The point " << insidePoint;
  std::cout << " is really inside the ellipse" << std::endl;
  }

itk::Point<double,3> outsidePoint;
outsidePoint.Fill(3.0);
if(!myEllipse->IsInside(outsidePoint))
  {
  std::cout << "The point " << outsidePoint;
  std::cout << " is really outside the ellipse" << std::endl;
  }
```

All spatial objects can be queried for a value at a point. The `IsEvaluableAt()` function returns a boolean to know if the object is evaluable at a particular point.

```
 if(myEllipse->IsEvaluableAt(insidePoint))
  {
  std::cout << "The point " << insidePoint;
  std::cout << " is evaluable at the point " << insidePoint << std::endl;
  }
```

If the object is evaluable at that point, the `ValueAt()` function returns the current value at that position. Most of the objects returns a boolean value which is set to true when the point is inside the object and false when it is outside. However, for some objects, it is more interesting to return a

value representing, for instance, the distance from the center of the object or the distance from from the boundary.

```
double value;
myEllipse->ValueAt(insidePoint,value);
std::cout << "The value inside the ellipse is: " << value << std::endl;
```

Like other spatial objects, we can also query the bounding box of the object by using `GetBoundingBox()`. The resulting bounding box is expressed in the local frame.

```
myEllipse->ComputeBoundingBox();
EllipseType::BoundingBoxType * boundingBox = myEllipse->GetBoundingBox();
std::cout << "Bounding Box: " << boundingBox->GetBounds() << std::endl;
```

5.5.5 GaussianSpatialObject

The source code for this section can be found in the file `GaussianSpatialObject.cxx`.

This example shows how to create a `itk::GaussianSpatialObject` which defines a Gaussian in a N-dimensional space. This object is particularly useful to query the value at a point in physical space. Let's begin by including the appropriate header file.

```
#include "itkGaussianSpatialObject.h"
```

The `itk::GaussianSpatialObject` is templated over the dimensionality of the object.

```
typedef itk::GaussianSpatialObject<3>   GaussianType;
GaussianType::Pointer myGaussian = GaussianType::New();
```

The `SetMaximum()` function is used to set the maximum value of the Gaussian.

```
myGaussian->SetMaximum(2);
```

The radius of the Gaussian is defined by the `SetRadius()` method. By default the radius is set to 1.0.

```
myGaussian->SetRadius(3);
```

The standard `ValueAt()` function is used to determine the value of the Gaussian at a particular point in physical space.

```
itk::Point<double,3> pt;
pt[0]=1;
pt[1]=2;
pt[2]=1;
double value;
myGaussian->ValueAt(pt, value);
std::cout << "ValueAt(" << pt << ") = " << value << std::endl;
```

5.5.6 GroupSpatialObject

The source code for this section can be found in the file `GroupSpatialObject.cxx`.

A `itk::GroupSpatialObject` does not have any data associated with it. It can be used to group objects or to add transforms to a current object. In this example we show how to use a GroupSpatialObject.

Let's begin by including the appropriate header file.

```
#include "itkGroupSpatialObject.h"
```

The `itk::GroupSpatialObject` is templated over the dimensionality of the object.

```
typedef itk::GroupSpatialObject<3>   GroupType;
GroupType::Pointer myGroup = GroupType::New();
```

Next, we create an `itk::EllipseSpatialObject` and add it to the group.

```
typedef itk::EllipseSpatialObject<3>   EllipseType;
EllipseType::Pointer myEllipse = EllipseType::New();
myEllipse->SetRadius(2);

myGroup->AddSpatialObject(myEllipse);
```

We then translate the group by 10mm in each direction. Therefore the ellipse is translated in physical space at the same time.

```
GroupType::VectorType offset;
offset.Fill(10);
myGroup->GetObjectToParentTransform()->SetOffset(offset);
myGroup->ComputeObjectToWorldTransform();
```

We can then query if a point is inside the group using the `IsInside()` function. We need to specify in this case that we want to consider all the hierarchy, therefore we set the depth to 2.

```
GroupType::PointType point;
point.Fill(10);
std::cout << "Is my point " << point << " inside?: "
  <<  myGroup->IsInside(point,2) << std::endl;
```

Like any other SpatialObjects we can remove the ellipse from the group using the `RemoveSpatialObject()` method.

```
myGroup->RemoveSpatialObject(myEllipse);
```

5.5.7 ImageSpatialObject

The source code for this section can be found in the file `ImageSpatialObject.cxx`.

An `itk::ImageSpatialObject` contains an `itk::Image` but adds the notion of spatial transformations and parent-child hierarchy. Let's begin the next example by including the appropriate header file.

```
#include "itkImageSpatialObject.h"
```

We first create a simple 2D image of size 10 by 10 pixels.

```
typedef itk::Image<short,2> Image;
Image::Pointer image = Image::New();
Image::SizeType size = {{ 10, 10 }};
Image::RegionType region;
region.SetSize(size);
image->SetRegions(region);
image->Allocate();
```

Next we fill the image with increasing values.

```
typedef itk::ImageRegionIterator<Image> Iterator;
Iterator it(image,region);
short pixelValue =0;
it.GoToBegin();
for(; !it.IsAtEnd(); ++it, ++pixelValue)
  {
  it.Set(pixelValue);
  }
```

We can now define the ImageSpatialObject which is templated over the dimension and the pixel type of the image.

```
typedef itk::ImageSpatialObject<2,short> ImageSpatialObject;
ImageSpatialObject::Pointer imageSO = ImageSpatialObject::New();
```

Then we set the itkImage to the ImageSpatialObject by using the `SetImage()` function.

```
imageSO->SetImage(image);
```

At this point we can use `IsInside()`, `ValueAt()` and `DerivativeAt()` functions inherent in SpatialObjects. The `IsInside()` value can be useful when dealing with registration.

```
typedef itk::Point<double,2> Point;
Point insidePoint;
insidePoint.Fill(9);

if( imageSO->IsInside(insidePoint) )
  {
  std::cout << insidePoint << " is inside the image." << std::endl;
  }
```

The `ValueAt()` returns the value of the closest pixel, i.e no interpolation, to a given physical point.

```
double returnedValue;
imageSO->ValueAt(insidePoint,returnedValue);
std::cout << "ValueAt(" << insidePoint << ") = " << returnedValue
          << std::endl;
```

The derivative at a specified position in space can be computed using the `DerivativeAt()` function. The first argument is the point in physical coordinates where we are evaluating the derivatives. The second argument is the order of the derivation, and the third argument is the result expressed as a `itk::Vector`. Derivatives are computed iteratively using finite differences and, like the `ValueAt()`, no interpolator is used.

```
ImageSpatialObject::OutputVectorType returnedDerivative;
imageSO->DerivativeAt(insidePoint,1,returnedDerivative);
std::cout << "First derivative at " << insidePoint;
std::cout << " = " << returnedDerivative << std::endl;
```

5.5.8 ImageMaskSpatialObject

The source code for this section can be found in the file `ImageMaskSpatialObject.cxx`.

An `itk::ImageMaskSpatialObject` is similar to the `itk::ImageSpatialObject` and derived from it. However, the main difference is that the `IsInside()` returns true if the pixel intensity in the image is not zero.

The supported pixel types does not include `itk::RGBPixel`, `itk::RGBAPixel`, etc... So far it only allows to manage images of simple types like unsigned short, unsigned int, or `itk::Vector`. Let's begin by including the appropriate header file.

```
#include "itkImageMaskSpatialObject.h"
```

The ImageMaskSpatialObject is templated over the dimensionality.

```
typedef itk::ImageMaskSpatialObject<3> ImageMaskSpatialObject;
```

Next we create an itk::Image of size 50x50x50 filled with zeros except a bright square in the middle which defines the mask.

```
typedef ImageMaskSpatialObject::PixelType     PixelType;
typedef ImageMaskSpatialObject::ImageType     ImageType;
typedef itk::ImageRegionIterator< ImageType > Iterator;

ImageType::Pointer image = ImageType::New();
ImageType::SizeType size = {{ 50, 50, 50 }};
ImageType::IndexType index = {{ 0, 0, 0 }};
ImageType::RegionType region;

region.SetSize(size);
region.SetIndex(index);

image->SetRegions( region );
image->Allocate(true); // initialize buffer to zero

ImageType::RegionType insideRegion;
ImageType::SizeType insideSize   = {{ 30, 30, 30 }};
ImageType::IndexType insideIndex = {{ 10, 10, 10 }};
insideRegion.SetSize( insideSize );
insideRegion.SetIndex( insideIndex );

Iterator it( image, insideRegion );
it.GoToBegin();

while( !it.IsAtEnd() )
  {
  it.Set( itk::NumericTraits< PixelType >::max() );
  ++it;
  }
```

Then, we create an ImageMaskSpatialObject.

```
ImageMaskSpatialObject::Pointer maskSO = ImageMaskSpatialObject::New();
```

We then pass the corresponding pointer to the image.

```
maskSO->SetImage(image);
```

We can then test if a physical itk::Point is inside or outside the mask image. This is particularly useful during the registration process when only a part of the image should be used to compute the metric.

```
ImageMaskSpatialObject::PointType  inside;
inside.Fill(20);
std::cout << "Is my point " << inside << " inside my mask? "
  << maskSO->IsInside(inside) << std::endl;
ImageMaskSpatialObject::PointType  outside;
outside.Fill(45);
std::cout << "Is my point " << outside << " outside my mask? "
  << !maskSO->IsInside(outside) << std::endl;
```

5.5.9 LandmarkSpatialObject

The source code for this section can be found in the file `LandmarkSpatialObject.cxx`.

`itk::LandmarkSpatialObject` contains a list of `itk::SpatialObjectPoint`s which have a position and a color. Let's begin this example by including the appropriate header file.

```
#include "itkLandmarkSpatialObject.h"
```

LandmarkSpatialObject is templated over the dimension of the space.

Here we create a 3-dimensional landmark.

```
typedef itk::LandmarkSpatialObject<3>  LandmarkType;
typedef LandmarkType::Pointer          LandmarkPointer;
typedef itk::SpatialObjectPoint<3>     LandmarkPointType;

LandmarkPointer landmark = LandmarkType::New();
```

Next, we set some properties of the object like its name and its identification number.

```
landmark->GetProperty()->SetName("Landmark1");
landmark->SetId(1);
```

We are now ready to add points into the landmark. We first create a list of SpatialObjectPoint and for each point we set the position and the color.

```
LandmarkType::PointListType list;

for( unsigned int i=0; i<5; i++)
  {
  LandmarkPointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetColor(1,0,0,1);
  list.push_back(p);
  }
```

Then we add the list to the object using the `SetPoints()` method.

```
landmark->SetPoints(list);
```

The current point list can be accessed using the `GetPoints()` method. The method returns a reference to the (STL) list.

```
unsigned int nPoints = landmark->GetPoints().size();
std::cout << "Number of Points in the landmark: " << nPoints << std::endl;

LandmarkType::PointListType::const_iterator it
                                        = landmark->GetPoints().begin();
while(it != landmark->GetPoints().end())
  {
  std::cout << "Position: " << (*it).GetPosition() << std::endl;
  std::cout << "Color: " << (*it).GetColor() << std::endl;
  it++;
  }
```

5.5.10 LineSpatialObject

The source code for this section can be found in the file `LineSpatialObject.cxx`.

`itk::LineSpatialObject` defines a line in an n-dimensional space. A line is defined as a list of points which compose the line, i.e a polyline. We begin the example by including the appropriate header files.

```
#include "itkLineSpatialObject.h"
```

LineSpatialObject is templated over the dimension of the space. A LineSpatialObject contains a list of LineSpatialObjectPoints. A LineSpatialObjectPoint has a position, $n-1$ normals and a color. Each normal is expressed as a `itk::CovariantVector` of size N.

First, we define some type definitions and we create our line.

```
typedef itk::LineSpatialObject<3>        LineType;
typedef LineType::Pointer                LinePointer;
typedef itk::LineSpatialObjectPoint<3>   LinePointType;
typedef itk::CovariantVector<double,3>    VectorType;

LinePointer Line = LineType::New();
```

We create a point list and we set the position of each point in the local coordinate system using the `SetPosition()` method. We also set the color of each point to red.

The two normals are set using the `SetNormal()` function; the first argument is the normal itself and the second argument is the index of the normal.

```
LineType::PointListType list;

for(unsigned int i=0; i<3; i++)
  {
  LinePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetColor(1,0,0,1);

  VectorType normal1;
  VectorType normal2;
  for(unsigned int j=0;j<3;j++)
    {
    normal1[j]=j;
    normal2[j]=j*2;
    }

  p.SetNormal(normal1,0);
  p.SetNormal(normal2,1);
  list.push_back(p);
  }
```

Next, we set the name of the object using `SetName()`. We also set its identification number with `SetId()` and we set the list of points previously created.

```
Line->GetProperty()->SetName("Line1");
Line->SetId(1);
Line->SetPoints(list);
```

The `GetPoints()` method returns a reference to the internal list of points of the object.

```
LineType::PointListType pointList = Line->GetPoints();
std::cout << "Number of points representing the line: ";
std::cout << pointList.size() << std::endl;
```

Then we can access the points using standard STL iterators. The `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point. Using the GetNormal(unsigned int) function we can access each normal.

```
LineType::PointListType::const_iterator it = Line->GetPoints().begin();
while(it != Line->GetPoints().end())
  {
  std::cout << "Position = " << (*it).GetPosition() << std::endl;
  std::cout << "Color = " << (*it).GetColor() << std::endl;
  std::cout << "First normal = " << (*it).GetNormal(0) << std::endl;
  std::cout << "Second normal = " << (*it).GetNormal(1) << std::endl;
  std::cout << std::endl;
  it++;
  }
```

5.5.11 MeshSpatialObject

The source code for this section can be found in the file `MeshSpatialObject.cxx`.

A `itk::MeshSpatialObject` contains a pointer to an `itk::Mesh` but adds the notion of spatial transformations and parent-child hierarchy. This example shows how to create an `itk::MeshSpatialObject`, use it to form a binary image, and write the mesh to disk.

Let's begin by including the appropriate header file.

```
#include "itkSpatialObjectToImageFilter.h"
#include "itkMeshSpatialObject.h"
#include "itkSpatialObjectReader.h"
#include "itkSpatialObjectWriter.h"
```

The MeshSpatialObject wraps an `itk::Mesh`, therefore we first create a mesh.

```
  typedef itk::DefaultDynamicMeshTraits< float, 3, 3 > MeshTrait;
  typedef itk::Mesh< float, 3, MeshTrait >             MeshType;
  typedef MeshType::CellTraits                         CellTraits;
  typedef itk::CellInterface< float, CellTraits >      CellInterfaceType;
  typedef itk::TetrahedronCell< CellInterfaceType >    TetraCellType;
  typedef MeshType::PointType                          PointType;
  typedef MeshType::CellType                           CellType;
  typedef CellType::CellAutoPointer                    CellAutoPointer;
```

```
  MeshType::Pointer myMesh = MeshType::New();

  MeshType::CoordRepType testPointCoords[4][3]
    = { {0,0,0}, {9,0,0}, {9,9,0}, {0,0,9} };

  MeshType::PointIdentifier tetraPoints[4] = {0,1,2,4};

  int i;
  for(i=0; i < 4; ++i)
    {
    myMesh->SetPoint(i, PointType(testPointCoords[i]));
    }

  myMesh->SetCellsAllocationMethod(
      MeshType::CellsAllocatedDynamicallyCellByCell );
  CellAutoPointer testCell1;
  testCell1.TakeOwnership(  new TetraCellType );
  testCell1->SetPointIds(tetraPoints);
```

```
  myMesh->SetCell(0, testCell1 );
```

We then create a MeshSpatialObject which is templated over the type of mesh previously defined...

```
  typedef itk::MeshSpatialObject< MeshType > MeshSpatialObjectType;
  MeshSpatialObjectType::Pointer myMeshSpatialObject =
                                      MeshSpatialObjectType::New();
```

... and pass the Mesh pointer to the MeshSpatialObject

```
myMeshSpatialObject->SetMesh(myMesh);
```

The actual pointer to the passed mesh can be retrieved using the `GetMesh()` function, just like any other SpatialObjects.

```
myMeshSpatialObject->GetMesh();
```

The `GetBoundingBox()`, `ValueAt()`, `IsInside()` functions can be used to access important information.

```
std::cout << "Mesh bounds : " <<
  myMeshSpatialObject->GetBoundingBox()->GetBounds() << std::endl;
MeshSpatialObjectType::PointType myPhysicalPoint;
myPhysicalPoint.Fill(1);
std::cout << "Is my physical point inside? : " <<
  myMeshSpatialObject->IsInside(myPhysicalPoint) << std::endl;
```

Now that we have defined the MeshSpatialObject, we can save the actual mesh using the `itk::SpatialObjectWriter`. In order to do so, we need to specify the type of Mesh we are writing.

```
typedef itk::SpatialObjectWriter< 3, float, MeshTrait > WriterType;
WriterType::Pointer writer = WriterType::New();
```

Then we set the mesh spatial object and the name of the file and call the the `Update()` function.

```
writer->SetInput(myMeshSpatialObject);
writer->SetFileName("myMesh.meta");
writer->Update();
```

Reading the saved mesh is done using the `itk::SpatialObjectReader`. Once again we need to specify the type of mesh we intend to read.

```
typedef itk::SpatialObjectReader< 3, float, MeshTrait > ReaderType;
ReaderType::Pointer reader = ReaderType::New();
```

We set the name of the file we want to read and call update

```
reader->SetFileName("myMesh.meta");
reader->Update();
```

Next, we show how to create a binary image of a MeshSpatialObject using the `itk::SpatialObjectToImageFilter`. The resulting image will have ones inside and zeros outside the mesh. First we define and instantiate the SpatialObjectToImageFilter.

```
typedef itk::Image< unsigned char, 3 > ImageType;
typedef itk::GroupSpatialObject< 3 >   GroupType;
typedef itk::SpatialObjectToImageFilter< GroupType, ImageType >
                                    SpatialObjectToImageFilterType;
SpatialObjectToImageFilterType::Pointer imageFilter =
  SpatialObjectToImageFilterType::New();
```

Then we pass the output of the reader, i.e the MeshSpatialObject, to the filter.

```
imageFilter->SetInput(  reader->GetGroup()  );
```

Finally we trigger the execution of the filter by calling the `Update()` method. Note that depending on the size of the mesh, the computation time can increase significantly.

```
imageFilter->Update();
```

Then we can get the resulting binary image using the `GetOutput()` function.

```
ImageType::Pointer myBinaryMeshImage = imageFilter->GetOutput();
```

5.5.12 SurfaceSpatialObject

The source code for this section can be found in the file `SurfaceSpatialObject.cxx`.

`itk::SurfaceSpatialObject` defines a surface in n-dimensional space. A SurfaceSpatialObject is defined by a list of points which lie on the surface. Each point has a position and a unique normal. The example begins by including the appropriate header file.

```
#include "itkSurfaceSpatialObject.h"
```

SurfaceSpatialObject is templated over the dimension of the space. A SurfaceSpatialObject contains a list of SurfaceSpatialObjectPoints. A SurfaceSpatialObjectPoint has a position, a normal and a color.

First we define some type definitions

```
typedef itk::SurfaceSpatialObject<3>        SurfaceType;
typedef SurfaceType::Pointer                SurfacePointer;
typedef itk::SurfaceSpatialObjectPoint<3>   SurfacePointType;
typedef itk::CovariantVector<double,3>      VectorType;

SurfacePointer Surface = SurfaceType::New();
```

We create a point list and we set the position of each point in the local coordinate system using the `SetPosition()` method. We also set the color of each point to red.

```
SurfaceType::PointListType list;

for( unsigned int i=0; i<3; i++)
  {
  SurfacePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetColor(1,0,0,1);
  VectorType normal;
  for(unsigned int j=0;j<3;j++)
    {
    normal[j]=j;
    }
  p.SetNormal(normal);
  list.push_back(p);
  }
```

Next, we create the surface and set his name using `SetName()`. We also set its Identification number with `SetId()` and we add the list of points previously created.

```
Surface->GetProperty()->SetName("Surface1");
Surface->SetId(1);
Surface->SetPoints(list);
```

The `GetPoints()` method returns a reference to the internal list of points of the object.

```
SurfaceType::PointListType pointList = Surface->GetPoints();
std::cout << "Number of points representing the surface: ";
std::cout << pointList.size() << std::endl;
```

Then we can access the points using standard STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point. `GetNormal()` returns the normal as a `itk::CovariantVector`.

```
SurfaceType::PointListType::const_iterator it
                                        = Surface->GetPoints().begin();
while(it != Surface->GetPoints().end())
  {
  std::cout << "Position = " << (*it).GetPosition() << std::endl;
  std::cout << "Normal = " << (*it).GetNormal() << std::endl;
  std::cout << "Color = " << (*it).GetColor() << std::endl;
  std::cout << std::endl;
  it++;
  }
```

5.5.13 TubeSpatialObject

`itk::TubeSpatialObject` represents a base class for the representation of tubular structures using SpatialObjects. The classes `itk::VesselTubeSpatialObject` and `itk::DTITubeSpatialObject` derive from this base class. VesselTubeSpatialObject represents blood vessels extracted for an image and DTITubeSpatialObject is used to represent fiber

tracts from diffusion tensor images.

The source code for this section can be found in the file `TubeSpatialObject.cxx`.

`itk::TubeSpatialObject` defines an n-dimensional tube. A tube is defined as a list of centerline points which have a position, a radius, some normals and other properties. Let's start by including the appropriate header file.

```
#include "itkTubeSpatialObject.h"
```

TubeSpatialObject is templated over the dimension of the space. A TubeSpatialObject contains a list of TubeSpatialObjectPoints.

First we define some type definitions and we create the tube.

```
typedef itk::TubeSpatialObject<3>           TubeType;
typedef TubeType::Pointer                   TubePointer;
typedef itk::TubeSpatialObjectPoint<3>      TubePointType;
typedef TubePointType::CovariantVectorType  VectorType;

TubePointer tube = TubeType::New();
```

We create a point list and we set:

1. The position of each point in the local coordinate system using the `SetPosition()` method.
2. The radius of the tube at this position using `SetRadius()`.
3. The two normals at the tube is set using `SetNormal1()` and `SetNormal2()`.
4. The color of the point is set to red in our case.

```
TubeType::PointListType list;
for( i=0; i<5; i++)
  {
  TubePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetRadius(1);
  VectorType normal1;
  VectorType normal2;
  for(unsigned int j=0;j<3;j++)
    {
    normal1[j]=j;
    normal2[j]=j*2;
    }

  p.SetNormal1(normal1);
  p.SetNormal2(normal2);
  p.SetColor(1,0,0,1);

  list.push_back(p);
  }
```

Next, we create the tube and set its name using `SetName()`. We also set its identification number with `SetId()` and, at the end, we add the list of points previously created.

```
tube->GetProperty()->SetName("Tube1");
tube->SetId(1);
tube->SetPoints(list);
```

The `GetPoints()` method return a reference to the internal list of points of the object.

```
TubeType::PointListType pointList = tube->GetPoints();
std::cout << "Number of points representing the tube: ";
std::cout << pointList.size() << std::endl;
```

The `ComputeTangentAndNormals()` function computes the normals and the tangent for each point using finite differences.

```
tube->ComputeTangentAndNormals();
```

Then we can access the points using STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point. `GetRadius()` returns the radius at that point. `GetNormal1()` and `GetNormal1()` functions return a `itk::CovariantVector` and `GetTangent()` returns a `itk::Vector`.

```
TubeType::PointListType::const_iterator it = tube->GetPoints().begin();
i=0;
while(it != tube->GetPoints().end())
  {
  std::cout << std::endl;
  std::cout << "Point #" << i << std::endl;
  std::cout << "Position: " << (*it).GetPosition() << std::endl;
  std::cout << "Radius: " << (*it).GetRadius() << std::endl;
  std::cout << "Tangent: " << (*it).GetTangent() << std::endl;
  std::cout << "First Normal: " << (*it).GetNormal1() << std::endl;
  std::cout << "Second Normal: " << (*it).GetNormal2() << std::endl;
  std::cout << "Color = " << (*it).GetColor() << std::endl;
  it++;
  i++;
  }
```

VesselTubeSpatialObject

The source code for this section can be found in the file `VesselTubeSpatialObject.cxx`.

`itk::VesselTubeSpatialObject` derives from `itk::TubeSpatialObject`. It represents a blood vessel segmented from an image. A VesselTubeSpatialObject is described as a list of centerline points which have a position, a radius, and normals.

Let's start by including the appropriate header file.

```
#include "itkVesselTubeSpatialObject.h"
```

VesselTubeSpatialObject is templated over the dimension of the space. A VesselTubeSpatialObject contains a list of VesselTubeSpatialObjectPoints.

First we define some type definitions and we create the tube.

```
typedef itk::VesselTubeSpatialObject<3>            VesselTubeType;
typedef itk::VesselTubeSpatialObjectPoint<3>       VesselTubePointType;

VesselTubeType::Pointer VesselTube = VesselTubeType::New();
```

We create a point list and we set:

1. The position of each point in the local coordinate system using the `SetPosition()` method.
2. The radius of the tube at this position using `SetRadius()`.
3. The medialness value describing how the point lies in the middle of the vessel using `SetMedialness()`.
4. The ridgeness value describing how the point lies on the ridge using `SetRidgeness()`.
5. The branchness value describing if the point is a branch point using `SetBranchness()`.
6. The three alpha values corresponding to the eigenvalues of the Hessian using `SetAlpha1()`,`SetAlpha2()` and `SetAlpha3()`.
7. The mark value using `SetMark()`.
8. The color of the point is set to red in this example with an opacity of 1.

```
VesselTubeType::PointListType list;
for( i=0; i<5; i++)
  {
  VesselTubePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetRadius(1);
  p.SetAlpha1(i);
  p.SetAlpha2(i+1);
  p.SetAlpha3(i+2);
  p.SetMedialness(i);
  p.SetRidgeness(i);
  p.SetBranchness(i);
  p.SetMark(true);
  p.SetColor(1,0,0,1);
  list.push_back(p);
  }
```

Next, we create the tube and set its name using `SetName()`. We also set its identification number with `SetId()` and, at the end, we add the list of points previously created.

```
VesselTube->GetProperty()->SetName("VesselTube");
VesselTube->SetId(1);
VesselTube->SetPoints(list);
```

The `GetPoints()` method return a reference to the internal list of points of the object.

```
VesselTubeType::PointListType pointList = VesselTube->GetPoints();
std::cout << "Number of points representing the blood vessel: ";
std::cout << pointList.size() << std::endl;
```

Then we can access the points using STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point.

```
VesselTubeType::PointListType::const_iterator
          it = VesselTube->GetPoints().begin();
i=0;
while(it != VesselTube->GetPoints().end())
  {
  std::cout << std::endl;
  std::cout << "Point #" << i << std::endl;
  std::cout << "Position: " << (*it).GetPosition() << std::endl;
  std::cout << "Radius: " << (*it).GetRadius() << std::endl;
  std::cout << "Medialness: " << (*it).GetMedialness() << std::endl;
  std::cout << "Ridgeness: " << (*it).GetRidgeness() << std::endl;
  std::cout << "Branchness: " << (*it).GetBranchness() << std::endl;
  std::cout << "Mark: " << (*it).GetMark() << std::endl;
  std::cout << "Alpha1: " << (*it).GetAlpha1() << std::endl;
  std::cout << "Alpha2: " << (*it).GetAlpha2() << std::endl;
  std::cout << "Alpha3: " << (*it).GetAlpha3() << std::endl;
  std::cout << "Color = " << (*it).GetColor() << std::endl;
  it++;
  i++;
  }
```

DTITubeSpatialObject

The source code for this section can be found in the file `DTITubeSpatialObject.cxx`.

`itk::DTITubeSpatialObject` derives from `itk::TubeSpatialObject`. It represents a fiber tracts from Diffusion Tensor Imaging. A DTITubeSpatialObject is described as a list of centerline points which have a position, a radius, normals, the fractional anisotropy (FA) value, the ADC value, the geodesic anisotropy (GA) value, the eigenvalues and vectors as well as the full tensor matrix.

Let's start by including the appropriate header file.

```
#include "itkDTITubeSpatialObject.h"
```

DTITubeSpatialObject is templated over the dimension of the space. A DTITubeSpatialObject contains a list of DTITubeSpatialObjectPoints.

First we define some type definitions and we create the tube.

```
typedef itk::DTITubeSpatialObject<3>            DTITubeType;
typedef itk::DTITubeSpatialObjectPoint<3>       DTITubePointType;

DTITubeType::Pointer dtiTube = DTITubeType::New();
```

We create a point list and we set:

1. The position of each point in the local coordinate system using the `SetPosition()` method.
2. The radius of the tube at this position using `SetRadius()`.
3. The FA value using `AddField(DTITubePointType::FA)`.
4. The ADC value using `AddField(DTITubePointType::ADC)`.
5. The GA value using `AddField(DTITubePointType::GA)`.
6. The full tensor matrix supposed to be symmetric definite positive value using `SetTensorMatrix()`.
7. The color of the point is set to red in our case.

```
DTITubeType::PointListType list;
for( i=0; i<5; i++)
  {
  DTITubePointType p;
  p.SetPosition(i,i+1,i+2);
  p.SetRadius(1);
  p.AddField(DTITubePointType::FA,i);
  p.AddField(DTITubePointType::ADC,2*i);
  p.AddField(DTITubePointType::GA,3*i);
  p.AddField("Lambda1",4*i);
  p.AddField("Lambda2",5*i);
  p.AddField("Lambda3",6*i);
  float* v = new float[6];
  for(unsigned int k=0;k<6;k++)
    {
    v[k] = k;
    }
  p.SetTensorMatrix(v);
  delete[] v;
  p.SetColor(1,0,0,1);
  list.push_back(p);
  }
```

Next, we create the tube and set its name using `SetName()`. We also set its identification number with `SetId()` and, at the end, we add the list of points previously created.

```
dtiTube->GetProperty()->SetName("DTITube");
dtiTube->SetId(1);
dtiTube->SetPoints(list);
```

The `GetPoints()` method return a reference to the internal list of points of the object.

```
DTITubeType::PointListType pointList = dtiTube->GetPoints();
std::cout << "Number of points representing the fiber tract: ";
std::cout << pointList.size() << std::endl;
```

Then we can access the points using STL iterators. `GetPosition()` and `GetColor()` functions return respectively the position and the color of the point.

```
DTITubeType::PointListType::const_iterator it = dtiTube->GetPoints().begin();
i=0;
while(it != dtiTube->GetPoints().end())
  {
  std::cout << std::endl;
  std::cout << "Point #" << i << std::endl;
  std::cout << "Position: " << (*it).GetPosition() << std::endl;
  std::cout << "Radius: " << (*it).GetRadius() << std::endl;
  std::cout << "FA: " << (*it).GetField(DTITubePointType::FA) << std::endl;
  std::cout << "ADC: " << (*it).GetField(DTITubePointType::ADC) << std::endl;
  std::cout << "GA: " << (*it).GetField(DTITubePointType::GA) << std::endl;
  std::cout << "Lambda1: " << (*it).GetField("Lambda1") << std::endl;
  std::cout << "Lambda2: " << (*it).GetField("Lambda2") << std::endl;
  std::cout << "Lambda3: " << (*it).GetField("Lambda3") << std::endl;
  std::cout << "TensorMatrix: " << (*it).GetTensorMatrix()[0] << " : ";
  std::cout << (*it).GetTensorMatrix()[1] << " : ";
  std::cout << (*it).GetTensorMatrix()[2] << " : ";
  std::cout << (*it).GetTensorMatrix()[3] << " : ";
  std::cout << (*it).GetTensorMatrix()[4] << " : ";
  std::cout << (*it).GetTensorMatrix()[5] << std::endl;
  std::cout << "Color = " << (*it).GetColor() << std::endl;
  it++;
  i++;
  }
```

5.6 SceneSpatialObject

The source code for this section can be found in the file `SceneSpatialObject.cxx`.

This example describes how to use the `itk::SceneSpatialObject`. A SceneSpatialObject contains a collection of SpatialObjects. This example begins by including the appropriate header file.

```
#include "itkSceneSpatialObject.h"
```

An SceneSpatialObject is templated over the dimension of the space which requires all the objects referenced by the SceneSpatialObject to have the same dimension.

First we define some type definitions and we create the SceneSpatialObject.

```
typedef itk::SceneSpatialObject<3> SceneSpatialObjectType;
SceneSpatialObjectType::Pointer scene = SceneSpatialObjectType::New();
```

Then we create two `itk::EllipseSpatialObject`s.

```
typedef itk::EllipseSpatialObject<3> EllipseType;
EllipseType::Pointer ellipse1 = EllipseType::New();
ellipse1->SetRadius(1);
ellipse1->SetId(1);
EllipseType::Pointer ellipse2 = EllipseType::New();
ellipse2->SetId(2);
ellipse2->SetRadius(2);
```

Then we add the two ellipses into the SceneSpatialObject.

```
scene->AddSpatialObject(ellipse1);
scene->AddSpatialObject(ellipse2);
```

We can query the number of object in the SceneSpatialObject with the `GetNumberOfObjects()` function. This function takes two optional arguments: the depth at which we should count the number of objects (default is set to infinity) and the name of the object to count (default is set to NULL). This allows the user to count, for example, only ellipses.

```
std::cout << "Number of objects in the SceneSpatialObject = ";
std::cout << scene->GetNumberOfObjects() << std::endl;
```

The `GetObjectById()` returns the first object in the SceneSpatialObject that has the specified identification number.

```
std::cout << "Object in the SceneSpatialObject with an ID == 2: "
          << std::endl;
scene->GetObjectById(2)->Print(std::cout);
```

Objects can also be removed from the SceneSpatialObject using the `RemoveSpatialObject()` function.

```
scene->RemoveSpatialObject(ellipse1);
```

The list of current objects in the SceneSpatialObject can be retrieved using the `GetObjects()` method. Like the `GetNumberOfObjects()` method, `GetObjects()` can take two arguments: a search depth and a matching name.

```
SceneSpatialObjectType::ObjectListType * myObjectList =  scene->GetObjects();
std::cout << "Number of objects in the SceneSpatialObject = ";
std::cout << myObjectList->size() << std::endl;
```

In some cases, it is useful to define the hierarchy by using `ParentId()` and the current identification number. This results in having a flat list of SpatialObjects in the SceneSpatialObject. Therefore, the SceneSpatialObject provides the `FixHierarchy()` method which reorganizes the Parent-Child hierarchy based on identification numbers.

```
scene->FixHierarchy();
```

The scene can also be cleared by using the `Clear()` function.

```
scene->Clear();
```

5.7 Read/Write SpatialObjects

The source code for this section can be found in the file `ReadWriteSpatialObject.cxx`.

Reading and writing SpatialObjects is a fairly simple task. The classes `itk::SpatialObjectReader` and `itk::SpatialObjectWriter` are used to read and write these objects, respectively. (Note these classes make use of the MetaIO auxiliary I/O routines and therefore have a `.meta` file suffix.)

We begin this example by including the appropriate header files.

```
#include "itkSpatialObjectReader.h"
#include "itkSpatialObjectWriter.h"
#include "itkEllipseSpatialObject.h"
```

Next, we create a SpatialObjectWriter that is templated over the dimension of the object(s) we want to write.

```
typedef itk::SpatialObjectWriter<3> WriterType;
WriterType::Pointer writer = WriterType::New();
```

For this example, we create an `itk::EllipseSpatialObject`.

```
typedef itk::EllipseSpatialObject<3> EllipseType;
EllipseType::Pointer ellipse = EllipseType::New();
ellipse->SetRadius(3);
```

Finally, we set to the writer the object to write using the `SetInput()` method and we set the name of the file with `SetFileName()` and call the `Update()` method to actually write the information.

```
writer->SetInput(ellipse);
writer->SetFileName("ellipse.meta");
writer->Update();
```

Now we are ready to open the freshly created object. We first create a SpatialObjectReader which is also templated over the dimension of the object in the file. This means that the file should contain only objects with the same dimension.

```
typedef itk::SpatialObjectReader<3> ReaderType;
ReaderType::Pointer reader = ReaderType::New();
```

Next we set the name of the file to read using `SetFileName()` and we call the `Update()` method to read the file.

```
reader->SetFileName("ellipse.meta");
reader->Update();
```

To get the objects in the file you can call the `GetScene()` method or the `GetGroup()` method. `GetScene()` returns an pointer to a `itk::SceneSpatialObject`.

```
ReaderType::SceneType * scene = reader->GetScene();
std::cout << "Number of objects in the scene: ";
std::cout << scene->GetNumberOfObjects() << std::endl;
ReaderType::GroupType * group = reader->GetGroup();
std::cout << "Number of objects in the group: ";
std::cout << group->GetNumberOfChildren() << std::endl;
```

5.8 Statistics Computation via SpatialObjects

The source code for this section can be found in the file `SpatialObjectToImageStatisticsCalculator.cxx`.

This example describes how to use the `itk::SpatialObjectToImageStatisticsCalculator` to compute statistics of an `itk::Image` only in a region defined inside a given `itk::SpatialObject`.

```
#include "itkSpatialObjectToImageStatisticsCalculator.h"
```

We first create a test image using the `itk::RandomImageSource`

```
typedef itk::Image< unsigned char, 2 >      ImageType;
typedef itk::RandomImageSource< ImageType > RandomImageSourceType;
RandomImageSourceType::Pointer randomImageSource
                                        = RandomImageSourceType::New();
ImageType::SizeValueType size[2];
size[0] = 10;
size[1] = 10;
randomImageSource->SetSize(size);
randomImageSource->Update();
ImageType::Pointer image = randomImageSource->GetOutput();
```

Next we create an `itk::EllipseSpatialObject` with a radius of 2. We also move the ellipse to the center of the image by increasing the offset of the IndexToObjectTransform.

```
typedef itk::EllipseSpatialObject<2> EllipseType;
EllipseType::Pointer ellipse = EllipseType::New();
ellipse->SetRadius(2);
EllipseType::VectorType offset;
offset.Fill(5);
ellipse->GetIndexToObjectTransform()->SetOffset(offset);
ellipse->ComputeObjectToParentTransform();
```

Then we can create the `itk::SpatialObjectToImageStatisticsCalculator`.

```
typedef itk::SpatialObjectToImageStatisticsCalculator<
  ImageType, EllipseType > CalculatorType;
CalculatorType::Pointer calculator = CalculatorType::New();
```

We pass a pointer to the image to the calculator.

```
calculator->SetImage(image);
```

We also pass the SpatialObject. The statistics will be computed inside the SpatialObject (Internally the calculator is using the `IsInside()` function).

```
calculator->SetSpatialObject(ellipse);
```

At the end we trigger the computation via the `Update()` function and we can retrieve the mean and the covariance matrix using `GetMean()` and `GetCovarianceMatrix()` respectively.

```
calculator->Update();
std::cout << "Sample mean = " << calculator->GetMean() << std::endl;
std::cout << "Sample covariance = " << calculator->GetCovarianceMatrix();
```

Part III

Development Guidelines

CHAPTER

SIX

ITERATORS

This chapter introduces the *image iterator*, an important generic programming construct for image processing in ITK. An iterator is a generalization of the familiar C programming language pointer used to reference data in memory. ITK has a wide variety of image iterators, some of which are highly specialized to simplify common image processing tasks.

The next section is a brief introduction that defines iterators in the context of ITK. Section 6.2 describes the programming interface common to most ITK image iterators. Sections 6.3–6.4 document specific ITK iterator types and provide examples of how they are used.

6.1 Introduction

Generic programming models define functionally independent components called *containers* and *algorithms*. Container objects store data and algorithms operate on data. To access data in containers, algorithms use a third class of objects called *iterators*. An iterator is an abstraction of a memory pointer. Every container type must define its own iterator type, but all iterators are written to provide a common interface so that algorithm code can reference data in a generic way and maintain functional independence from containers.

The iterator is so named because it is used for *iterative*, sequential access of container values. Iterators appear in `for` and `while` loop constructs, visiting each data point in turn. A C pointer, for example, is a type of iterator. It can be moved forward (incremented) and backward (decremented) through memory to sequentially reference elements of an array. Many iterator implementations have an interface similar to a C pointer.

In ITK we use iterators to write generic image processing code for images instantiated with different combinations of pixel type, pixel container type, and dimensionality. Because ITK image iterators are specifically designed to work with *image* containers, their interface and implementation is optimized for image processing tasks. Using the ITK iterators instead of accessing data directly through the `itk::Image` interface has many advantages. Code is more compact and often generalizes automatically to higher dimensions, algorithms run much faster, and iterators simplify tasks such as

multithreading and neighborhood-based image processing.

6.2 Programming Interface

This section describes the standard ITK image iterator programming interface. Some specialized image iterators may deviate from this standard or provide additional methods.

6.2.1 Creating Iterators

All image iterators have at least one template parameter that is the image type over which they iterate. There is no restriction on the dimensionality of the image or on the pixel type of the image.

An iterator constructor requires at least two arguments, a smart pointer to the image to iterate across, and an image region. The image region, called the *iteration region*, is a rectilinear area in which iteration is constrained. The iteration region must be wholly contained within the image. More specifically, a valid iteration region is any subregion of the image within the current `BufferedRegion`. See Section 4.1 for more information on image regions.

There is a const and a non-const version of most ITK image iterators. A non-const iterator cannot be instantiated on a non-const image pointer. Const versions of iterators may read, but may not write pixel values.

Here is a simple example that defines and constructs a simple image iterator for an `itk::Image`.

```
typedef itk::Image<float, 3> ImageType;
typedef itk::ImageRegionConstIterator< ImageType > ConstIteratorType;
typedef itk::ImageRegionIterator< ImageType > IteratorType;

ImageType::Pointer image = SomeFilter->GetOutput();

ConstIteratorType constIterator( image, image->GetRequestedRegion() );
IteratorType iterator( image, image->GetRequestedRegion() );
```

6.2.2 Moving Iterators

An iterator is described as *walking* its iteration region. At any time, the iterator will reference, or "point to", one pixel location in the N-dimensional (ND) image. *Forward iteration* goes from the beginning of the iteration region to the end of the iteration region. *Reverse iteration*, goes from just past the end of the region back to the beginning. There are two corresponding starting positions for iterators, the *begin* position and the *end* position. An iterator can be moved directly to either of these two positions using the following methods.

- **`GoToBegin()`** Points the iterator to the first valid data element in the region.

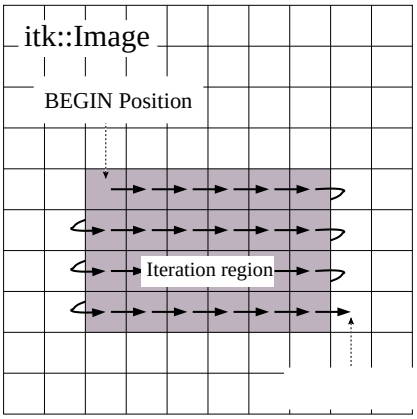

Figure 6.1: Normal path of an iterator through a 2D image. The iteration region is shown in a darker shade. An arrow denotes a single iterator step, the result of one ++ operation.

- **`GoToEnd()`** Points the iterator to *one position past* the last valid element in the region.

Note that the end position is not actually located within the iteration region. This is important to remember because attempting to dereference an iterator at its end position will have undefined results.

ITK iterators are moved back and forth across their iterations using the decrement and increment operators.

- **`operator++()`** Increments the iterator one position in the positive direction. Only the prefix increment operator is defined for ITK image iterators.
- **`operator--()`** Decrements the iterator one position in the negative direction. Only the prefix decrement operator is defined for ITK image iterators.

Figure 6.1 illustrates typical iteration over an image region. Most iterators increment and decrement in the direction of the fastest increasing image dimension, wrapping to the first position in the next higher dimension at region boundaries. In other words, an iterator first moves across columns, then down rows, then from slice to slice, and so on.

In addition to sequential iteration through the image, some iterators may define random access operators. Unlike the increment operators, random access operators may not be optimized for speed and require some knowledge of the dimensionality of the image and the extent of the iteration region to use properly.

- **`operator+=( OffsetType )`** Moves the iterator to the pixel position at the current index plus specified `itk::Offset`.

- **operator-=(OffsetType)** Moves the iterator to the pixel position at the current index minus specified Offset.
- **SetPosition(IndexType)** Moves the iterator to the given `itk::Index` position.

The `SetPosition()` method may be extremely slow for more complicated iterator types. In general, it should only be used for setting a starting iteration position, like you would use `GoToBegin()` or `GoToEnd()`.

Some iterators do not follow a predictable path through their iteration regions and have no fixed beginning or ending pixel locations. A conditional iterator, for example, visits pixels only if they have certain values or connectivities. Random iterators, increment and decrement to random locations and may even visit a given pixel location more than once.

An iterator can be queried to determine if it is at the end or the beginning of its iteration region.

- **bool IsAtEnd()** True if the iterator points to *one position past* the end of the iteration region.
- **bool IsAtBegin()** True if the iterator points to the first position in the iteration region. The method is typically used to test for the end of reverse iteration.

An iterator can also report its current image index position.

- **IndexType GetIndex()** Returns the Index of the image pixel that the iterator currently points to.

For efficiency, most ITK image iterators do not perform bounds checking. It is possible to move an iterator outside of its valid iteration region. Dereferencing an out-of-bounds iterator will produce undefined results.

6.2.3 Accessing Data

ITK image iterators define two basic methods for reading and writing pixel values.

- **PixelType Get()** Returns the value of the pixel at the iterator position.
- **void Set(PixelType)** Sets the value of the pixel at the iterator position. Not defined for const versions of iterators.

The `Get()` and `Set()` methods are inlined and optimized for speed so that their use is equivalent to dereferencing the image buffer directly. There are a few common cases, however, where using

`Get()` and `Set()` do incur a penalty. Consider the following code, which fetches, modifies, and then writes a value back to the same pixel location.

```
it.Set( it.Get() + 1 );
```

As written, this code requires one more memory dereference than is necessary. Some iterators define a third data access method that avoids this penalty.

- **`PixelType &Value()`** Returns a reference to the pixel at the iterator position.

The `Value()` method can be used as either an lval or an rval in an expression. It has all the properties of `operator*`. The `Value()` method makes it possible to rewrite our example code more efficiently.

```
it.Value()++;
```

Consider using the `Value()` method instead of `Get()` or `Set()` when a call to `operator=` on a pixel is non-trivial, such as when working with vector pixels, and operations are done in-place in the image. The disadvantage of using `Value` is that it cannot support image adapters (see Section 7 on page 179 for more information about image adaptors).

6.2.4 Iteration Loops

Using the methods described in the previous sections, we can now write a simple example to do pixel-wise operations on an image. The following code calculates the squares of all values in an input image and writes them to an output image.

```
ConstIteratorType in( inputImage,   inputImage->GetRequestedRegion() );
IteratorType out( outputImage, inputImage->GetRequestedRegion() );

for ( in.GoToBegin(), out.GoToBegin(); !in.IsAtEnd(); ++in, ++out )
  {
  out.Set( in.Get() * in.Get() );
  }
```

Notice that both the input and output iterators are initialized over the same region, the `RequestedRegion` of `inputImage`. This is good practice because it ensures that the output iterator walks exactly the same set of pixel indices as the input iterator, but does not require that the output and input be the same size. The only requirement is that the input image must contain a region (a starting index and size) that matches the `RequestedRegion` of the output image.

Equivalent code can be written by iterating through the image in reverse. The syntax is slightly more awkward because the *end* of the iteration region is not a valid position and we can only test whether the iterator is strictly *equal* to its beginning position. It is often more convenient to write reverse iteration in a `while` loop.

```
in.GoToEnd();
out.GoToEnd();
while ( ! in.IsAtBegin() )
  {
  --in;
  --out;
  out.Set( in.Get() * in.Get() );
  }
```

6.3 Image Iterators

This section describes iterators that walk rectilinear image regions and reference a single pixel at a time. The itk::ImageRegionIterator is the most basic ITK image iterator and the first choice for most applications. The rest of the iterators in this section are specializations of ImageRegionIterator that are designed make common image processing tasks more efficient or easier to implement.

6.3.1 ImageRegionIterator

The source code for this section can be found in the file ImageRegionIterator.cxx.

The itk::ImageRegionIterator is optimized for iteration speed and is the first choice for iterative, pixel-wise operations when location in the image is not important. ImageRegionIterator is the least specialized of the ITK image iterator classes. It implements all of the methods described in the preceding section.

The following example illustrates the use of itk::ImageRegionConstIterator and ImageRegionIterator. Most of the code constructs introduced apply to other ITK iterators as well. This simple application crops a subregion from an image by copying its pixel values into to a second, smaller image.

We begin by including the appropriate header files.

```
#include "itkImageRegionIterator.h"
```

Next we define a pixel type and corresponding image type. ITK iterator classes expect the image type as their template parameter.

```
const unsigned int Dimension = 2;

typedef unsigned char                     PixelType;
typedef itk::Image< PixelType, Dimension > ImageType;

typedef itk::ImageRegionConstIterator< ImageType > ConstIteratorType;
typedef itk::ImageRegionIterator< ImageType>       IteratorType;
```

Information about the subregion to copy is read from the command line. The subregion is defined by an `itk::ImageRegion` object, with a starting grid index and a size (Section 4.1).

```
ImageType::RegionType inputRegion;

ImageType::RegionType::IndexType inputStart;
ImageType::RegionType::SizeType  size;

inputStart[0] = ::atoi( argv[3] );
inputStart[1] = ::atoi( argv[4] );

size[0]  = ::atoi( argv[5] );
size[1]  = ::atoi( argv[6] );

inputRegion.SetSize( size );
inputRegion.SetIndex( inputStart );
```

The destination region in the output image is defined using the input region size, but a different start index. The starting index for the destination region is the corner of the newly generated image.

```
ImageType::RegionType outputRegion;

ImageType::RegionType::IndexType outputStart;

outputStart[0] = 0;
outputStart[1] = 0;

outputRegion.SetSize( size );
outputRegion.SetIndex( outputStart );
```

After reading the input image and checking that the desired subregion is, in fact, contained in the input, we allocate an output image. It is fundamental to set valid values to some of the basic image information during the copying process. In particular, the starting index of the output region is now filled up with zero values and the coordinates of the physical origin are computed as a shift from the origin of the input image. This is quite important since it will allow us to later register the extracted region against the original image.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( outputRegion );
const ImageType::SpacingType& spacing = reader->GetOutput()->GetSpacing();
const ImageType::PointType& inputOrigin = reader->GetOutput()->GetOrigin();
double   outputOrigin[ Dimension ];

for(unsigned int i=0; i< Dimension; i++)
  {
  outputOrigin[i] = inputOrigin[i] + spacing[i] * inputStart[i];
  }

outputImage->SetSpacing( spacing );
outputImage->SetOrigin(  outputOrigin );
outputImage->Allocate();
```

The necessary images and region definitions are now in place. All that is left to do is to create the iterators and perform the copy. Note that image iterators are not accessed via smart pointers so they are light-weight objects that are instantiated on the stack. Also notice how the input and output iterators are defined over the *same corresponding region*. Though the images are different sizes, they both contain the same target subregion.

```
ConstIteratorType inputIt(   reader->GetOutput(), inputRegion  );
IteratorType      outputIt(  outputImage,         outputRegion );

inputIt.GoToBegin();
outputIt.GoToBegin();

while( !inputIt.IsAtEnd() )
  {
  outputIt.Set(  inputIt.Get()  );
  ++inputIt;
  ++outputIt;
  }
```

The `while` loop above is a common construct in ITK. The beauty of these four lines of code is that they are equally valid for one, two, three, or even ten dimensional data, and no knowledge of the size of the image is necessary. Consider the ugly alternative of ten nested `for` loops for traversing an image.

Let's run this example on the image `FatMRISlice.png` found in `Examples/Data`. The command line arguments specify the input and output file names, then the x, y origin and the x, y size of the cropped subregion.

```
ImageRegionIterator FatMRISlice.png ImageRegionIteratorOutput.png 20 70 210 140
```

The output is the cropped subregion shown in Figure 6.2.

6.3.2 ImageRegionIteratorWithIndex

The source code for this section can be found in the file `ImageRegionIteratorWithIndex.cxx`.

The "WithIndex" family of iterators was designed for algorithms that use both the value and the location of image pixels in calculations. Unlike `itk::ImageRegionIterator`, which calculates an index only when asked for, `itk::ImageRegionIteratorWithIndex` maintains its index location as a member variable that is updated during the increment or decrement process. Iteration speed is penalized, but the index queries are more efficient.

The following example illustrates the use of ImageRegionIteratorWithIndex. The algorithm mirrors a 2D image across its x-axis (see `itk::FlipImageFilter` for an ND version). The algorithm makes extensive use of the `GetIndex()` method.

We start by including the proper header file.

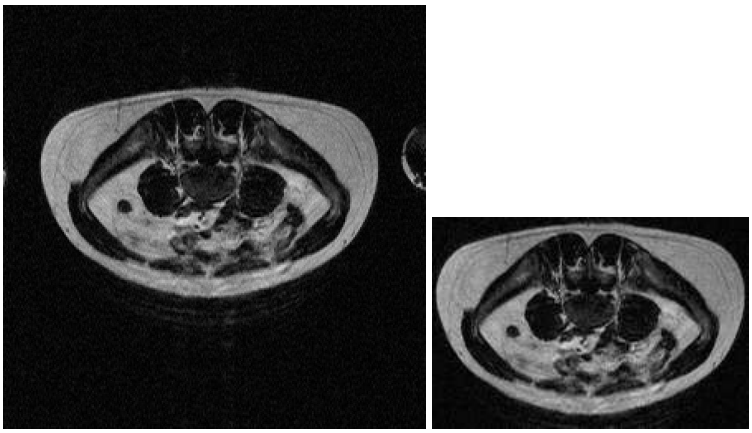

Figure 6.2: Cropping a region from an image. The original image is shown at left. The image on the right is the result of applying the ImageRegionIterator example code.

```
#include "itkImageRegionIteratorWithIndex.h"
```

For this example, we will use an RGB pixel type so that we can process color images. Like most other ITK image iterator, ImageRegionIteratorWithIndex class expects the image type as its single template parameter.

```
const unsigned int Dimension = 2;

typedef itk::RGBPixel< unsigned char >        RGBPixelType;
typedef itk::Image< RGBPixelType, Dimension > ImageType;

typedef itk::ImageRegionIteratorWithIndex< ImageType > IteratorType;
```

An `ImageType` smart pointer called `inputImage` points to the output of the image reader. After updating the image reader, we can allocate an output image of the same size, spacing, and origin as the input image.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( inputImage->GetRequestedRegion() );
outputImage->CopyInformation( inputImage );
outputImage->Allocate();
```

Next we create the iterator that walks the output image. This algorithm requires no iterator for the input image.

```
IteratorType outputIt( outputImage, outputImage->GetRequestedRegion() );
```

This axis flipping algorithm works by iterating through the output image, querying the iterator for

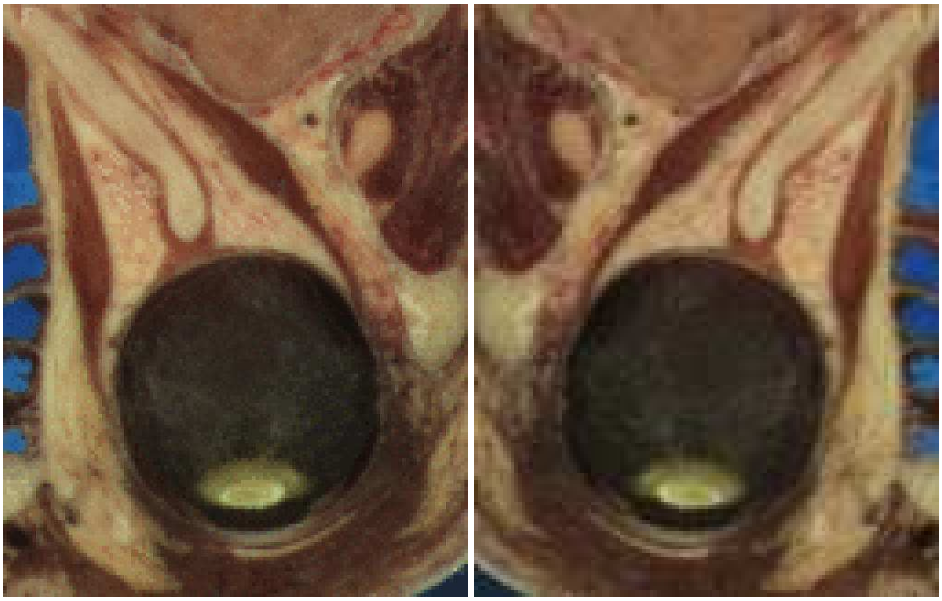

Figure 6.3: Results of using ImageRegionIteratorWithIndex to mirror an image across an axis. The original image is shown at left. The mirrored output is shown at right.

its index, and copying the value from the input at an index mirrored across the x-axis.

```
ImageType::IndexType requestedIndex =
             outputImage->GetRequestedRegion().GetIndex();
ImageType::SizeType requestedSize =
             outputImage->GetRequestedRegion().GetSize();

for ( outputIt.GoToBegin(); !outputIt.IsAtEnd(); ++outputIt)
  {
  ImageType::IndexType idx = outputIt.GetIndex();
  idx[0] =  requestedIndex[0] + requestedSize[0] - 1 - idx[0];
  outputIt.Set( inputImage->GetPixel(idx) );
  }
```

Let's run this example on the image `VisibleWomanEyeSlice.png` found in the `Examples/Data` directory. Figure 6.3 shows how the original image has been mirrored across its x-axis in the output.

6.3.3 ImageLinearIteratorWithIndex

The source code for this section can be found in the file `ImageLinearIteratorWithIndex.cxx`.

The `itk::ImageLinearIteratorWithIndex` is designed for line-by-line processing of an image.

It walks a linear path along a selected image direction parallel to one of the coordinate axes of the image. This iterator conceptually breaks an image into a set of parallel lines that span the selected image dimension.

Like all image iterators, movement of the ImageLinearIteratorWithIndex is constrained within an image region R. The line ℓ through which the iterator moves is defined by selecting a direction and an origin. The line ℓ extends from the origin to the upper boundary of R. The origin can be moved to any position along the lower boundary of R.

Several additional methods are defined for this iterator to control movement of the iterator along the line ℓ and movement of the origin of ℓ.

- **`NextLine()`** Moves the iterator to the beginning pixel location of the next line in the image. The origin of the next line is determined by incrementing the current origin along the fastest increasing dimension of the subspace of the image that excludes the selected dimension.
- **`PreviousLine()`** Moves the iterator to the *last valid pixel location* in the previous line. The origin of the previous line is determined by decrementing the current origin along the fastest increasing dimension of the subspace of the image that excludes the selected dimension.
- **`GoToBeginOfLine()`** Moves the iterator to the beginning pixel of the current line.
- **`GoToEndOfLine()`** Moves the iterator to *one past* the last valid pixel of the current line.
- **`GoToReverseBeginOfLine()`** Moves the iterator to *the last valid pixel* of the current line.
- **`IsAtReverseEndOfLine()`** Returns true if the iterator points to *one position before* the beginning pixel of the current line.
- **`IsAtEndOfLine()`** Returns true if the iterator points to *one position past* the last valid pixel of the current line.

The following code example shows how to use the ImageLinearIteratorWithIndex. It implements the same algorithm as in the previous example, flipping an image across its x-axis. Two line iterators are iterated in opposite directions across the x-axis. After each line is traversed, the iterator origins are stepped along the y-axis to the next line.

Headers for both the const and non-const versions are needed.

```
#include "itkImageLinearIteratorWithIndex.h"
```

The RGB image and pixel types are defined as in the previous example. The ImageLinearIterator-WithIndex class and its const version each have single template parameters, the image type.

```
typedef itk::ImageLinearIteratorWithIndex< ImageType >      IteratorType;
typedef itk::ImageLinearConstIteratorWithIndex<
                                        ImageType > ConstIteratorType;
```

After reading the input image, we allocate an output image that of the same size, spacing, and origin.

```
ImageType::Pointer outputImage = ImageType::New();
outputImage->SetRegions( inputImage->GetRequestedRegion() );
outputImage->CopyInformation( inputImage );
outputImage->Allocate();
```

Next we create the two iterators. The const iterator walks the input image, and the non-const iterator walks the output image. The iterators are initialized over the same region. The direction of iteration is set to 0, the x dimension.

```
ConstIteratorType inputIt( inputImage, inputImage->GetRequestedRegion() );
IteratorType outputIt( outputImage, inputImage->GetRequestedRegion() );

inputIt.SetDirection(0);
outputIt.SetDirection(0);
```

Each line in the input is copied to the output. The input iterator moves forward across columns while the output iterator moves backwards.

```
for ( inputIt.GoToBegin(),  outputIt.GoToBegin(); ! inputIt.IsAtEnd();
      outputIt.NextLine(),  inputIt.NextLine())
  {
  inputIt.GoToBeginOfLine();
  outputIt.GoToEndOfLine();
  while ( ! inputIt.IsAtEndOfLine() )
    {
    --outputIt;
    outputIt.Set( inputIt.Get() );
    ++inputIt;
    }
  }
```

Running this example on `VisibleWomanEyeSlice.png` produces the same output image shown in Figure 6.3.

The source code for this section can be found in the file `ImageLinearIteratorWithIndex2.cxx`.

This example shows how to use the `itk::ImageLinearIteratorWithIndex` for computing the mean across time of a 4D image where the first three dimensions correspond to spatial coordinates and the fourth dimension corresponds to time. The result of the mean across time is to be stored in a 3D image.

```
#include "itkImageLinearConstIteratorWithIndex.h"
```

First we declare the types of the images, the 3D and 4D readers.

```
typedef unsigned char              PixelType;
typedef itk::Image< PixelType, 3 >  Image3DType;
typedef itk::Image< PixelType, 4 >  Image4DType;

typedef itk::ImageFileReader< Image4DType > Reader4DType;
typedef itk::ImageFileWriter< Image3DType > Writer3DType;
```

Next, define the necessary types for indices, points, spacings, and size.

```
Image3DType::Pointer image3D = Image3DType::New();
typedef Image3DType::IndexType    Index3DType;
typedef Image3DType::SizeType     Size3DType;
typedef Image3DType::RegionType   Region3DType;
typedef Image3DType::SpacingType  Spacing3DType;
typedef Image3DType::PointType    Origin3DType;

typedef Image4DType::IndexType    Index4DType;
typedef Image4DType::SizeType     Size4DType;
typedef Image4DType::SpacingType  Spacing4DType;
typedef Image4DType::PointType    Origin4DType;
```

Here we make sure that the values for our resultant 3D mean image match up with the input 4D image.

```
for( unsigned int i=0; i < 3; i++)
  {
  size3D[i]    = size4D[i];
  index3D[i]   = index4D[i];
  spacing3D[i] = spacing4D[i];
  origin3D[i]  = origin4D[i];
  }

image3D->SetSpacing( spacing3D );
image3D->SetOrigin(  origin3D  );

Region3DType region3D;
region3D.SetIndex( index3D );
region3D.SetSize( size3D );

image3D->SetRegions( region3D  );
image3D->Allocate();
```

Next we iterate over time in the input image series, compute the average, and store that value in the corresponding pixel of the output 3D image.

```
IteratorType it( image4D, region4D );
it.SetDirection( 3 ); // Walk along time dimension
it.GoToBegin();
while( !it.IsAtEnd() )
  {
  SumType sum = itk::NumericTraits< SumType >::ZeroValue();
  it.GoToBeginOfLine();
  index4D = it.GetIndex();
  while( !it.IsAtEndOfLine() )
    {
    sum += it.Get();
    ++it;
    }
  MeanType mean = static_cast< MeanType >( sum ) /
                  static_cast< MeanType >( timeLength );

  index3D[0] = index4D[0];
  index3D[1] = index4D[1];
  index3D[2] = index4D[2];

  image3D->SetPixel( index3D, static_cast< PixelType >( mean ) );
  it.NextLine();
  }
```

As you can see, we avoid to use a 3D iterator to walk over the mean image. The reason is that there is no guarantee that the 3D iterator will walk in the same order as the 4D. Iterators just adhere to their contract of visiting every pixel, but do not enforce any particular order for the visits. The linear iterator guarantees it will visit the pixels along a line of the image in the order in which they are placed in the line, but does not state in what order one line will be visited with respect to other lines. Here we simply take advantage of knowing the first three components of the 4D iterator index, and use them to place the resulting mean value in the output 3D image.

6.3.4 ImageSliceIteratorWithIndex

The source code for this section can be found in the file `ImageSliceIteratorWithIndex.cxx`.

The `itk::ImageSliceIteratorWithIndex` class is an extension of `itk::ImageLinearIteratorWithIndex` from iteration along lines to iteration along both lines *and planes* in an image. A *slice* is a 2D plane spanned by two vectors pointing along orthogonal coordinate axes. The slice orientation of the slice iterator is defined by specifying its two spanning axes.

- **`SetFirstDirection()`** Specifies the first coordinate axis direction of the slice plane.
- **`SetSecondDirection()`** Specifies the second coordinate axis direction of the slice plane.

Several new methods control movement from slice to slice.

- **`NextSlice()`** Moves the iterator to the beginning pixel location of the next slice in the image. The origin of the next slice is calculated by incrementing the current origin index along the fastest increasing dimension of the image subspace which excludes the first and second dimensions of the iterator.
- **`PreviousSlice()`** Moves the iterator to the *last valid pixel location* in the previous slice. The origin of the previous slice is calculated by decrementing the current origin index along the fastest increasing dimension of the image subspace which excludes the first and second dimensions of the iterator.
- **`IsAtReverseEndOfSlice()`** Returns true if the iterator points to *one position before* the beginning pixel of the current slice.
- **`IsAtEndOfSlice()`** Returns true if the iterator points to *one position past* the last valid pixel of the current slice.

The slice iterator moves line by line using `NextLine()` and `PreviousLine()`. The line direction is parallel to the *second* coordinate axis direction of the slice plane (see also Section 6.3.3).

The next code example calculates the maximum intensity projection along one of the coordinate axes of an image volume. The algorithm is straightforward using ImageSliceIteratorWithIndex because we can coordinate movement through a slice of the 3D input image with movement through the 2D planar output.

Here is how the algorithm works. For each 2D slice of the input, iterate through all the pixels line by line. Copy a pixel value to the corresponding position in the 2D output image if it is larger than the value already contained there. When all slices have been processed, the output image is the desired maximum intensity projection.

We include a header for the const version of the slice iterator. For writing values to the 2D projection image, we use the linear iterator from the previous section. The linear iterator is chosen because it can be set to follow the same path in its underlying 2D image that the slice iterator follows over each slice of the 3D image.

```
#include "itkImageSliceConstIteratorWithIndex.h"
#include "itkImageLinearIteratorWithIndex.h"
```

The pixel type is defined as `unsigned short`. For this application, we need two image types, a 3D image for the input, and a 2D image for the intensity projection.

```
  typedef unsigned short             PixelType;
  typedef itk::Image< PixelType, 2 >  ImageType2D;
  typedef itk::Image< PixelType, 3 >  ImageType3D;
```

A slice iterator type is defined to walk the input image.

```
  typedef itk::ImageLinearIteratorWithIndex< ImageType2D > LinearIteratorType;
  typedef itk::ImageSliceConstIteratorWithIndex< ImageType3D
                                                 > SliceIteratorType;
```

The projection direction is read from the command line. The projection image will be the size of the 2D plane orthogonal to the projection direction. Its spanning vectors are the two remaining coordinate axes in the volume. These axes are recorded in the `direction` array.

```
unsigned int projectionDirection =
  static_cast<unsigned int>( ::atoi( argv[3] ) );

unsigned int i, j;
unsigned int direction[2];
for (i = 0, j = 0; i < 3; ++i )
  {
  if (i != projectionDirection)
    {
    direction[j] = i;
    j++;
    }
  }
```

The `direction` array is now used to define the projection image size based on the input image size. The output image is created so that its common dimension(s) with the input image are the same size. For example, if we project along the x axis of the input, the size and origin of the y axes of the input and output will match. This makes the code slightly more complicated, but prevents a counter-intuitive rotation of the output.

```
ImageType2D::RegionType region;
ImageType2D::RegionType::SizeType size;
ImageType2D::RegionType::IndexType index;

ImageType3D::RegionType requestedRegion = inputImage->GetRequestedRegion();

index[ direction[0] ]    = requestedRegion.GetIndex()[ direction[0] ];
index[ 1- direction[0] ] = requestedRegion.GetIndex()[ direction[1] ];
size[ direction[0] ]     = requestedRegion.GetSize()[  direction[0] ];
size[ 1- direction[0] ]  = requestedRegion.GetSize()[  direction[1] ];

region.SetSize( size );
region.SetIndex( index );

ImageType2D::Pointer outputImage = ImageType2D::New();

outputImage->SetRegions( region );
outputImage->Allocate();
```

Next we create the necessary iterators. The const slice iterator walks the 3D input image, and the non-const linear iterator walks the 2D output image. The iterators are initialized to walk the same linear path through a slice. Remember that the *second* direction of the slice iterator defines the direction that linear iteration walks within a slice.

```
SliceIteratorType  inputIt(  inputImage, inputImage->GetRequestedRegion() );
LinearIteratorType outputIt( outputImage,
                                        outputImage->GetRequestedRegion() );

inputIt.SetFirstDirection(  direction[1] );
inputIt.SetSecondDirection( direction[0] );

outputIt.SetDirection( 1 - direction[0] );
```

Now we are ready to compute the projection. The first step is to initialize all of the projection values to their nonpositive minimum value. The projection values are then updated row by row from the first slice of the input. At the end of the first slice, the input iterator steps to the first row in the next slice, while the output iterator, whose underlying image consists of only one slice, rewinds to its first row. The process repeats until the last slice of the input is processed.

```
outputIt.GoToBegin();
while ( ! outputIt.IsAtEnd() )
  {
  while ( ! outputIt.IsAtEndOfLine() )
    {
    outputIt.Set( itk::NumericTraits<unsigned short>::NonpositiveMin() );
    ++outputIt;
    }
  outputIt.NextLine();
  }

inputIt.GoToBegin();
outputIt.GoToBegin();

while( !inputIt.IsAtEnd() )
  {
  while ( !inputIt.IsAtEndOfSlice() )
    {
    while ( !inputIt.IsAtEndOfLine() )
      {
      outputIt.Set( vnl_math_max( outputIt.Get(), inputIt.Get() ));
      ++inputIt;
      ++outputIt;
      }
    outputIt.NextLine();
    inputIt.NextLine();

    }
  outputIt.GoToBegin();
  inputIt.NextSlice();
  }
```

Running this example code on the 3D image `Examples/Data/BrainProtonDensity3Slices.mha` using the z-axis as the axis of projection gives the image shown in Figure 6.4.

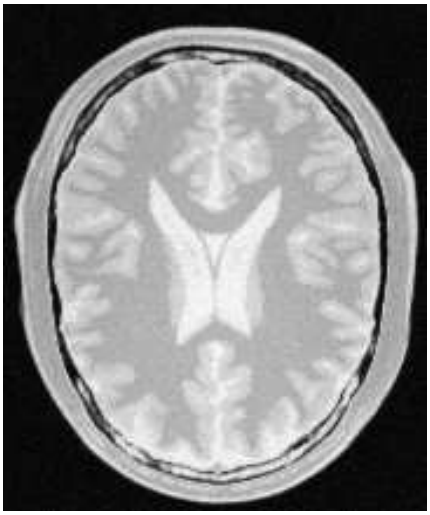

Figure 6.4: The maximum intensity projection through three slices of a volume.

6.3.5 ImageRandomConstIteratorWithIndex

The source code for this section can be found in the file `ImageRandomConstIteratorWithIndex.cxx`.

`itk::ImageRandomConstIteratorWithIndex` was developed to randomly sample pixel values. When incremented or decremented, it jumps to a random location in its image region.

The user must specify a sample size when creating this iterator. The sample size, rather than a specific image index, defines the end position for the iterator. `IsAtEnd()` returns `true` when the current sample number equals the sample size. `IsAtBegin()` returns `true` when the current sample number equals zero. An important difference from other image iterators is that ImageRandomConstIteratorWithIndex may visit the same pixel more than once.

Let's use the random iterator to estimate some simple image statistics. The next example calculates an estimate of the arithmetic mean of pixel values.

First, include the appropriate header and declare pixel and image types.

```
#include "itkImageRandomConstIteratorWithIndex.h"

  const unsigned int Dimension = 2;

  typedef unsigned short                                    PixelType;
  typedef itk::Image< PixelType, Dimension >                ImageType;
  typedef itk::ImageRandomConstIteratorWithIndex<
                                      ImageType > ConstIteratorType;
```

The input image has been read as `inputImage`. We now create an iterator with a number of samples

	Sample Size			
	10	**100**	**1000**	**10000**
RatLungSlice1.mha	50.5	52.4	53.0	52.4
RatLungSlice2.mha	46.7	47.5	47.4	47.6
BrainT1Slice.png	47.2	64.1	68.0	67.8

Table 6.1: Estimates of mean image pixel value using the ImageRandomConstIteratorWithIndex at different sample sizes.

set by command line argument. The call to `ReinitializeSeed` seeds the random number generator. The iterator is initialized over the entire valid image region.

```
ConstIteratorType inputIt(  inputImage,  inputImage->GetRequestedRegion() );
inputIt.SetNumberOfSamples( ::atoi( argv[2]) );
inputIt.ReinitializeSeed();
```

Now take the specified number of samples and calculate their average value.

```
float mean = 0.0f;
for ( inputIt.GoToBegin(); ! inputIt.IsAtEnd(); ++inputIt)
  {
  mean += static_cast<float>( inputIt.Get() );
  }
mean = mean / ::atof( argv[2] );
```

The following table shows the results of running this example on several of the data files from `Examples/Data` with a range of sample sizes.

6.4 Neighborhood Iterators

In ITK, a pixel neighborhood is loosely defined as a small set of pixels that are locally adjacent to one another in an image. The size and shape of a neighborhood, as well the connectivity among pixels in a neighborhood, may vary with the application.

Many image processing algorithms are neighborhood-based, that is, the result at a pixel i is computed from the values of pixels in the ND neighborhood of i. Consider finite difference operations in 2D. A derivative at pixel index $i = (j,k)$, for example, is taken as a weighted difference of the values at $(j+1,k)$ and $(j-1,k)$. Other common examples of neighborhood operations include convolution filtering and image morphology.

This section describes a class of ITK image iterators that are designed for working with pixel neighborhoods. An ITK neighborhood iterator walks an image region just like a normal image iterator, but instead of only referencing a single pixel at each step, it simultaneously points to the entire ND neighborhood of pixels. Extensions to the standard iterator interface provide read and write access to

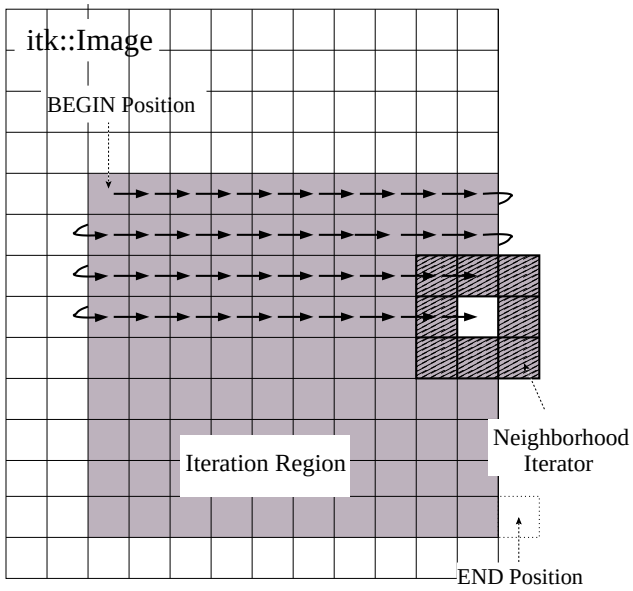

Figure 6.5: Path of a $3x3$ neighborhood iterator through a 2D image region. The extent of the neighborhood is indicated by the hashing around the iterator position. Pixels that lie within this extent are accessible through the iterator. An arrow denotes a single iterator step, the result of one ++ operation.

all neighborhood pixels and information such as the size, extent, and location of the neighborhood.

Neighborhood iterators use the same operators defined in Section 6.2 and the same code constructs as normal iterators for looping through an image. Figure 6.5 shows a neighborhood iterator moving through an iteration region. This iterator defines a $3x3$ neighborhood around each pixel that it visits. The *center* of the neighborhood iterator is always positioned over its current index and all other neighborhood pixel indices are referenced as offsets from the center index. The pixel under the center of the neighborhood iterator and all pixels under the shaded area, or *extent*, of the iterator can be dereferenced.

In addition to the standard image pointer and iteration region (Section 6.2), neighborhood iterator constructors require an argument that specifies the extent of the neighborhood to cover. Neighborhood extent is symmetric across its center in each axis and is given as an array of N distances that are collectively called the *radius*. Each element d of the radius, where $0 < d < N$ and N is the dimensionality of the neighborhood, gives the extent of the neighborhood in pixels for dimension N. The length of each face of the resulting ND hypercube is $2d+1$ pixels, a distance of d on either side of the single pixel at the neighbor center. Figure 6.6 shows the relationship between the radius of the iterator and the size of the neighborhood for a variety of 2D iterator shapes.

The radius of the neighborhood iterator is queried after construction by calling the `GetRadius()` method. Some other methods provide some useful information about the iterator and its underlying

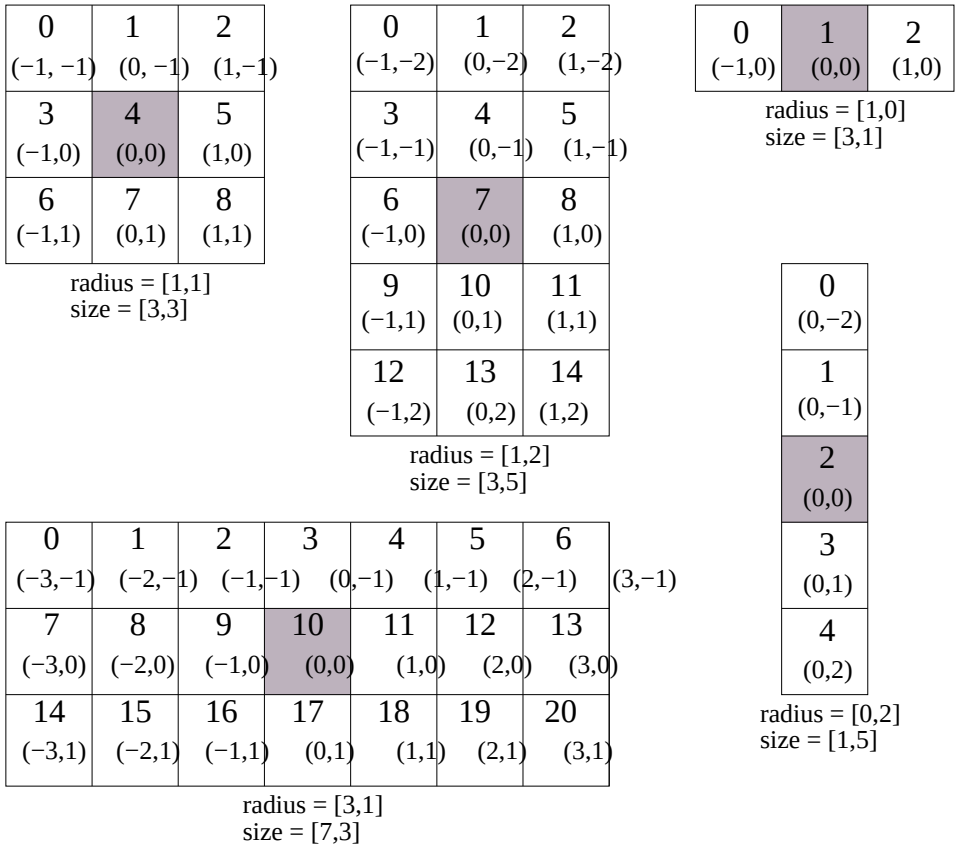

Figure 6.6: Several possible 2D neighborhood iterator shapes are shown along with their radii and sizes. A neighborhood pixel can be dereferenced by its integer index (top) or its offset from the center (bottom). The center pixel of each iterator is shaded.

image.

- **`SizeType GetRadius()`** Returns the ND radius of the neighborhood as an `itk::Size`.
- **`const ImageType *GetImagePointer()`** Returns the pointer to the image referenced by the iterator.
- **`unsigned long Size()`** Returns the size in number of pixels of the neighborhood.

The neighborhood iterator interface extends the normal ITK iterator interface for setting and getting pixel values. One way to dereference pixels is to think of the neighborhood as a linear array where

each pixel has a unique integer index. The index of a pixel in the array is determined by incrementing from the upper-left-forward corner of the neighborhood along the fastest increasing image dimension: first column, then row, then slice, and so on. In Figure 6.6, the unique integer index is shown at the top of each pixel. The center pixel is always at position $n/2$, where n is the size of the array.

- **`PixelType GetPixel(const unsigned int i)`** Returns the value of the pixel at neighborhood position `i`.
- **`void SetPixel(const unsigned int i, PixelType p)`** Sets the value of the pixel at position `i` to `p`.

Another way to think about a pixel location in a neighborhood is as an ND offset from the neighborhood center. The upper-left-forward corner of a $3x3x3$ neighborhood, for example, can be described by offset $(-1,-1,-1)$. The bottom-right-back corner of the same neighborhood is at offset $(1,1,1)$. In Figure 6.6, the offset from center is shown at the bottom of each neighborhood pixel.

- **`PixelType GetPixel(const OffsetType &o)`** Get the value of the pixel at the position offset `o` from the neighborhood center.
- **`void SetPixel(const OffsetType &o, PixelType p)`** Set the value at the position offset `o` from the neighborhood center to the value `p`.

The neighborhood iterators also provide a shorthand for setting and getting the value at the center of the neighborhood.

- **`PixelType GetCenterPixel()`** Gets the value at the center of the neighborhood.
- **`void SetCenterPixel(PixelType p)`** Sets the value at the center of the neighborhood to the value `p`

There is another shorthand for setting and getting values for pixels that lie some integer distance from the neighborhood center along one of the image axes.

- **`PixelType GetNext(unsigned int d)`** Get the value immediately adjacent to the neighborhood center in the positive direction along the `d` axis.
- **`void SetNext(unsigned int d, PixelType p)`** Set the value immediately adjacent to the neighborhood center in the positive direction along the `d` axis to the value `p`.
- **`PixelType GetPrevious(unsigned int d)`** Get the value immediately adjacent to the neighborhood center in the negative direction along the `d` axis.

- **void SetPrevious(unsigned int d, PixelType p)** Set the value immediately adjacent to the neighborhood center in the negative direction along the d axis to the value p.
- **PixelType GetNext(unsigned int d, unsigned int s)** Get the value of the pixel located s pixels from the neighborhood center in the positive direction along the d axis.
- **void SetNext(unsigned int d, unsigned int s, PixelType p)** Set the value of the pixel located s pixels from the neighborhood center in the positive direction along the d axis to value p.
- **PixelType GetPrevious(unsigned int d, unsigned int s)** Get the value of the pixel located s pixels from the neighborhood center in the positive direction along the d axis.
- **void SetPrevious(unsigned int d, unsigned int s, PixelType p)** Set the value of the pixel located s pixels from the neighborhood center in the positive direction along the d axis to value p.

It is also possible to extract or set all of the neighborhood values from an iterator at once using a regular ITK neighborhood object. This may be useful in algorithms that perform a particularly large number of calculations in the neighborhood and would otherwise require multiple dereferences of the same pixels.

- **NeighborhoodType GetNeighborhood()** Return a itk::Neighborhood of the same size and shape as the neighborhood iterator and contains all of the values at the iterator position.
- **void SetNeighborhood(NeighborhoodType &N)** Set all of the values in the neighborhood at the iterator position to those contained in Neighborhood N, which must be the same size and shape as the iterator.

Several methods are defined to provide information about the neighborhood.

- **IndexType GetIndex()** Return the image index of the center pixel of the neighborhood iterator.
- **IndexType GetIndex(OffsetType o)** Return the image index of the pixel at offset o from the neighborhood center.
- **IndexType GetIndex(unsigned int i)** Return the image index of the pixel at array position i.
- **OffsetType GetOffset(unsigned int i)** Return the offset from the neighborhood center of the pixel at array position i.

- **`unsigned long GetNeighborhoodIndex(OffsetType o)`** Return the array position of the pixel at offset `o` from the neighborhood center.
- **`std::slice GetSlice(unsigned int n)`** Return a `std::slice` through the iterator neighborhood along axis `n`.

A neighborhood-based calculation in a neighborhood close to an image boundary may require data that falls outside the boundary. The iterator in Figure 6.5, for example, is centered on a boundary pixel such that three of its neighbors actually do not exist in the image. When the extent of a neighborhood falls outside the image, pixel values for missing neighbors are supplied according to a rule, usually chosen to satisfy the numerical requirements of the algorithm. A rule for supplying out-of-bounds values is called a *boundary condition*.

ITK neighborhood iterators automatically detect out-of-bounds dereferences and will return values according to boundary conditions. The boundary condition type is specified by the second, optional template parameter of the iterator. By default, neighborhood iterators use a Neumann condition where the first derivative across the boundary is zero. The Neumann rule simply returns the closest in-bounds pixel value to the requested out-of-bounds location. Several other common boundary conditions can be found in the ITK toolkit. They include a periodic condition that returns the pixel value from the opposite side of the data set, and is useful when working with periodic data such as Fourier transforms, and a constant value condition that returns a set value v for all out-of-bounds pixel dereferences. The constant value condition is equivalent to padding the image with value v.

Bounds checking is a computationally expensive operation because it occurs each time the iterator is incremented. To increase efficiency, a neighborhood iterator automatically disables bounds checking when it detects that it is not necessary. A user may also explicitly disable or enable bounds checking. Most neighborhood based algorithms can minimize the need for bounds checking through clever definition of iteration regions. These techniques are explored in Section 6.4.1.

- **`void NeedToUseBoundaryConditionOn()`** Explicitly turn bounds checking on. This method should be used with caution because unnecessarily enabling bounds checking may result in a significant performance decrease. In general you should allow the iterator to automatically determine this setting.
- **`void NeedToUseBoundaryConditionOff()`** Explicitly disable bounds checking. This method should be used with caution because disabling bounds checking when it is needed will result in out-of-bounds reads and undefined results.
- **`void OverrideBoundaryCondition(BoundaryConditionType *b)`** Overrides the templated boundary condition, using boundary condition object `b` instead. Object `b` should not be deleted until it has been released by the iterator. This method can be used to change iterator behavior at run-time.
- **`void ResetBoundaryCondition()`** Discontinues the use of any run-time specified boundary condition and returns to using the condition specified in the template argument.

- **`void SetPixel(unsigned int i, PixelType p, bool status)`** Sets the value at neighborhood array position `i` to value `p`. If the position `i` is out-of-bounds, `status` is set to `false`, otherwise `status` is set to `true`.

The following sections describe the two ITK neighborhood iterator classes, `itk::NeighborhoodIterator` and `itk::ShapedNeighborhoodIterator`. Each has a const and a non-const version. The shaped iterator is a refinement of the standard NeighborhoodIterator that supports an arbitrarily-shaped (non-rectilinear) neighborhood.

6.4.1 NeighborhoodIterator

The standard neighborhood iterator class in ITK is the `itk::NeighborhoodIterator`. Together with its `const` version, `itk::ConstNeighborhoodIterator`, it implements the complete API described above. This section provides several examples to illustrate the use of NeighborhoodIterator.

Basic neighborhood techniques: edge detection

The source code for this section can be found in the file `NeighborhoodIterators1.cxx`.

This example uses the `itk::NeighborhoodIterator` to implement a simple Sobel edge detection algorithm [4]. The algorithm uses the neighborhood iterator to iterate through an input image and calculate a series of finite difference derivatives. Since the derivative results cannot be written back to the input image without affecting later calculations, they are written instead to a second, output image. Most neighborhood processing algorithms follow this read-only model on their inputs.

We begin by including the proper header files. The `itk::ImageRegionIterator` will be used to write the results of computations to the output image. A const version of the neighborhood iterator is used because the input image is read-only.

```
#include "itkConstNeighborhoodIterator.h"
#include "itkImageRegionIterator.h"
```

The finite difference calculations in this algorithm require floating point values. Hence, we define the image pixel type to be `float` and the file reader will automatically cast fixed-point data to `float`.

We declare the iterator types using the image type as the template parameter. The second template parameter of the neighborhood iterator, which specifies the boundary condition, has been omitted because the default condition is appropriate for this algorithm.

```
  typedef float                           PixelType;
  typedef itk::Image< PixelType, 2 >      ImageType;
  typedef itk::ImageFileReader< ImageType > ReaderType;

  typedef itk::ConstNeighborhoodIterator< ImageType > NeighborhoodIteratorType;
  typedef itk::ImageRegionIterator< ImageType>        IteratorType;
```

The following code creates and executes the ITK image reader. The `Update` call on the reader object is surrounded by the standard `try/catch` blocks to handle any exceptions that may be thrown by the reader.

```
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
try
  {
  reader->Update();
  }
catch ( itk::ExceptionObject &err)
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return -1;
  }
```

We can now create a neighborhood iterator to range over the output of the reader. For Sobel edge-detection in 2D, we need a square iterator that extends one pixel away from the neighborhood center in every dimension.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(1);
NeighborhoodIteratorType it( radius, reader->GetOutput(),
                             reader->GetOutput()->GetRequestedRegion() );
```

The following code creates an output image and iterator.

```
ImageType::Pointer output = ImageType::New();
output->SetRegions(reader->GetOutput()->GetRequestedRegion());
output->Allocate();

IteratorType out(output, reader->GetOutput()->GetRequestedRegion());
```

Sobel edge detection uses weighted finite difference calculations to construct an edge magnitude image. Normally the edge magnitude is the root sum of squares of partial derivatives in all directions, but for simplicity this example only calculates the x component. The result is a derivative image biased toward maximally vertical edges.

The finite differences are computed from pixels at six locations in the neighborhood. In this example, we use the iterator `GetPixel()` method to query the values from their offsets in the neighborhood. The example in Section 6.4.1 uses convolution with a Sobel kernel instead.

Six positions in the neighborhood are necessary for the finite difference calculations. These positions are recorded in `offset1` through `offset6`.

```
NeighborhoodIteratorType::OffsetType offset1 = {{-1,-1}};
NeighborhoodIteratorType::OffsetType offset2 = {{1,-1}};
NeighborhoodIteratorType::OffsetType offset3 = {{-1,0 }};
NeighborhoodIteratorType::OffsetType offset4 = {{1,0}};
NeighborhoodIteratorType::OffsetType offset5 = {{-1,1}};
NeighborhoodIteratorType::OffsetType offset6 = {{1,1}};
```

It is equivalent to use the six corresponding integer array indices instead. For example, the offsets `(-1,-1)` and `(1, -1)` are equivalent to the integer indices `0` and `2`, respectively.

The calculations are done in a `for` loop that moves the input and output iterators synchronously across their respective images. The `sum` variable is used to sum the results of the finite differences.

```
for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
  {
  float sum;
  sum = it.GetPixel(offset2) - it.GetPixel(offset1);
  sum += 2.0 * it.GetPixel(offset4) - 2.0 * it.GetPixel(offset3);
  sum += it.GetPixel(offset6) - it.GetPixel(offset5);
  out.Set(sum);
  }
```

The last step is to write the output buffer to an image file. Writing is done inside a `try/catch` block to handle any exceptions. The output is rescaled to intensity range $[0, 255]$ and cast to unsigned char so that it can be saved and visualized as a PNG image.

```
typedef unsigned char                           WritePixelType;
typedef itk::Image< WritePixelType, 2 >        WriteImageType;
typedef itk::ImageFileWriter< WriteImageType > WriterType;

typedef itk::RescaleIntensityImageFilter<
            ImageType, WriteImageType > RescaleFilterType;

RescaleFilterType::Pointer rescaler = RescaleFilterType::New();

rescaler->SetOutputMinimum(   0 );
rescaler->SetOutputMaximum( 255 );
rescaler->SetInput(output);

WriterType::Pointer writer = WriterType::New();
writer->SetFileName( argv[2] );
writer->SetInput(rescaler->GetOutput());
try
  {
  writer->Update();
  }
catch ( itk::ExceptionObject &err)
  {
  std::cout << "ExceptionObject caught !" << std::endl;
  std::cout << err << std::endl;
  return -1;
  }
```

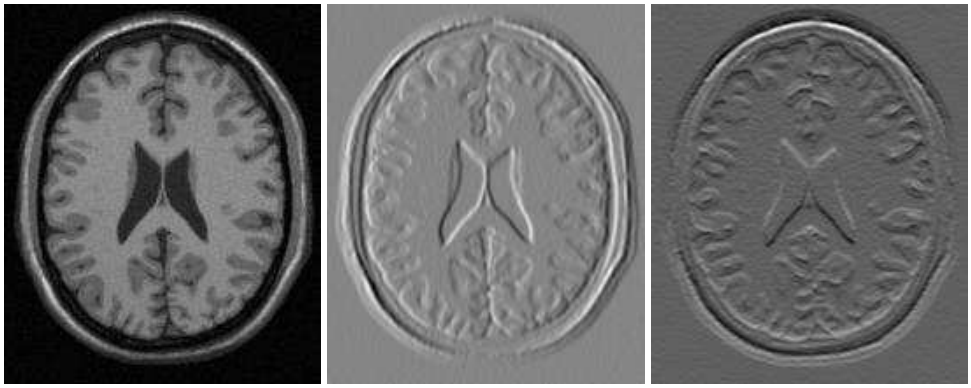

Figure 6.7: Applying the Sobel operator in different orientations to an MRI image (left) produces x (center) and y (right) derivative images.

The center image of Figure 6.7 shows the output of the Sobel algorithm applied to `Examples/Data/BrainT1Slice.png`.

Convolution filtering: Sobel operator

The source code for this section can be found in the file `NeighborhoodIterators2.cxx`.

In this example, the Sobel edge-detection routine is rewritten using convolution filtering. Convolution filtering is a standard image processing technique that can be implemented numerically as the inner product of all image neighborhoods with a convolution kernel [4] [2]. In ITK, we use a class of objects called *neighborhood operators* as convolution kernels and a special function object called `itk::NeighborhoodInnerProduct` to calculate inner products.

The basic ITK convolution filtering routine is to step through the image with a neighborhood iterator and use NeighborhoodInnerProduct to find the inner product of each neighborhood with the desired kernel. The resulting values are written to an output image. This example uses a neighborhood operator called the `itk::SobelOperator`, but all neighborhood operators can be convolved with images using this basic routine. Other examples of neighborhood operators include derivative kernels, Gaussian kernels, and morphological operators. `itk::NeighborhoodOperatorImageFilter` is a generalization of the code in this section to ND images and arbitrary convolution kernels.

We start writing this example by including the header files for the Sobel kernel and the inner product function.

```
#include "itkSobelOperator.h"
#include "itkNeighborhoodInnerProduct.h"
```

Refer to the previous example for a description of reading the input image and setting up the output

image and iterator.

The following code creates a Sobel operator. The Sobel operator requires a direction for its partial derivatives. This direction is read from the command line. Changing the direction of the derivatives changes the bias of the edge detection, i.e. maximally vertical or maximally horizontal.

```
itk::SobelOperator<PixelType, 2> sobelOperator;
sobelOperator.SetDirection( ::atoi(argv[3]) );
sobelOperator.CreateDirectional();
```

The neighborhood iterator is initialized as before, except that now it takes its radius directly from the radius of the Sobel operator. The inner product function object is templated over image type and requires no initialization.

```
NeighborhoodIteratorType::RadiusType radius = sobelOperator.GetRadius();
NeighborhoodIteratorType it( radius, reader->GetOutput(),
                             reader->GetOutput()->GetRequestedRegion() );

itk::NeighborhoodInnerProduct<ImageType> innerProduct;
```

Using the Sobel operator, inner product, and neighborhood iterator objects, we can now write a very simple `for` loop for performing convolution filtering. As before, out-of-bounds pixel values are supplied automatically by the iterator.

```
for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
  {
  out.Set( innerProduct( it, sobelOperator ) );
  }
```

The output is rescaled and written as in the previous example. Applying this example in the x and y directions produces the images at the center and right of Figure 6.7. Note that x-direction operator produces the same output image as in the previous example.

Optimizing iteration speed

The source code for this section can be found in the file `NeighborhoodIterators3.cxx`.

This example illustrates a technique for improving the efficiency of neighborhood calculations by eliminating unnecessary bounds checking. As described in Section 6.4, the neighborhood iterator automatically enables or disables bounds checking based on the iteration region in which it is initialized. By splitting our image into boundary and non-boundary regions, and then processing each region using a different neighborhood iterator, the algorithm will only perform bounds-checking on those pixels for which it is actually required. This trick can provide a significant speedup for simple algorithms such as our Sobel edge detection, where iteration speed is a critical.

Splitting the image into the necessary regions is an easy task when you use the `itk::ImageBoundaryFacesCalculator`. The face calculator is so named because it returns a list

of the "faces" of the ND dataset. Faces are those regions whose pixels all lie within a distance d from the boundary, where d is the radius of the neighborhood stencil used for the numerical calculations. In other words, faces are those regions where a neighborhood iterator of radius d will always overlap the boundary of the image. The face calculator also returns the single *inner* region, in which out-of-bounds values are never required and bounds checking is not necessary.

The face calculator object is defined in `itkNeighborhoodAlgorithm.h`. We include this file in addition to those from the previous two examples.

```
#include "itkNeighborhoodAlgorithm.h"
```

First we load the input image and create the output image and inner product function as in the previous examples. The image iterators will be created in a later step. Next we create a face calculator object. An empty list is created to hold the regions that will later on be returned by the face calculator.

```
  typedef itk::NeighborhoodAlgorithm
    ::ImageBoundaryFacesCalculator< ImageType > FaceCalculatorType;

  FaceCalculatorType faceCalculator;
  FaceCalculatorType::FaceListType faceList;
```

The face calculator function is invoked by passing it an image pointer, an image region, and a neighborhood radius. The image pointer is the same image used to initialize the neighborhood iterator, and the image region is the region that the algorithm is going to process. The radius is the radius of the iterator.

Notice that in this case the image region is given as the region of the *output* image and the image pointer is given as that of the *input* image. This is important if the input and output images differ in size, i.e. the input image is larger than the output image. ITK image filters, for example, operate on data from the input image but only generate results in the `RequestedRegion` of the output image, which may be smaller than the full extent of the input.

```
  faceList = faceCalculator(reader->GetOutput(), output->GetRequestedRegion(),
                            sobelOperator.GetRadius());
```

The face calculator has returned a list of $2N + 1$ regions. The first element in the list is always the inner region, which may or may not be important depending on the application. For our purposes it does not matter because all regions are processed the same way. We use an iterator to traverse the list of faces.

```
  FaceCalculatorType::FaceListType::iterator fit;
```

We now rewrite the main loop of the previous example so that each region in the list is processed by a separate iterator. The iterators `it` and `out` are reinitialized over each region in turn. Bounds checking is automatically enabled for those regions that require it, and disabled for the region that does not.

```
  IteratorType out;
  NeighborhoodIteratorType it;

  for ( fit=faceList.begin(); fit != faceList.end(); ++fit)
    {
    it = NeighborhoodIteratorType( sobelOperator.GetRadius(),
                                   reader->GetOutput(), *fit );
    out = IteratorType( output, *fit );

    for (it.GoToBegin(), out.GoToBegin(); ! it.IsAtEnd(); ++it, ++out)
      {
      out.Set( innerProduct(it, sobelOperator) );
      }
    }
```

The output is written as before. Results for this example are the same as the previous example. You may not notice the speedup except on larger images. When moving to 3D and higher dimensions, the effects are greater because the volume to surface area ratio is usually larger. In other words, as the number of interior pixels increases relative to the number of face pixels, there is a corresponding increase in efficiency from disabling bounds checking on interior pixels.

Separable convolution: Gaussian filtering

The source code for this section can be found in the file `NeighborhoodIterators4.cxx`.

We now introduce a variation on convolution filtering that is useful when a convolution kernel is separable. In this example, we create a different neighborhood iterator for each axial direction of the image and then take separate inner products with a 1D discrete Gaussian kernel. The idea of using several neighborhood iterators at once has applications beyond convolution filtering and may improve efficiency when the size of the whole neighborhood relative to the portion of the neighborhood used in calculations becomes large.

The only new class necessary for this example is the Gaussian operator.

```
#include "itkGaussianOperator.h"
```

The Gaussian operator, like the Sobel operator, is instantiated with a pixel type and a dimensionality. Additionally, we set the variance of the Gaussian, which has been read from the command line as standard deviation.

```
  itk::GaussianOperator< PixelType, 2 > gaussianOperator;
  gaussianOperator.SetVariance( ::atof(argv[3]) * ::atof(argv[3]) );
```

The only further changes from the previous example are in the main loop. Once again we use the results from face calculator to construct a loop that processes boundary and non-boundary image regions separately. Separable convolution, however, requires an additional, outer loop over all the image dimensions. The direction of the Gaussian operator is reset at each iteration of the outer loop

using the new dimension. The iterators change direction to match because they are initialized with the radius of the Gaussian operator.

Input and output buffers are swapped at each iteration so that the output of the previous iteration becomes the input for the current iteration. The swap is not performed on the last iteration.

```
ImageType::Pointer input = reader->GetOutput();
for (unsigned int i = 0; i < ImageType::ImageDimension; ++i)
  {
  gaussianOperator.SetDirection(i);
  gaussianOperator.CreateDirectional();

  faceList = faceCalculator(input, output->GetRequestedRegion(),
                            gaussianOperator.GetRadius());

  for ( fit=faceList.begin(); fit != faceList.end(); ++fit )
    {
    it = NeighborhoodIteratorType( gaussianOperator.GetRadius(),
                                   input, *fit );

    out = IteratorType( output, *fit );

    for (it.GoToBegin(), out.GoToBegin(); ! it.IsAtEnd(); ++it, ++out)
      {
      out.Set( innerProduct(it, gaussianOperator) );
      }
    }

  // Swap the input and output buffers
  if (i != ImageType::ImageDimension - 1)
    {
    ImageType::Pointer tmp = input;
    input = output;
    output = tmp;
    }
  }
```

The output is rescaled and written as in the previous examples. Figure 6.8 shows the results of Gaussian blurring the image `Examples/Data/BrainT1Slice.png` using increasing kernel widths.

Slicing the neighborhood

The source code for this section can be found in the file `NeighborhoodIterators5.cxx`.

This example introduces slice-based neighborhood processing. A slice, in this context, is a 1D path through an ND neighborhood. Slices are defined for generic arrays by the `std::slice` class as a start index, a step size, and an end index. Slices simplify the implementation of certain neighborhood calculations. They also provide a mechanism for taking inner products with subregions of neighborhoods.

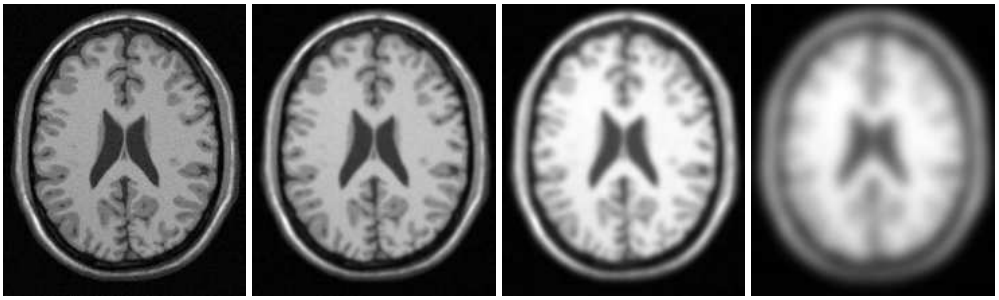

Figure 6.8: Results of convolution filtering with a Gaussian kernel of increasing standard deviation σ (from left to right, $\sigma = 0$, $\sigma = 1$, $\sigma = 2$, $\sigma = 5$). Increased blurring reduces contrast and changes the average intensity value of the image, which causes the image to appear brighter when rescaled.

Suppose, for example, that we want to take partial derivatives in the y direction of a neighborhood, but offset those derivatives by one pixel position along the positive x direction. For a 3×3, 2D neighborhood iterator, we can construct an `std::slice, (start = 2, stride = 3, end = 8)`, that represents the neighborhood offsets $(1,-1)$, $(1,0)$, $(1,1)$ (see Figure 6.6). If we pass this slice as an extra argument to the `itk::NeighborhoodInnerProduct` function, then the inner product is taken only along that slice. This "sliced" inner product with a 1D `itk::DerivativeOperator` gives the desired derivative.

The previous separable Gaussian filtering example can be rewritten using slices and slice-based inner products. In general, slice-based processing is most useful when doing many different calculations on the same neighborhood, where defining multiple iterators as in Section 6.4.1 becomes impractical or inefficient. Good examples of slice-based neighborhood processing can be found in any of the ND anisotropic diffusion function objects, such as `itk::CurvatureNDAnisotropicDiffusionFunction`.

The first difference between this example and the previous example is that the Gaussian operator is only initialized once. Its direction is not important because it is only a 1D array of coefficients.

```
itk::GaussianOperator< PixelType, 2 > gaussianOperator;
gaussianOperator.SetDirection(0);
gaussianOperator.SetVariance( ::atof(argv[3]) * ::atof(argv[3]) );
gaussianOperator.CreateDirectional();
```

Next we need to define a radius for the iterator. The radius in all directions matches that of the single extent of the Gaussian operator, defining a square neighborhood.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill( gaussianOperator.GetRadius()[0] );
```

The inner product and face calculator are defined for the main processing loop as before, but now the iterator is reinitialized each iteration with the square `radius` instead of the radius of the operator. The inner product is taken using a slice along the axial direction corresponding to the current itera-

tion. Note the use of `GetSlice()` to return the proper slice from the iterator itself. `GetSlice()` can only be used to return the slice along the complete extent of the axial direction of a neighborhood.

```
ImageType::Pointer input = reader->GetOutput();
faceList = faceCalculator(input, output->GetRequestedRegion(), radius);

for (unsigned int i = 0; i < ImageType::ImageDimension; ++i)
  {
  for ( fit=faceList.begin(); fit != faceList.end(); ++fit )
    {
    it = NeighborhoodIteratorType( radius, input, *fit );
    out = IteratorType( output, *fit );
    for (it.GoToBegin(), out.GoToBegin(); ! it.IsAtEnd(); ++it, ++out)
      {
      out.Set( innerProduct(it.GetSlice(i), it, gaussianOperator) );
      }
    }

  // Swap the input and output buffers
  if (i != ImageType::ImageDimension - 1)
    {
    ImageType::Pointer tmp = input;
    input = output;
    output = tmp;
    }
  }
```

This technique produces exactly the same results as the previous example. A little experimentation, however, will reveal that it is less efficient since the neighborhood iterator is keeping track of extra, unused pixel locations for each iteration, while the previous example only references those pixels that it needs. In cases, however, where an algorithm takes multiple derivatives or convolution products over the same neighborhood, slice-based processing can increase efficiency and simplify the implementation.

Random access iteration

The source code for this section can be found in the file `NeighborhoodIterators6.cxx`.

Some image processing routines do not need to visit every pixel in an image. Flood-fill and connected-component algorithms, for example, only visit pixels that are locally connected to one another. Algorithms such as these can be efficiently written using the random access capabilities of the neighborhood iterator.

The following example finds local minima. Given a seed point, we can search the neighborhood of that point and pick the smallest value m. While m is not at the center of our current neighborhood, we move in the direction of m and repeat the analysis. Eventually we discover a local minimum and stop. This algorithm is made trivially simple in ND using an ITK neighborhood iterator.

To illustrate the process, we create an image that descends everywhere to a single minimum: a

positive distance transform to a point. The details of creating the distance transform are not relevant to the discussion of neighborhood iterators, but can be found in the source code of this example. Some noise has been added to the distance transform image for additional interest.

The variable `input` is the pointer to the distance transform image. The local minimum algorithm is initialized with a seed point read from the command line.

```
ImageType::IndexType index;
index[0] = ::atoi(argv[2]);
index[1] = ::atoi(argv[3]);
```

Next we create the neighborhood iterator and position it at the seed point.

```
NeighborhoodIteratorType::RadiusType radius;
radius.Fill(1);
NeighborhoodIteratorType it(radius, input, input->GetRequestedRegion());

it.SetLocation(index);
```

Searching for the local minimum involves finding the minimum in the current neighborhood, then shifting the neighborhood in the direction of that minimum. The `for` loop below records the `itk::Offset` of the minimum neighborhood pixel. The neighborhood iterator is then moved using that offset. When a local minimum is detected, `flag` will remain false and the `while` loop will exit. Note that this code is valid for an image of any dimensionality.

```
bool flag = true;
while ( flag == true )
  {
  NeighborhoodIteratorType::OffsetType nextMove;
  nextMove.Fill(0);

  flag = false;

  PixelType min = it.GetCenterPixel();
  for (unsigned i = 0; i < it.Size(); i++)
    {
    if ( it.GetPixel(i) < min )
      {
      min = it.GetPixel(i);
      nextMove = it.GetOffset(i);
      flag = true;
      }
    }
  it.SetCenterPixel( 255.0 );
  it += nextMove;
  }
```

Figure 6.9 shows the results of the algorithm for several seed points. The white line is the path of the iterator from the seed point to the minimum in the center of the image. The effect of the additive noise is visible as the small perturbations in the paths.

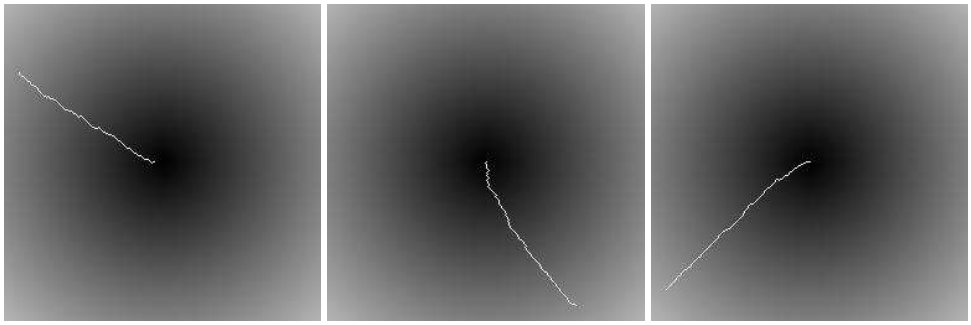

Figure 6.9: Paths traversed by the neighborhood iterator from different seed points to the local minimum. The true minimum is at the center of the image. The path of the iterator is shown in white. The effect of noise in the image is seen as small perturbations in each path.

6.4.2 ShapedNeighborhoodIterator

This section describes a variation on the neighborhood iterator called a *shaped* neighborhood iterator. A shaped neighborhood is defined like a bit mask, or *stencil*, with different offsets in the rectilinear neighborhood of the normal neighborhood iterator turned off or on to create a pattern. Inactive positions (those not in the stencil) are not updated during iteration and their values cannot be read or written. The shaped iterator is implemented in the class `itk::ShapedNeighborhoodIterator`, which is a subclass of `itk::NeighborhoodIterator`. A const version, `itk::ConstShapedNeighborhoodIterator`, is also available.

Like a regular neighborhood iterator, a shaped neighborhood iterator must be initialized with an ND radius object, but the radius of the neighborhood of a shaped iterator only defines the set of *possible* neighbors. Any number of possible neighbors can then be activated or deactivated. The shaped neighborhood iterator defines an API for activating neighbors. When a neighbor location, defined relative to the center of the neighborhood, is activated, it is placed on the *active list* and is then part of the stencil. An iterator can be "reshaped" at any time by adding or removing offsets from the active list.

- **`void ActivateOffset(OffsetType &o)`** Include the offset o in the stencil of active neighborhood positions. Offsets are relative to the neighborhood center.
- **`void DeactivateOffset(OffsetType &o)`** Remove the offset o from the stencil of active neighborhood positions. Offsets are relative to the neighborhood center.
- **`void ClearActiveList()`** Deactivate all positions in the iterator stencil by clearing the active list.
- **`unsigned int GetActiveIndexListSize()`** Return the number of pixel locations that are currently active in the shaped iterator stencil.

Because the neighborhood is less rigidly defined in the shaped iterator, the set of pixel access methods is restricted. Only the `GetPixel()` and `SetPixel()` methods are available, and calling these methods on an inactive neighborhood offset will return undefined results.

For the common case of traversing all pixel offsets in a neighborhood, the shaped iterator class provides an iterator through the active offsets in its stencil. This *stencil iterator* can be incremented or decremented and defines `Get()` and `Set()` for reading and writing the values in the neighborhood.

- **`ShapedNeighborhoodIterator::Iterator Begin()`** Return a const or non-const iterator through the shaped iterator stencil that points to the first valid location in the stencil.
- **`ShapedNeighborhoodIterator::Iterator End()`** Return a const or non-const iterator through the shaped iterator stencil that points *one position past* the last valid location in the stencil.

The functionality and interface of the shaped neighborhood iterator is best described by example. We will use the ShapedNeighborhoodIterator to implement some binary image morphology algorithms (see [4], [2], et al.). The examples that follow implement erosion and dilation.

Shaped neighborhoods: morphological operations

The source code for this section can be found in the file `ShapedNeighborhoodIterators1.cxx`.

This example uses `itk::ShapedNeighborhoodIterator` to implement a binary erosion algorithm. If we think of an image I as a set of pixel indices, then erosion of I by a smaller set E, called the *structuring element*, is the set of all indices at locations x in I such that when E is positioned at x, every element in E is also contained in I.

This type of algorithm is easy to implement with shaped neighborhood iterators because we can use the iterator itself as the structuring element E and move it sequentially through all positions x. The result at x is obtained by checking values in a simple iteration loop through the neighborhood stencil.

We need two iterators, a shaped iterator for the input image and a regular image iterator for writing results to the output image.

```
#include "itkConstShapedNeighborhoodIterator.h"
#include "itkImageRegionIterator.h"
```

Since we are working with binary images in this example, an `unsigned char` pixel type will do. The image and iterator types are defined using the pixel type.

```
typedef unsigned char              PixelType;
typedef itk::Image< PixelType, 2 >  ImageType;

typedef itk::ConstShapedNeighborhoodIterator<
                                      ImageType
                                        > ShapedNeighborhoodIteratorType;

typedef itk::ImageRegionIterator< ImageType> IteratorType;
```

Refer to the examples in Section 6.4.1 or the source code of this example for a description of how to read the input image and allocate a matching output image.

The size of the structuring element is read from the command line and used to define a radius for the shaped neighborhood iterator. Using the method developed in section 6.4.1 to minimize bounds checking, the iterator itself is not initialized until entering the main processing loop.

```
unsigned int element_radius = ::atoi( argv[3] );
ShapedNeighborhoodIteratorType::RadiusType radius;
radius.Fill(element_radius);
```

The face calculator object introduced in Section 6.4.1 is created and used as before.

```
typedef itk::NeighborhoodAlgorithm::ImageBoundaryFacesCalculator<
                                          ImageType > FaceCalculatorType;

FaceCalculatorType faceCalculator;
FaceCalculatorType::FaceListType faceList;
FaceCalculatorType::FaceListType::iterator fit;

faceList = faceCalculator( reader->GetOutput(),
                           output->GetRequestedRegion(),
                           radius );
```

Now we initialize some variables and constants.

```
IteratorType out;

const PixelType background_value = 0;
const PixelType foreground_value = 255;
const float rad = static_cast<float>(element_radius);
```

The outer loop of the algorithm is structured as in previous neighborhood iterator examples. Each region in the face list is processed in turn. As each new region is processed, the input and output iterators are initialized on that region.

The shaped iterator that ranges over the input is our structuring element and its active stencil must be created accordingly. For this example, the structuring element is shaped like a circle of radius `element_radius`. Each of the appropriate neighborhood offsets is activated in the double `for` loop.

```
for ( fit=faceList.begin(); fit != faceList.end(); ++fit)
  {
  ShapedNeighborhoodIteratorType it( radius, reader->GetOutput(), *fit );
  out = IteratorType( output, *fit );

  // Creates a circular structuring element by activating all the pixels less
  // than radius distance from the center of the neighborhood.

  for (float y = -rad; y <= rad; y++)
    {
    for (float x = -rad; x <= rad; x++)
      {
      ShapedNeighborhoodIteratorType::OffsetType off;

      float dis = std::sqrt( x*x + y*y );
      if (dis <= rad)
        {
        off[0] = static_cast<int>(x);
        off[1] = static_cast<int>(y);
        it.ActivateOffset(off);
        }
      }
    }
```

The inner loop, which implements the erosion algorithm, is fairly simple. The `for` loop steps the input and output iterators through their respective images. At each step, the active stencil of the shaped iterator is traversed to determine whether all pixels underneath the stencil contain the foreground value, i.e. are contained within the set I. Note the use of the stencil iterator, `ci`, in performing this check.

```
  // Implements erosion
  for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
    {
    ShapedNeighborhoodIteratorType::ConstIterator ci;

    bool flag = true;
    for (ci = it.Begin(); ci != it.End(); ci++)
      {
      if (ci.Get() == background_value)
        {
        flag = false;
        break;
        }
      }
    if (flag == true)
      {
      out.Set(foreground_value);
      }
    else
      {
      out.Set(background_value);
      }
    }
  }
```

Figure 6.10: The effects of morphological operations on a binary image using a circular structuring element of size 4. From left to right are the original image, erosion, dilation, opening, and closing. The opening operation is erosion of the image followed by dilation. Closing is dilation of the image followed by erosion.

The source code for this section can be found in the file `ShapedNeighborhoodIterators2.cxx`.

The logic of the inner loop can be rewritten to perform dilation. Dilation of the set I by E is the set of all x such that E positioned at x contains at least one element in I.

```
// Implements dilation
for (it.GoToBegin(), out.GoToBegin(); !it.IsAtEnd(); ++it, ++out)
  {
  ShapedNeighborhoodIteratorType::ConstIterator ci;

  bool flag = false;
  for (ci = it.Begin(); ci != it.End(); ci++)
    {
    if (ci.Get() != background_value)
      {
      flag = true;
      break;
      }
    }
  if (flag == true)
    {
    out.Set(foreground_value);
    }
  else
    {
    out.Set(background_value);
    }
  }
}
```

The output image is written and visualized directly as a binary image of `unsigned chars`. Figure 6.10 illustrates some results of erosion and dilation on the image `Examples/Data/BinaryImage.png`. Applying erosion and dilation in sequence effects the morphological operations of opening and closing.

CHAPTER

SEVEN

IMAGE ADAPTORS

The purpose of an *image adaptor* is to make one image appear like another image, possibly of a different pixel type. A typical example is to take an image of pixel type `unsigned char` and present it as an image of pixel type `float`. The motivation for using image adaptors in this case is to avoid the extra memory resources required by using a casting filter. When we use the `itk::CastImageFilter` for the conversion, the filter creates a memory buffer large enough to store the `float` image. The `float` image requires four times the memory of the original image and contains no useful additional information. Image adaptors, on the other hand, do not require the extra memory as pixels are converted only when they are read using image iterators (see Chapter 6).

Image adaptors are particularly useful when there is infrequent pixel access, since the actual conversion occurs on the fly during the access operation. In such cases the use of image adaptors may reduce overall computation time as well as reduce memory usage. The use of image adaptors, however, can be disadvantageous in some situations. For example, when the downstream filter is executed multiple times, a CastImageFilter will cache its output after the first execution and will not re-execute when the filter downstream is updated. Conversely, an image adaptor will compute the cast every time.

Another application for image adaptors is to perform lightweight pixel-wise operations replacing the need for a filter. In the toolkit, adaptors are defined for many single valued and single parameter functions such as trigonometric, exponential and logarithmic functions. For example,

- `itk::ExpImageAdaptor`
- `itk::SinImageAdaptor`
- `itk::CosImageAdaptor`

The following examples illustrate common applications of image adaptors.

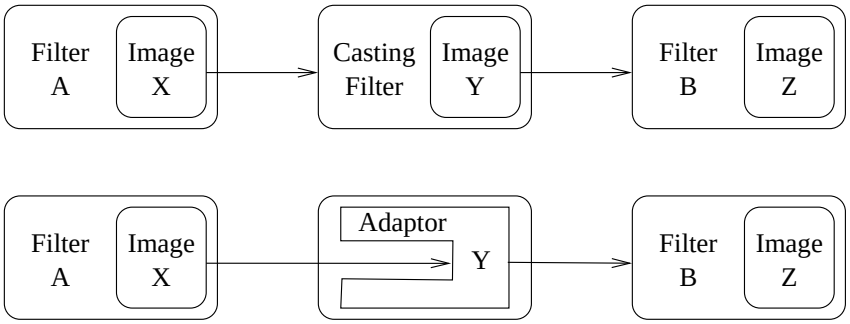

Figure 7.1: The difference between using a CastImageFilter and an ImageAdaptor. ImageAdaptors convert pixel values when they are accessed by iterators. Thus, they do not produces an intermediate image. In the example illustrated by this figure, the *Image Y* is not created by the ImageAdaptor; instead, the image is simulated on the fly each time an iterator from the filter downstream attempts to access the image data.

7.1 Image Casting

The source code for this section can be found in the file `ImageAdaptor1.cxx`.

This example illustrates how the `itk::ImageAdaptor` can be used to cast an image from one pixel type to another. In particular, we will *adapt* an `unsigned char` image to make it appear as an image of pixel type `float`.

We begin by including the relevant headers.

```
#include "itkImageAdaptor.h"
```

First, we need to define a *pixel accessor* class that does the actual conversion. Note that in general, the only valid operations for pixel accessors are those that only require the value of the input pixel. As such, neighborhood type operations are not possible. A pixel accessor must provide methods `Set()` and `Get()`, and define the types of `InternalPixelType` and `ExternalPixelType`. The `InternalPixelType` corresponds to the pixel type of the image to be adapted (`unsigned char` in this example). The `ExternalPixelType` corresponds to the pixel type we wish to emulate with the ImageAdaptor (`float` in this case).

```
class CastPixelAccessor
{
public:
  typedef unsigned char InternalType;
  typedef float         ExternalType;

  static void Set(InternalType & output, const ExternalType & input)
    {
    output = static_cast<InternalType>( input );
    }

  static ExternalType Get( const InternalType & input )
    {
    return static_cast<ExternalType>( input );
    }
};
```

The CastPixelAccessor class simply applies a `static_cast` to the pixel values. We now use this pixel accessor to define the image adaptor type and create an instance using the standard `New()` method.

```
  typedef unsigned char  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >   ImageType;

  typedef itk::ImageAdaptor< ImageType, CastPixelAccessor > ImageAdaptorType;
  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

We also create an image reader templated over the input image type and read the input image from file.

```
  typedef itk::ImageFileReader< ImageType >   ReaderType;
  ReaderType::Pointer reader = ReaderType::New();
```

The output of the reader is then connected as the input to the image adaptor.

```
  adaptor->SetImage( reader->GetOutput() );
```

In the following code, we visit the image using an iterator instantiated using the adapted image type and compute the sum of the pixel values.

```
  typedef itk::ImageRegionIteratorWithIndex< ImageAdaptorType >  IteratorType;
  IteratorType  it( adaptor, adaptor->GetBufferedRegion() );

  double sum = 0.0;
  it.GoToBegin();
  while( !it.IsAtEnd() )
    {
    float value = it.Get();
    sum += value;
    ++it;
    }
```

Although in this example, we are just performing a simple summation, the key concept is that access to pixels is performed as if the pixel is of type `float`. Additionally, it should be noted that the adaptor is used as if it was an actual image and not as a filter. ImageAdaptors conform to the same API as the `itk::Image` class.

7.2 Adapting RGB Images

The source code for this section can be found in the file `ImageAdaptor2.cxx`.

This example illustrates how to use the `itk::ImageAdaptor` to access the individual components of an RGB image. In this case, we create an ImageAdaptor that will accept a RGB image as input and presents it as a scalar image. The pixel data will be taken directly from the red channel of the original image.

As with the previous example, the bulk of the effort in creating the image adaptor is associated with the definition of the pixel accessor class. In this case, the accessor converts a RGB vector to a scalar containing the red channel component. Note that in the following, we do not need to define the `Set()` method since we only expect the adaptor to be used for reading data from the image.

```
class RedChannelPixelAccessor
{
public:
  typedef itk::RGBPixel<float>   InternalType;
  typedef              float     ExternalType;

  static ExternalType Get( const InternalType & input )
    {
    return static_cast<ExternalType>( input.GetRed() );
    }
};
```

The `Get()` method simply calls the `GetRed()` method defined in the `itk::RGBPixel` class.

Now we use the internal pixel type of the pixel accessor to define the input image type, and then proceed to instantiate the ImageAdaptor type.

```
  typedef RedChannelPixelAccessor::InternalType  InputPixelType;
  const   unsigned int   Dimension = 2;
  typedef itk::Image< InputPixelType, Dimension >   ImageType;

  typedef itk::ImageAdaptor<  ImageType,
                              RedChannelPixelAccessor > ImageAdaptorType;

  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

We create an image reader and connect the output to the adaptor as before.

```
typedef itk::ImageFileReader< ImageType >   ReaderType;
ReaderType::Pointer reader = ReaderType::New();
```

```
adaptor->SetImage( reader->GetOutput() );
```

We create an `itk::RescaleIntensityImageFilter` and an `itk::ImageFileWriter` to rescale the dynamic range of the pixel values and send the extracted channel to an image file. Note that the image type used for the rescaling filter is the `ImageAdaptorType` itself. That is, the adaptor type is used in the same context as an image type.

```
typedef itk::Image< unsigned char, Dimension >   OutputImageType;
typedef itk::RescaleIntensityImageFilter< ImageAdaptorType,
                                          OutputImageType
                                             >   RescalerType;

RescalerType::Pointer rescaler = RescalerType::New();
typedef itk::ImageFileWriter< OutputImageType >   WriterType;
WriterType::Pointer writer = WriterType::New();
```

Now we connect the adaptor as the input to the rescaler and set the parameters for the intensity rescaling.

```
rescaler->SetOutputMinimum(  0  );
rescaler->SetOutputMaximum( 255 );

rescaler->SetInput( adaptor );
writer->SetInput( rescaler->GetOutput() );
```

Finally, we invoke the `Update()` method on the writer and take precautions to catch any exception that may be thrown during the execution of the pipeline.

```
try
  {
  writer->Update();
  }
catch( itk::ExceptionObject & excp )
  {
  std::cerr << "Exception caught " << excp << std::endl;
  return 1;
  }
```

ImageAdaptors for the green and blue channels can easily be implemented by modifying the pixel accessor of the red channel and then using the new pixel accessor for instantiating the type of an image adaptor. The following define a green channel pixel accessor.

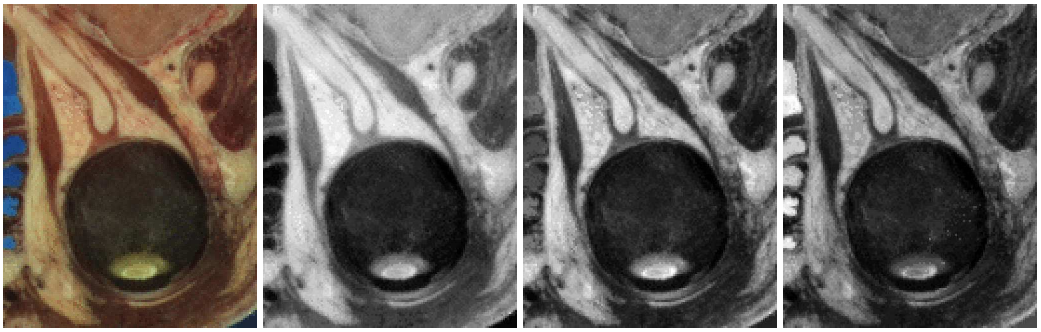

Figure 7.2: Using ImageAdaptor to extract the components of an RGB image. The image on the left is a subregion of the Visible Woman cryogenic data set. The red, green and blue components are shown from left to right as scalar images extracted with an ImageAdaptor.

```
class GreenChannelPixelAccessor
{
public:
  typedef itk::RGBPixel<float>   InternalType;
  typedef              float     ExternalType;

  static ExternalType Get( const InternalType & input )
    {
    return static_cast<ExternalType>( input.GetGreen() );
    }
  };
```

A blue channel pixel accessor is similarly defined.

```
class BlueChannelPixelAccessor
  {
public:
  typedef itk::RGBPixel<float>   InternalType;
  typedef              float     ExternalType;

  static ExternalType Get( const InternalType & input )
    {
    return static_cast<ExternalType>( input.GetBlue() );
    }
  };
```

Figure 7.2 shows the result of extracting the red, green and blue components from a region of the Visible Woman cryogenic data set.

7.3 Adapting Vector Images

The source code for this section can be found in the file
`ImageAdaptor3.cxx`.

This example illustrates the use of `itk::ImageAdaptor` to obtain access to the components of a vector image. Specifically, it shows how to manage pixel accessors containing internal parameters. In this example we create an image of vectors by using a gradient filter. Then, we use an image adaptor to extract one of the components of the vector image. The vector type used by the gradient filter is the `itk::CovariantVector` class.

We start by including the relevant headers.

```
#include "itkGradientRecursiveGaussianImageFilter.h"
```

A pixel accessors class may have internal parameters that affect the operations performed on input pixel data. Image adaptors support parameters in their internal pixel accessor by using the assignment operator. Any pixel accessor which has internal parameters must therefore implement the assignment operator. The following defines a pixel accessor for extracting components from a vector pixel. The `m_Index` member variable is used to select the vector component to be returned.

```
class VectorPixelAccessor
{
public:
  typedef itk::CovariantVector<float,2>   InternalType;
  typedef                     float       ExternalType;

  VectorPixelAccessor() : m_Index(0) {}

  VectorPixelAccessor & operator=( const VectorPixelAccessor & vpa )
    {
      m_Index = vpa.m_Index;
      return *this;
    }
  ExternalType Get( const InternalType & input ) const
    {
    return static_cast<ExternalType>( input[ m_Index ] );
    }
  void SetIndex( unsigned int index )
    {
    m_Index = index;
    }

private:
  unsigned int m_Index;
};
```

The `Get()` method simply returns the i-th component of the vector as indicated by the index. The assignment operator transfers the value of the index member variable from one instance of the pixel accessor to another.

In order to test the pixel accessor, we generate an image of vectors using the itk::GradientRecursiveGaussianImageFilter. This filter produces an output image of itk::CovariantVector pixel type. Covariant vectors are the natural representation for gradients since they are the equivalent of normals to iso-values manifolds.

```
typedef unsigned char  InputPixelType;
const   unsigned int   Dimension = 2;
typedef itk::Image< InputPixelType,  Dimension  >   InputImageType;
typedef itk::CovariantVector< float, Dimension  >   VectorPixelType;
typedef itk::Image< VectorPixelType, Dimension  >   VectorImageType;
typedef itk::GradientRecursiveGaussianImageFilter< InputImageType,
                                    VectorImageType> GradientFilterType;

GradientFilterType::Pointer gradient = GradientFilterType::New();
```

We instantiate the ImageAdaptor using the vector image type as the first template parameter and the pixel accessor as the second template parameter.

```
typedef itk::ImageAdaptor<  VectorImageType,
                            itk::VectorPixelAccessor > ImageAdaptorType;

ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

The index of the component to be extracted is specified from the command line. In the following, we create the accessor, set the index and connect the accessor to the image adaptor using the `SetPixelAccessor()` method.

```
itk::VectorPixelAccessor  accessor;
accessor.SetIndex( atoi( argv[3] ) );
adaptor->SetPixelAccessor( accessor );
```

We create a reader to load the image specified from the command line and pass its output as the input to the gradient filter.

```
typedef itk::ImageFileReader< InputImageType >   ReaderType;
ReaderType::Pointer reader = ReaderType::New();
gradient->SetInput( reader->GetOutput() );

reader->SetFileName( argv[1] );
gradient->Update();
```

We now connect the output of the gradient filter as input to the image adaptor. The adaptor emulates a scalar image whose pixel values are taken from the selected component of the vector image.

```
adaptor->SetImage( gradient->GetOutput() );
```

As in the previous example, we rescale the scalar image before writing the image out to file. Figure 7.3 shows the result of applying the example code for extracting both components of a two dimensional gradient.

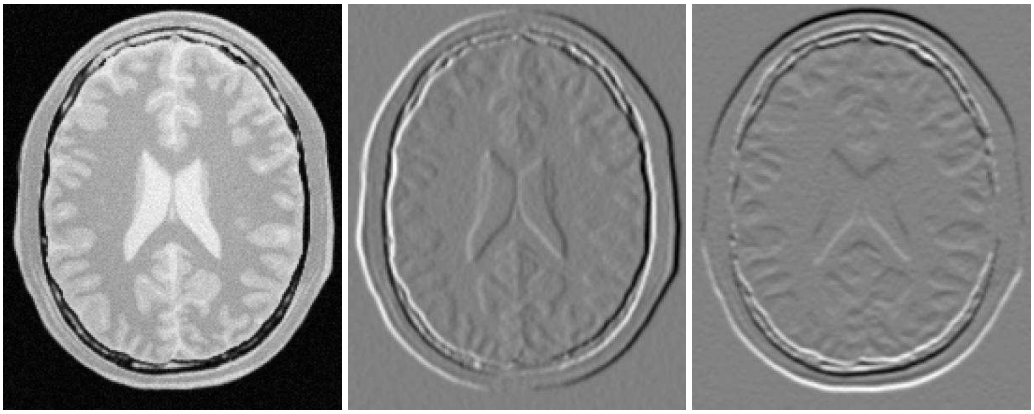

Figure 7.3: Using ImageAdaptor to access components of a vector image. The input image on the left was passed through a gradient image filter and the two components of the resulting vector image were extracted using an image adaptor.

7.4 Adaptors for Simple Computation

The source code for this section can be found in the file `ImageAdaptor4.cxx`.

Image adaptors can also be used to perform simple pixel-wise computations on image data. The following example illustrates how to use the `itk::ImageAdaptor` for image thresholding.

A pixel accessor for image thresholding requires that the accessor maintain the threshold value. Therefore, it must also implement the assignment operator to set this internal parameter.

```
class ThresholdingPixelAccessor
{
public:
  typedef unsigned char      InternalType;
  typedef unsigned char      ExternalType;

  ThresholdingPixelAccessor() : m_Threshold(0) {};

  ExternalType Get( const InternalType & input ) const
    {
    return (input > m_Threshold) ? 1 : 0;
    }
  void SetThreshold( const InternalType threshold )
    {
    m_Threshold = threshold;
    }

  ThresholdingPixelAccessor &
    operator=( const ThresholdingPixelAccessor & vpa )
    {
    m_Threshold = vpa.m_Threshold;
    return *this;
    }

private:
  InternalType m_Threshold;
};
}
```

The `Get()` method returns one if the input pixel is above the threshold and zero otherwise. The assignment operator transfers the value of the threshold member variable from one instance of the pixel accessor to another.

To create an image adaptor, we first instantiate an image type whose pixel type is the same as the internal pixel type of the pixel accessor.

```
  typedef itk::ThresholdingPixelAccessor::InternalType     PixelType;
  const   unsigned int   Dimension = 2;
  typedef itk::Image< PixelType,  Dimension >   ImageType;
```

We instantiate the ImageAdaptor using the image type as the first template parameter and the pixel accessor as the second template parameter.

```
  typedef itk::ImageAdaptor< ImageType,
                             itk::ThresholdingPixelAccessor > ImageAdaptorType;

  ImageAdaptorType::Pointer adaptor = ImageAdaptorType::New();
```

The threshold value is set from the command line. A threshold pixel accessor is created and connected to the image adaptor in the same manner as in the previous example.

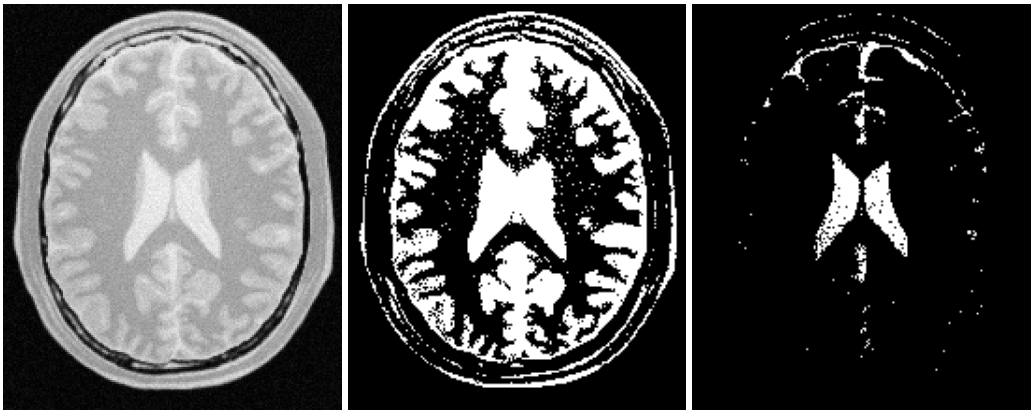

Figure 7.4: Using ImageAdaptor to perform a simple image computation. An ImageAdaptor is used to perform binary thresholding on the input image on the left. The center image was created using a threshold of 180, while the image on the right corresponds to a threshold of 220.

```
itk::ThresholdingPixelAccessor  accessor;
accessor.SetThreshold( atoi( argv[3] ) );
adaptor->SetPixelAccessor( accessor );
```

We create a reader to load the input image and connect the output of the reader as the input to the adaptor.

```
typedef itk::ImageFileReader< ImageType >   ReaderType;
ReaderType::Pointer reader = ReaderType::New();
reader->SetFileName( argv[1] );
reader->Update();

adaptor->SetImage( reader->GetOutput() );
```

As before, we rescale the emulated scalar image before writing it out to file. Figure 7.4 illustrates the result of applying the thresholding adaptor to a typical gray scale image using two different threshold values. Note that the same effect could have been achieved by using the `itk::BinaryThresholdImageFilter` but at the price of holding an extra copy of the image in memory.

7.5 Adaptors and Writers

Image adaptors will not behave correctly when connected directly to a writer. The reason is that writers tend to get direct access to the image buffer from their input, since image adaptors do not have a real buffer their behavior in this circumstances is incorrect. You should avoid instantiating the `ImageFileWriter` or the `ImageSeriesWriter` over an image adaptor type.

CHAPTER

EIGHT

HOW TO WRITE A FILTER

This purpose of this chapter is help developers create their own filter (process object). This chapter is divided into four major parts. An initial definition of terms is followed by an overview of the filter creation process. Next, data streaming is discussed. The way data is streamed in ITK must be understood in order to write correct filters. Finally, a section on multithreading describes what you must do in order to take advantage of shared memory parallel processing.

8.1 Terminology

The following is some basic terminology for the discussion that follows. Chapter 3 provides additional background information.

- The **data processing pipeline** is a directed graph of **process** and **data objects**. The pipeline inputs, operators on, and outputs data.
- A **filter**, or **process object**, has one or more inputs, and one or more outputs.
- A **source**, or source process object, initiates the data processing pipeline, and has one or more outputs.
- A **mapper**, or mapper process object, terminates the data processing pipeline. The mapper has one or more outputs, and may write data to disk, interface with a display system, or interface to any other system.
- A **data object** represents and provides access to data. In ITK, the data object (ITK class `itk::DataObject`) is typically of type `itk::Image` or `itk::Mesh`.
- A **region** (ITK class `itk::Region`) represents a piece, or subset of the entire data set.
- An **image region** (ITK class `itk::ImageRegion`) represents a structured portion of data. ImageRegion is implemented using the `itk::Index` and `itk::Size` classes

- A **mesh region** (ITK class `itk::MeshRegion`) represents an unstructured portion of data.
- The **LargestPossibleRegion** is the theoretical single, largest piece (region) that could represent the entire dataset. The LargestPossibleRegion is used in the system as the measure of the largest possible data size.
- The **BufferedRegion** is a contiguous block of memory that is less than or equal to in size to the LargestPossibleRegion. The buffered region is what has actually been allocated by a filter to hold its output.
- The **RequestedRegion** is the piece of the dataset that a filter is required to produce. The RequestedRegion is less than or equal in size to the BufferedRegion. The RequestedRegion may differ in size from the BufferedRegion due to performance reasons. The RequestedRegion may be set by a user, or by an application that needs just a portion of the data.
- The **modified time** (represented by ITK class `itk::TimeStamp`) is a monotonically increasing integer value that characterizes a point in time when an object was last modified.
- **Downstream** is the direction of dataflow, from sources to mappers.
- **Upstream** is the opposite of downstream, from mappers to sources.
- The **pipeline modified time** for a particular data object is the maximum modified time of all upstream data objects and process objects.
- The term **information** refers to metadata that characterizes data. For example, index and dimensions are information characterizing an image region.

8.2 Overview of Filter Creation

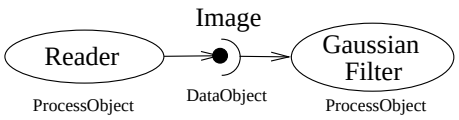

Figure 8.1: Relationship between DataObject and ProcessObject.

Filters are defined with respect to the type of data they input (if any), and the type of data they output (if any). The key to writing a ITK filter is to identify the number and types of input and output. Having done so, there are often superclasses that simplify this task via class derivation. For example, most filters in ITK take a single image as input, and produce a single image on output. The superclass `itk::ImageToImageFilter` is a convenience class that provide most of the functionality needed for such a filter.

Some common base classes for new filters include:

- `ImageToImageFilter`: the most common filter base for segmentation algorithms. Takes an image and produces a new image, by default of the same dimensions. Override `GenerateOutputInformation` to produce a different size.

- `UnaryFunctorImageFilter`: used when defining a filter that applies a function to an image.
- `BinaryFunctorImageFilter`: used when defining a filter that applies an operation to two images.
- `ImageFunction`: a functor that can be applied to an image, evaluating $f(x)$ at each point in the image.
- `MeshToMeshFilter`: a filter that transforms meshes, such as tessellation, polygon reduction, and so on.
- `LightObject`: abstract base for filters that don't fit well anywhere else in the class hierarchy. Also useful for "calculator" filters; ie. a sink filter that takes an input and calculates a result which is retrieved using a `Get()` method.

Once the appropriate superclass is identified, the filter writer implements the class defining the methods required by most all ITK objects: `New()`, `PrintSelf()`, and protected constructor, copy constructor, delete, and operator=, and so on. Also, don't forget standard typedefs like `Self`, `Superclass`, `Pointer`, and `ConstPointer`. Then the filter writer can focus on the most important parts of the implementation: defining the API, data members, and other implementation details of the algorithm. In particular, the filter writer will have to implement either a `GenerateData()` (non-threaded) or `ThreadedGenerateData()` method. (See Section 3.2.7 for an overview of multithreading in ITK.)

An important note: the GenerateData() method is required to allocate memory for the output. The ThreadedGenerateData() method is not. In default implementation (see `itk::ImageSource`, a superclass of `itk::ImageToImageFilter`) `GenerateData()` allocates memory and then invokes `ThreadedGenerateData()`.

One of the most important decisions that the developer must make is whether the filter can stream data; that is, process just a portion of the input to produce a portion of the output. Often superclass behavior works well: if the filter processes the input using single pixel access, then the default behavior is adequate. If not, then the user may have to a) find a more specialized superclass to derive from, or b) override one or more methods that control how the filter operates during pipeline execution. The next section describes these methods.

8.3 Streaming Large Data

The data associated with multi-dimensional images is large and becoming larger. This trend is due to advances in scanning resolution, as well as increases in computing capability. Any practical segmentation and registration software system must address this fact in order to be useful in application. ITK addresses this problem via its data streaming facility.

In ITK, streaming is the process of dividing data into pieces, or regions, and then processing this data through the data pipeline. Recall that the pipeline consists of process objects that generate data

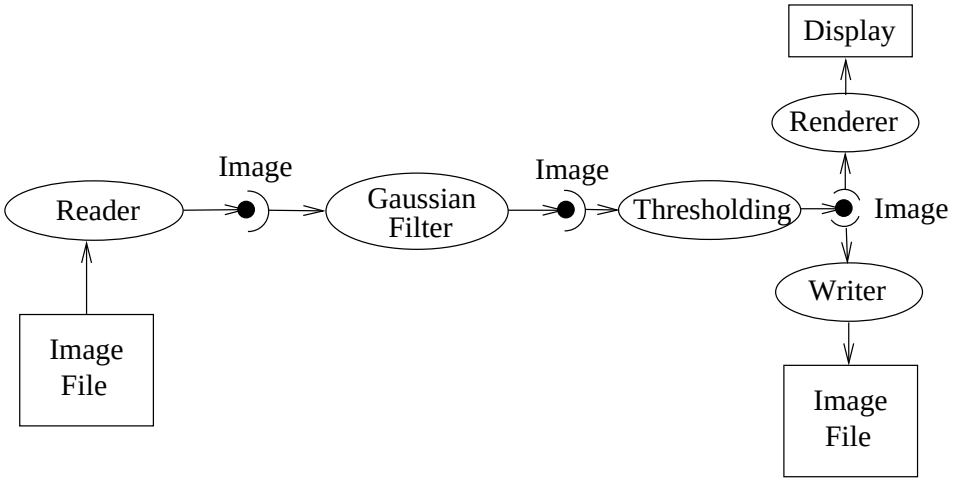

Figure 8.2: The Data Pipeline

objects, connected into a pipeline topology. The input to a process object is a data object (unless the process initiates the pipeline and then it is a source process object). These data objects in turn are consumed by other process objects, and so on, until a directed graph of data flow is constructed. Eventually the pipeline is terminated by one or more mappers, that may write data to storage, or interface with a graphics or other system. This is illustrated in figures 8.1 and 8.2.

A significant benefit of this architecture is that the relatively complex process of managing pipeline execution is designed into the system. This means that keeping the pipeline up to date, executing only those portions of the pipeline that have changed, multithreading execution, managing memory allocation, and streaming is all built into the architecture. However, these features do introduce complexity into the system, the bulk of which is seen by class developers. The purpose of this chapter is to describe the pipeline execution process in detail, with a focus on data streaming.

8.3.1 Overview of Pipeline Execution

The pipeline execution process performs several important functions.

1. It determines which filters, in a pipeline of filters, need to execute. This prevents redundant execution and minimizes overall execution time.

2. It initializes the (filter's) output data objects, preparing them for new data. In addition, it determines how much memory each filter must allocate for its output, and allocates it.

3. The execution process determines how much data a filter must process in order to produce an output of sufficient size for downstream filters; it also takes into account any limits on memory

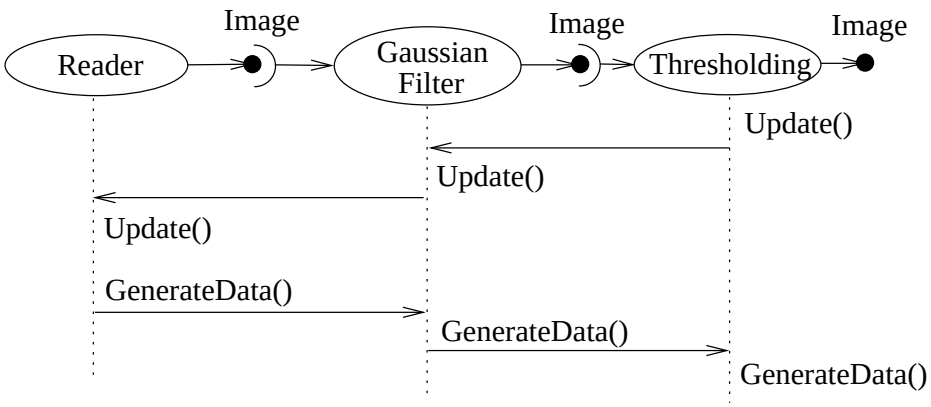

Figure 8.3: Sequence of the Data Pipeline updating mechanism

or special filter requirements. Other factors include the size of data processing kernels, that affect how much data input data (extra padding) is required.

4. It subdivides data into subpieces for multithreading. (Note that the division of data into subpieces is exactly same problem as dividing data into pieces for streaming; hence multithreading comes for free as part of the streaming architecture.)

5. It may free (or release) output data if filters no longer need it to compute, and the user requests that data is to be released. (Note: a filter's output data object may be considered a "cache". If the cache is allowed to remain (`ReleaseDataFlagOff()`) between pipeline execution, and the filter, or the input to the filter, never changes, then process objects downstream of the filter just reuse the filter's cache to re-execute.)

To perform these functions, the execution process negotiates with the filters that define the pipeline. Only each filter can know how much data is required on input to produce a particular output. For example, a shrink filter with a shrink factor of two requires an image twice as large (in terms of its x-y dimensions) on input to produce a particular size output. An image convolution filter would require extra input (boundary padding) depending on the size of the convolution kernel. Some filters require the entire input to produce an output (for example, a histogram), and have the option of requesting the entire input. (In this case streaming does not work unless the developer creates a filter that can request multiple pieces, caching state between each piece to assemble the final output.)

Ultimately the negotiation process is controlled by the request for data of a particular size (i.e., region). It may be that the user asks to process a region of interest within a large image, or that memory limitations result in processing the data in several pieces. For example, an application may compute the memory required by a pipeline, and then use `itk::StreamingImageFilter` to break the data processing into several pieces. The data request is propagated through the pipeline in the upstream direction, and the negotiation process configures each filter to produce output data of a particular size.

The secret to creating a streaming filter is to understand how this negotiation process works, and how to override its default behavior by using the appropriate virtual functions defined in itk::ProcessObject. The next section describes the specifics of these methods, and when to override them. Examples are provided along the way to illustrate concepts.

8.3.2 Details of Pipeline Execution

Typically pipeline execution is initiated when a process object receives the ProcessObject::Update() method invocation. This method is simply delegated to the output of the filter, invoking the DataObject::Update() method. Note that this behavior is typical of the interaction between ProcessObject and DataObject: a method invoked on one is eventually delegated to the other. In this way the data request from the pipeline is propagated upstream, initiating data flow that returns downstream.

The DataObject::Update() method in turn invokes three other methods:

- DataObject::UpdateOutputInformation()
- DataObject::PropagateRequestedRegion()
- DataObject::UpdateOutputData()

UpdateOutputInformation()

The UpdateOutputInformation() method determines the pipeline modified time. It may set the RequestedRegion and the LargestPossibleRegion depending on how the filters are configured. (The RequestedRegion is set to process all the data, i.e., the LargestPossibleRegion, if it has not been set.) The UpdateOutputInformation() propagates upstream through the entire pipeline and terminates at the sources.

During UpdateOutputInformation(), filters have a chance to override the ProcessObject::GenerateOutputInformation() method (GenerateOutputInformation() is invoked by UpdateOutputInformation()). The default behavior is for the GenerateOutputInformation() to copy the metadata describing the input to the output (via DataObject::CopyInformation()). Remember, information is metadata describing the output, such as the origin, spacing, and LargestPossibleRegion (i.e., largest possible size) of an image.

A good example of this behavior is itk::ShrinkImageFilter. This filter takes an input image and shrinks it by some integral value. The result is that the spacing and LargestPossibleRegion of the output will be different to that of the input. Thus, GenerateOutputInformation() is overloaded.

PropagateRequestedRegion()

The `PropagateRequestedRegion()` call propagates upstream to satisfy a data request. In typical application this data request is usually the LargestPossibleRegion, but if streaming is necessary, or the user is interested in updating just a portion of the data, the RequestedRegion may be any valid region within the LargestPossibleRegion.

The function of `PropagateRequestedRegion()` is, given a request for data (the amount is specified by RequestedRegion), propagate upstream configuring the filter's input and output process object's to the correct size. Eventually, this means configuring the BufferedRegion, that is the amount of data actually allocated.

The reason for the buffered region is this: the output of a filter may be consumed by more than one downstream filter. If these consumers each request different amounts of input (say due to kernel requirements or other padding needs), then the upstream, generating filter produces the data to satisfy both consumers, that may mean it produces more data than one of the consumers needs.

The `ProcessObject::PropagateRequestedRegion()` method invokes three methods that the filter developer may choose to overload.

- `EnlargeOutputRequestedRegion(DataObject *output)` gives the (filter) subclass a chance to indicate that it will provide more data than required for the output. This can happen, for example, when a source can only produce the whole output (i.e., the LargestPossibleRegion).
- `GenerateOutputRequestedRegion(DataObject *output)` gives the subclass a chance to define how to set the requested regions for each of its outputs, given this output's requested region. The default implementation is to make all the output requested regions the same. A subclass may need to override this method if each output is a different resolution. This method is only overridden if a filter has multiple outputs.
- `GenerateInputRequestedRegion()` gives the subclass a chance to request a larger requested region on the inputs. This is necessary when, for example, a filter requires more data at the "internal" boundaries to produce the boundary values - due to kernel operations or other region boundary effects.

`itk::RGBGibbsPriorFilter` is an example of a filter that needs to invoke `EnlargeOutputRequestedRegion()`. The designer of this filter decided that the filter should operate on all the data. Note that a subtle interplay between this method and `GenerateInputRequestedRegion()` is occurring here. The default behavior of `GenerateInputRequestedRegion()` (at least for `itk::ImageToImageFilter`) is to set the input RequestedRegion to the output's ReqestedRegion. Hence, by overriding the method `EnlargeOutputRequestedRegion()` to set the output to the LargestPossibleRegion, effectively sets the input to this filter to the LargestPossibleRegion (and probably causing all upstream filters to process their LargestPossibleRegion as well. This means that the filter, and therefore the pipeline,

does not stream. This could be fixed by reimplementing the filter with the notion of streaming built in to the algorithm.)

itk::GradientMagnitudeImageFilter is an example of a filter that needs to invoke GenerateInputRequestedRegion(). It needs a larger input requested region because a kernel is required to compute the gradient at a pixel. Hence the input needs to be "padded out" so the filter has enough data to compute the gradient at each output pixel.

UpdateOutputData()

UpdateOutputData() is the third and final method as a result of the Update() method. The purpose of this method is to determine whether a particular filter needs to execute in order to bring its output up to date. (A filter executes when its GenerateData() method is invoked.) Filter execution occurs when a) the filter is modified as a result of modifying an instance variable; b) the input to the filter changes; c) the input data has been released; or d) an invalid RequestedRegion was set previously and the filter did not produce data. Filters execute in order in the downstream direction. Once a filter executes, all filters downstream of it must also execute.

DataObject::UpdateOutputData() is delegated to the DataObject's source (i.e., the ProcessObject that generated it) only if the DataObject needs to be updated. A comparison of modified time, pipeline time, release data flag, and valid requested region is made. If any one of these conditions indicate that the data needs regeneration, then the source's ProcessObject::UpdateOutputData() is invoked. These calls are made recursively up the pipeline until a source filter object is encountered, or the pipeline is determined to be up to date and valid. At this point, the recursion unrolls, and the execution of the filter proceeds. (This means that the output data is initialized, StartEvent is invoked, the filters GenerateData() is called, EndEvent is invoked, and input data to this filter may be released, if requested. In addition, this filter's InformationTime is updated to the current time.)

The developer will never override UpdateOutputData(). The developer need only write the GenerateData() method (non-threaded) or ThreadedGenerateData() method. A discussion of threading follows in the next section.

8.4 Threaded Filter Execution

Filters that can process data in pieces can typically multi-process using the data parallel, shared memory implementation built into the pipeline execution process. To create a multithreaded filter, simply define and implement a ThreadedGenerateData() method. For example, a itk::ImageToImageFilter would create the method:

```
void ThreadedGenerateData(const OutputImageRegionType&
                          outputRegionForThread, int threadId)
```

The key to threading is to generate output for the output region given (as the first parameter in the

argument list above). In ITK, this is simple to do because an output iterator can be created using the region provided. Hence the output can be iterated over, accessing the corresponding input pixels as necessary to compute the value of the output pixel.

Multi-threading requires caution when performing I/O (including using `cout` or `cerr`) or invoking events. A safe practice is to allow only thread id zero to perform I/O or generate events. (The thread id is passed as argument into `ThreadedGenerateData()`). If more than one thread tries to write to the same place at the same time, the program can behave badly, and possibly even deadlock or crash.

8.5 Filter Conventions

In order to fully participate in the ITK pipeline, filters are expected to follow certain conventions, and provide certain interfaces. This section describes the minimum requirements for a filter to integrate into the ITK framework.

The class declaration for a filter should include the macro `ITK_EXPORT`, so that on certain platforms an export declaration can be included.

A filter should define public types for the class itself (`Self`) and its `Superclass`, and `const` and non-`const` smart pointers, thus:

```
typedef ExampleImageFilter                Self;
typedef ImageToImageFilter<TImage,TImage> Superclass;
typedef SmartPointer<Self>                Pointer;
typedef SmartPointer<const Self>          ConstPointer;
```

The `Pointer` type is particularly useful, as it is a smart pointer that will be used by all client code to hold a reference-counted instantiation of the filter.

Once the above types have been defined, you can use the following convenience macros, which permit your filter to participate in the object factory mechanism, and to be created using the canonical `::New()`:

```
/** Method for creation through the object factory. */
itkNewMacro(Self);

/** Run-time type information (and related methods). */
itkTypeMacro(ExampleImageFilter, ImageToImageFilter);
```

The default constructor should be `protected`, and provide sensible defaults (usually zero) for all parameters. The copy constructor and assignment operator should be declared `private` and not implemented, to prevent instantiating the filter without the factory methods (above).

Finally, the template implementation code (in the `.hxx` file) should be included, bracketed by a test for manual instantiation, thus:

```
#ifndef ITK_MANUAL_INSTANTIATION
#include "itkExampleFilter.hxx"
#endif
```

8.5.1 Optional

A filter can be printed to an `std::ostream` (such as `std::cout`) by implementing the following method:

```
  void PrintSelf( std::ostream& os, Indent indent ) const;
```

and writing the name-value pairs of the filter parameters to the supplied output stream. This is particularly useful for debugging.

8.5.2 Useful Macros

Many convenience macros are provided by ITK, to simplify filter coding. Some of these are described below:

itkStaticConstMacro Declares a static variable of the given type, with the specified initial value.

itkGetMacro Defines an accessor method for the specified scalar data member. The convention is for data members to have a prefix of `m_`.

itkSetMacro Defines a mutator method for the specified scalar data member, of the supplied type. This will automatically set the `Modified` flag, so the filter stage will be executed on the next `Update()`.

itkBooleanMacro Defines a pair of `OnFlag` and `OffFlag` methods for a boolean variable `m_Flag`.

itkGetObjectMacro, itkSetObjectMacro Defines an accessor and mutator for an ITK object. The Get form returns a smart pointer to the object.

Much more useful information can be learned from browsing the source in `Code/Common/itkMacro.h` and for the `itk::Object` and `itk::LightObject` classes.

8.6 How To Write A Composite Filter

In general, most ITK filters implement one particular algorithm, whether it be image filtering, an information metric, or a segmentation algorithm. In the previous section, we saw how to write new filters from scratch. However, it is often very useful to be able to make a new filter by combining

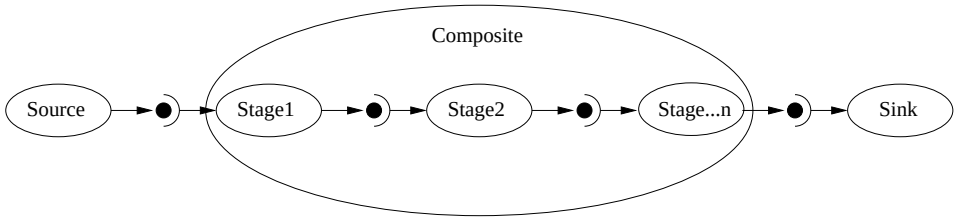

Figure 8.4: A Composite filter encapsulates a number of other filters.

two or more existing filters, which can then be used as a building block in a complex pipeline. This approach follows the Composite pattern [3], whereby the composite filter itself behaves just as a regular filter, providing its own (potentially higher level) interface and using other filters (whose detail is hidden to users of the class) for the implementation. This composite structure is shown in Figure 8.4, where the various `Stage-n` filters are combined into one by the `Composite` filter. The `Source` and `Sink` filters only see the interface published by the `Composite`. Using the Composite pattern, a composite filter can encapsulate a pipeline of arbitrary complexity. These can in turn be nested inside other pipelines.

8.6.1 Implementing a Composite Filter

There are a few considerations to take into account when implementing a composite filter. All the usual requirements for filters apply (as discussed above), but the following guidelines should be considered:

1. The template arguments it takes must be sufficient to instantiate all of the component filters. Each component filter needs a type supplied by either the implementor or the enclosing class. For example, an `ImageToImageFilter` normally takes an input and output image type (which may be the same). But if the output of the composite filter is a classified image, we need to either decide on the output type inside the composite filter, or restrict the choices of the user when she/he instantiates the filter.

2. The types of the component filters should be declared in the header, preferably with `protected` visibility. This is because the internal structure normally should not be visible to users of the class, but should be to descendent classes that may need to modify or customize the behavior.

3. The component filters should be private data members of the composite class, as in `FilterType::Pointer`.

4. The default constructor should build the pipeline by creating the stages and connect them together, along with any default parameter settings, as appropriate.

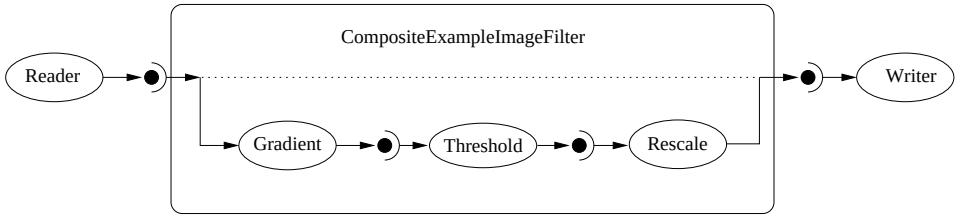

Figure 8.5: Example of a typical composite filter. Note that the output of the last filter in the internal pipeline must be grafted into the output of the composite filter.

5. The input and output of the composite filter need to be grafted on to the head and tail (respectively) of the component filters.

This grafting process is illustrated in Figure 8.5.

8.6.2 A Simple Example

The source code for this section can be found in the file `CompositeFilterExample.cxx`.

The composite filter we will build combines three filters: a gradient magnitude operator, which will calculate the first-order derivative of the image; a thresholding step to select edges over a given strength; and finally a rescaling filter, to ensure the resulting image data is visible by scaling the intensity to the full spectrum of the output image type.

Since this filter takes an image and produces another image (of identical type), we will specialize the ImageToImageFilter:

Next we include headers for the component filters:

```
#include "itkGradientMagnitudeImageFilter.h"
#include "itkThresholdImageFilter.h"
#include "itkRescaleIntensityImageFilter.h"
```

Now we can declare the filter itself. It is within the ITK namespace, and we decide to make it use the same image type for both input and output, so that the template declaration needs only one parameter. Deriving from `ImageToImageFilter` provides default behavior for several important aspects, notably allocating the output image (and making it the same dimensions as the input).

```
namespace itk {

template <class TImageType>
class CompositeExampleImageFilter :
    public ImageToImageFilter<TImageType, TImageType>
{
public:
```

Next we have the standard declarations, used for object creation with the object factory:

```
  typedef CompositeExampleImageFilter              Self;
  typedef ImageToImageFilter<TImageType,TImageType> Superclass;
  typedef SmartPointer<Self>                       Pointer;
  typedef SmartPointer<const Self>                  ConstPointer;
```

Here we declare an alias (to save typing) for the image's pixel type, which determines the type of the threshold value. We then use the convenience macros to define the Get and Set methods for this parameter.

```
  typedef typename TImageType::PixelType PixelType;

  itkGetMacro( Threshold, PixelType);
  itkSetMacro( Threshold, PixelType);
```

Now we can declare the component filter types, templated over the enclosing image type:

```
protected:

  typedef ThresholdImageFilter< TImageType >                    ThresholdType;
  typedef GradientMagnitudeImageFilter< TImageType, TImageType > GradientType;
  typedef RescaleIntensityImageFilter< TImageType, TImageType >  RescalerType;
```

The component filters are declared as data members, all using the smart pointer types.

```
  typename GradientType::Pointer     m_GradientFilter;
  typename ThresholdType::Pointer    m_ThresholdFilter;
  typename RescalerType::Pointer     m_RescaleFilter;

  PixelType m_Threshold;
};

} /* namespace itk */
```

The constructor sets up the pipeline, which involves creating the stages, connecting them together, and setting default parameters.

```
template <class TImageType>
CompositeExampleImageFilter<TImageType>
::CompositeExampleImageFilter()
{
  m_Threshold = 1;
  m_GradientFilter = GradientType::New();
  m_ThresholdFilter = ThresholdType::New();
  m_ThresholdFilter->SetInput( m_GradientFilter->GetOutput() );
  m_RescaleFilter = RescalerType::New();
  m_RescaleFilter->SetInput( m_ThresholdFilter->GetOutput() );
  m_RescaleFilter->SetOutputMinimum(
                                NumericTraits<PixelType>::NonpositiveMin());
  m_RescaleFilter->SetOutputMaximum(NumericTraits<PixelType>::max());
}
```

The `GenerateData()` is where the composite magic happens. First, we connect the first component filter to the inputs of the composite filter (the actual input, supplied by the upstream stage). Then we graft the output of the last stage onto the output of the composite, which ensures the filter regions are updated. We force the composite pipeline to be processed by calling `Update()` on the final stage, then graft the output back onto the output of the enclosing filter, so it has the result available to the downstream filter.

```
template <class TImageType>
void
CompositeExampleImageFilter<TImageType>::
GenerateData()
{
  m_GradientFilter->SetInput( this->GetInput() );

  m_ThresholdFilter->ThresholdBelow( this->m_Threshold );

  m_RescaleFilter->GraftOutput( this->GetOutput() );
  m_RescaleFilter->Update();
  this->GraftOutput( m_RescaleFilter->GetOutput() );
}
```

Finally we define the `PrintSelf` method, which (by convention) prints the filter parameters. Note how it invokes the superclass to print itself first, and also how the indentation prefixes each line.

```
template <class TImageType>
void
CompositeExampleImageFilter<TImageType>::
PrintSelf( std::ostream& os, Indent indent ) const
{
  Superclass::PrintSelf(os,indent);

  os
    << indent << "Threshold:" << this->m_Threshold
    << std::endl;
}

} /* end namespace itk */
```

It is important to note that in the above example, none of the internal details of the pipeline were exposed to users of the class. The interface consisted of the Threshold parameter (which happened to change the value in the component filter) and the regular ImageToImageFilter interface. This example pipeline is illustrated in Figure 8.5.

CHAPTER

NINE

SOFTWARE PROCESS

An outstanding feature of ITK is the software process used to develop, maintain and test the toolkit. The Insight Toolkit software continues to evolve rapidly due to the efforts of developers and users located around the world, so the software process is essential to maintaining its quality. If you are planning to contribute to ITK, or use the Git source code repository, you need to know something about this process (see 1.3 on page 5 to learn more about obtaining ITK using Git). This information will help you know when and how to update and work with the software as it changes. The following sections describe key elements of the process.

9.1 Git Source Code Repository

Git) is a tool for version control. It is a valuable resource for software projects involving multiple developers. The primary purpose of Git is to keep track of changes to software. Git date and version stamps every addition to files in the repository. Additionally, a user may set a tag to mark a particular of the whole software. Thus, it is possible to return to a particular state or point of time whenever desired. The differences between any two points is represented by a "diff" file, that is a compact, incremental representation of change. Git supports concurrent development so that two developers can edit the same file at the same time, that are then (usually) merged together without incident (and marked if there is a conflict). In addition, branches off of the main development trunk provide parallel development of software.

Developers and users can check out the software from the Git repository. When developers introduce changes in the system, Git facilitates to update the local copies of other developers and users by downloading only the differences between their local copy and the version on the repository. This is an important advantage for those who are interested in keeping up to date with the leading edge of the toolkit. Bug fixes can be obtained in this way as soon as they have been checked into the system.

ITK source code, data, and examples are maintained in a Git repository. The principal advantage of a system like Git is that it frees developers to try new ideas and introduce changes without fear of losing a previous working version of the software. It also provides a simple way to incrementally

update code as new features are added to the repository.

The ITK community use Git, and the Google web software tool Gerrit (http://review.source.kitware.com) to facilitate a structured, orderly method for developers to contribute new code and bug fixes to ITK. The Gerrit review process allows anyone to submit a proposed change to ITK, after which it will be reviewed by other developers before being approved and merged into ITK. For more information, see http://www.itk.org/Wiki/ITK/Git/Develop.

9.2 CDash Regression Testing System

One of the unique features of the ITK software process is its use of the CDash regression testing system (http://www.cdash.org). In a nutshell, what CDash does is to provide quantifiable feedback to developers as they check in new code and make changes. The feedback consists of the results of a variety of tests, and the results are posted on a publicly-accessible Web page (to which we refer as a *dashboard*) as shown in Figure 9.1. The most recent dashboard is accessible from http://www.itk.org/ITK/resources/testing.html). Since all users and developers of ITK can view the Web page, the CDash dashboard serves as a vehicle for developer communication, especially when new additions to the software is found to be faulty. The dashboard should be consulted before considering updating software via Git.

Note that CDash is independent of ITK and can be used to manage quality control for any software project. It is itself an open-source package and can be obtained from

http://www.cdash.org

CDash supports a variety of test types. These include the following.

Compilation. All source and test code is compiled and linked. Any resulting errors and warnings are reported on the dashboard.

Regression. Some ITK tests produce images as output. Testing requires comparing each test's output against a valid baseline image. If the images match then the test passes. The comparison must be performed carefully since many 3D graphics systems (e.g., OpenGL) produce slightly different results on different platforms.

Memory. Problems relating to memory such as leaks, uninitialized memory reads, and reads/ writes beyond allocated space can cause unexpected results and program crashes. ITK checks runtime memory access and management using Purify, a commercial package produced by Rational. (Other memory checking programs will be added in the future.)

PrintSelf. All classes in ITK are expected to print out all their instance variables (i.e., those with associated Set and Get methods) correctly. This test checks to make sure that this is the case.

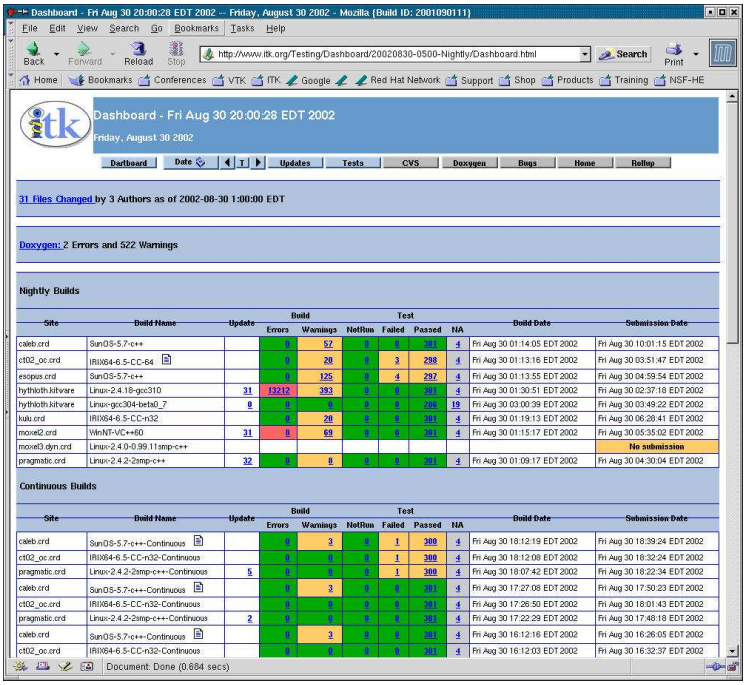

Figure 9.1: On-line presentation of the quality dashboard generated by CDash.

Unit. Each class in ITK should have a corresponding unit test where the class functionalities are exercised and quantitatively compared against expected results. These tests are typically written by the class developer and should endeavor to cover all lines of code including `Set/Get` methods and error handling.

Coverage. There is a saying among ITK developers: *If it isn't covered, then it's broke.* What this means is that code that is not executed during testing is likely to be wrong. The coverage tests identify lines that are not executed in the Insight Toolkit test suite, reporting a total percentage covered at the end of the test. While it is nearly impossible to bring the coverage to 100% because of error handling code and similar constructs that are rarely encountered in practice, the coverage numbers should be 75% or higher. Code that is not covered well enough requires additional tests.

Figure 9.1 shows the top-level dashboard web page. Each row in the dashboard corresponds to a particular platform (hardware + operating system + compiler). The data on the row indicates the number of compile errors and warnings as well as the results of running hundreds of small test programs. In this way the toolkit is tested both at compile time and run time.

When a user or developer decides to update ITK source code from Git it is important to first verify that the current dashboard is in good shape. This can be rapidly judged by the general coloration of

the dashboard. A green state means that the software is building correctly and it is a good day to start with ITK or to get an upgrade. A red state, on the other hand, is an indication of instability on the system and hence users should refrain from checking out or upgrading the source code.

Another nice feature of CDash is that it maintains a history of changes to the source code (by coordinating with Git) and summarizes the changes as part of the dashboard. This is useful for tracking problems and keeping up to date with new additions to ITK.

9.3 Working The Process

The ITK software process functions across three cycles—the continuous cycle, the daily cycle, and the release cycle.

The continuous cycle revolves around the actions of developers as they check code into Git. When changed or new code is checked into Git, the CDash continuous testing process kicks in. A small number of tests are performed (including compilation), and if something breaks, email is sent to all developers who checked code in during the continuous cycle. Developers are expected to fix the problem immediately.

The daily cycle occurs over a 24-hour period. Changes to the source base made during the day are extensively tested by the nightly CDash regression testing sequence. These tests occur on different combinations of computers and operating systems located around the world, and the results are posted every day to the CDash dashboard. Developers who checked in code are expected to visit the dashboard and ensure their changes are acceptable—that is, they do not introduce compilation errors or warnings, or break any other tests including regression, memory, PrintSelf, and Set/Get. Again, developers are expected to fix problems immediately.

The release cycle occurs a small number of times a year. This requires tagging and branching the Git repository, updating documentation, and producing new release packages. Although additional testing is performed to insure the consistency of the package, keeping the daily Git build error free minimizes the work required to cut a release.

ITK users typically work with releases, since they are the most stable. Developers work with the Git repository, or sometimes with periodic release snapshots, in order to take advantage of newly-added features. It is extremely important that developers watch the dashboard carefully, and *update their software only when the dashboard is in good condition (i.e., is "green")*. Failure to do so can cause significant disruption if a particular day's software release is unstable.

9.4 The Effectiveness of the Process

The effectiveness of this process is profound. By providing immediate feedback to developers through email and Web pages (e.g., the dashboard), the quality of ITK is exceptionally high, especially considering the complexity of the algorithms and system. Errors, when accidently introduced,

are caught quickly, as compared to catching them at the point of release. To wait to the point of release is to wait too long, since the causal relationship between a code change or addition and a bug is lost. The process is so powerful that it routinely catches errors in vendor's graphics drivers (e.g., OpenGL drivers) or changes to external subsystems such as the VXL/VNL numerics library. All of these tools that make up the process (CMake, Git, and CDash) are open-source. Many large and small systems such as VTK (The Visualization Toolkit http://www.vtk.org) use the same process with similar results. We encourage the adoption of the process in your environment.

Appendices

APPENDIX

ONE

LICENSES

A.1 Insight Toolkit License

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation

source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the

Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or,

within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be

liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

```
   See the License for the specific language governing permissions and
   limitations under the License.
```

A.2 Third Party Licenses

The Insight Toolkit bundles a number of third party libraries that are used internally. The licenses of these libraries are as follows.

A.2.1 DICOM Parser

```
/*=========================================================================

  Program:   DICOMParser
  Module:    Copyright.txt
  Language:  C++
  Date:      $Date$
  Version:   $Revision$


Copyright (c) 2003 Matt Turek
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

 * The name of Matt Turek nor the names of any contributors may be used to
   endorse or promote products derived from this software without specific
   prior written permission.

 * Modified source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS''
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

```
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

============================================================================*/
```

A.2.2 Double Conversion

```
Copyright 2006-2011, the V8 project authors. All rights reserved.
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.
    * Neither the name of Google Inc. nor the names of its
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

A.2.3 Expat

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

A.2.4 GDCM

```
/*=========================================================================

  Program: GDCM (Grassroots DICOM). A DICOM library
  Module:  $URL: https://gdcm.svn.sourceforge.net/svnroot/gdcm/trunk/Copyright.txt $

Copyright (c) 2006-2010 Mathieu Malaterre
Copyright (c) 1993-2005 CREATIS
(CREATIS = Centre de Recherche et d'Applications en Traitement de l'Image)
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.
```

```
 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

 * Neither name of Mathieu Malaterre, or CREATIS, nor the names of any
   contributors (CNRS, INSERM, UCB, Universite Lyon I), may be used to
   endorse or promote products derived from this software without specific
   prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS''
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=========================================================================*/
```

A.2.5 GIFTI

```
The gifticlib code is released into the public domain.  Developers are
encouraged to incorporate the library into their application, and to
contribute changes or enhancements to gifticlib.

Author: Richard Reynolds, SSCC, DIRP, NIMH, National Institutes of Health
        May 13, 2008 (release version 1.0.0)

http://www.nitrc.org/projects/gifti
```

A.2.6 HDF5

```
Copyright Notice and License Terms for
HDF5 (Hierarchical Data Format 5) Software Library and Utilities
-----------------------------------------------------------------------------

HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 2006-2011 by The HDF Group.
```

The Independent JPEG Group (JPEG), Jean-loup Gailly and Mark Adler (gzip),
and Digital Equipment Corporation (DEC).

Portions of HDF5 were developed with support from the Lawrence Berkeley
National Laboratory (LBNL) and the United States Department of Energy
under Prime Contract No. DE-AC02-05CH11231.

Portions of HDF5 were developed with support from the University of
California, Lawrence Livermore National Laboratory (UC LLNL).
The following statement applies to those portions of the product and must
be retained in any redistribution of source code, binaries, documentation,
and/or accompanying materials:

> This work was partially produced at the University of California,
> Lawrence Livermore National Laboratory (UC LLNL) under contract
> no. W-7405-ENG-48 (Contract 48) between the U.S. Department of Energy
> (DOE) and The Regents of the University of California (University)
> for the operation of UC LLNL.
>
> DISCLAIMER:
> This work was prepared as an account of work sponsored by an agency of
> the United States Government. Neither the United States Government nor
> the University of California nor any of their employees, makes any
> warranty, express or implied, or assumes any liability or responsibility
> for the accuracy, completeness, or usefulness of any information,
> apparatus, product, or process disclosed, or represents that its use
> would not infringe privately- owned rights. Reference herein to any
> specific commercial products, process, or service by trade name,
> trademark, manufacturer, or otherwise, does not necessarily constitute
> or imply its endorsement, recommendation, or favoring by the United
> States Government or the University of California. The views and
> opinions of authors expressed herein do not necessarily state or reflect
> those of the United States Government or the University of California,
> and shall not be used for advertising or product endorsement purposes.

A.2.7 JPEG

The authors make NO WARRANTY or representation, either express or implied,
with respect to this software, its quality, accuracy, merchantability, or
fitness for a particular purpose. This software is provided "AS IS", and you,
its user, assume the entire risk as to its quality and accuracy.

This software is copyright (C) 1991-2010, Thomas G. Lane, Guido Vollbeding.
All Rights Reserved except as specified below.

Permission is hereby granted to use, copy, modify, and distribute this
software (or portions thereof) for any purpose, without fee, subject to these
conditions:
(1) If any part of the source code for this software is distributed, then this
README file must be included, with this copyright and no-warranty notice
unaltered; and any additions, deletions, or changes to the original files
must be clearly indicated in accompanying documentation.
(2) If only executable code is distributed, then the accompanying
documentation must state that "this software is based in part on the work of
the Independent JPEG Group".
(3) Permission for use of this software is granted only if the user accepts
full responsibility for any undesirable consequences; the authors accept
NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code,
not just to the unmodified library. If you use our work, you ought to
acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name
in advertising or publicity relating to this software or products derived from
it. This software may be referred to only as "the Independent JPEG Group's
software".

We specifically permit and encourage the use of this software as the basis of
commercial products, provided that all warranty or liability claims are
assumed by the product vendor.

ansi2knr.c is included in this distribution by permission of L. Peter Deutsch,
sole proprietor of its copyright holder, Aladdin Enterprises of Menlo Park, CA.
ansi2knr.c is NOT covered by the above copyright and conditions, but instead
by the usual distribution terms of the Free Software Foundation; principally,
that you must include source code if you redistribute it. (See the file

```
ansi2knr.c for full details.)  However, since ansi2knr.c is not needed as part
of any program generated from the IJG code, this does not limit you more than
the foregoing paragraphs do.

The Unix configuration script "configure" was produced with GNU Autoconf.
It is copyright by the Free Software Foundation but is freely distributable.
The same holds for its supporting scripts (config.guess, config.sub,
ltmain.sh).  Another support script, install-sh, is copyright by X Consortium
but is also freely distributable.

The IJG distribution formerly included code to read and write GIF files.
To avoid entanglement with the Unisys LZW patent, GIF reading support has
been removed altogether, and the GIF writer has been simplified to produce
"uncompressed GIFs".  This technique does not use the LZW algorithm; the
resulting GIF files are larger than usual, but are readable by all standard
GIF decoders.

We are required to state that
    "The Graphics Interchange Format(c) is the Copyright property of
    CompuServe Incorporated.  GIF(sm) is a Service Mark property of
    CompuServe Incorporated."
```

A.2.8 KWSys

```
KWSys - Kitware System Library
Copyright 2000-2009 Kitware, Inc., Insight Software Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.

* Neither the names of Kitware, Inc., the Insight Software Consortium,
  nor the names of their contributors may be used to endorse or promote
  products derived from this software without specific prior written
  permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

A.2.9 MetaIO

```
The following license applies to all code, without exception,
in the MetaIO library.

/*=========================================================================

Copyright (c) 1999-2007 Insight Software Consortium
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.

 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

 * The name of the Insight Software Consortium, nor the names of any
   consortium members, nor of any contributors, may be used to endorse or
   promote products derived from this software without specific prior written
   permission.

 * Modified source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS ``AS IS''
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

```
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

=============================================================================*/
```

A.2.10 Netlib's SLATEC

This code is in the public domain. From http://www.netlib.org/slatec/guide:

```
SECTION 4.  OBTAINING THE LIBRARY

The Library is in the public domain and distributed by the Energy Science
and Technology Software Center.

              Energy Science and Technology Software Center
              P.O. Box 1020
              Oak Ridge, TN  37831

              Telephone  615-576-2606
              E-mail  estsc%a1.adonis.mrouter@zeus.osti.gov
```

A.2.11 NIFTI

```
Niftilib has been developed by members of the NIFTI DFWG and volunteers in the
neuroimaging community and serves as a reference implementation of the nifti-1
file format.

http://nifti.nimh.nih.gov/

Nifticlib code is released into the public domain, developers are encouraged to
incorporate niftilib code into their applications, and, to contribute changes
and enhancements to niftilib.
```

A.2.12 NrrdIO

```
------------------------------------------------------------------------
License ---------------------------------------------------------------
------------------------------------------------------------------------

  NrrdIO: stand-alone code for basic nrrd functionality
  Copyright (C) 2013, 2012, 2011, 2010, 2009  University of Chicago
  Copyright (C) 2008, 2007, 2006, 2005  Gordon Kindlmann
  Copyright (C) 2004, 2003, 2002, 2001, 2000, 1999, 1998  University of Utah

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any
  damages arising from the use of this software.

  Permission is granted to anyone to use this software for any
  purpose, including commercial applications, and to alter it and
  redistribute it freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must
     not claim that you wrote the original software. If you use this
     software in a product, an acknowledgment in the product
     documentation would be appreciated but is not required.

  2. Altered source versions must be plainly marked as such, and must
     not be misrepresented as being the original software.

  3. This notice may not be removed or altered from any source distribution.

------------------------------------------------------------------------
General information ---------------------------------------------------
------------------------------------------------------------------------

** NOTE: These source files have been copied and/or modified from Teem,
** <http://teem.sf.net>.  Teem is licensed under a weakened GNU Lesser Public
** License (the weakening is to remove burdens on those releasing binaries
** that statically link against Teem) . The non-reciprocal licensing defined
** above applies to only the source files in the NrrdIO distribution, and not
** to Teem.

NrrdIO is a modified and highly abbreviated version of the Teem. NrrdIO
contains only the source files (or portions thereof) required for
creating and destroying nrrds, and for getting them into and out of
```

files. The NrrdIO sources are created from the Teem sources by using
GNU Make (pre-GNUmakefile in the NrrdIO distribution).

NrrdIO makes it very easy to add support for the NRRD file format to your
program, which is a good thing considering and design and flexibility of the
NRRD file format, and the existence of the "unu" command-line tool for
operating on nrrds. Using NrrdIO requires exactly one header file,
"NrrdIO.h", and exactly one library, libNrrdIO.

Currently, the API presented by NrrdIO is a strict subset of the Teem API.
There is no additional encapsulation or abstraction. This could be annoying
in the sense that you still have to deal with the biff (for error messages)
and the air (for utilities) library function calls. Or it could be good and
sane in the sense that code which uses NrrdIO can be painlessly "upgraded" to
use more of Teem. Also, the API documentation for the same functionality in
Teem will apply directly to NrrdIO.

NrrdIO was originally created with the help of Josh Cates in order to add
support for the NRRD file format to the Insight Toolkit (ITK).

NrrdIO API crash course ----------

Please read <http://teem.sourceforge.net/nrrd/lib.html>. The functions that
are explained in detail are all present in NrrdIO. Be aware, however, that
NrrdIO currently supports ONLY the NRRD file format, and not: PNG, PNM, VTK,
or EPS.

The functionality in Teem's nrrd library which is NOT in NrrdIO is basically
all those non-trivial manipulations of the values in the nrrd, or their
ordering in memory. Still, NrrdIO can do a fair amount, namely all the
functions listed in these sections of the "Overview of rest of API" in the
above web page:

- Basic "methods"
- Manipulation of per-axis meta-information
- Utility functions
- Comments in nrrd
- Key/value pairs
- Endianness (byte ordering)
- Getting/Setting values (crude!)
- Input from, Output to files

```
------------------------------------------------------------------------
Files comprising NrrdIO -----------------------------------------------
------------------------------------------------------------------------

NrrdIO.h: The single header file that declares all the functions and variables
that NrrdIO provides.

sampleIO.c: Tiny little command-line program demonstrating the basic NrrdIO
API.  Read this for examples of how NrrdIO is used to read and write NRRD
files.

CMakeLists.txt: to build NrrdIO with CMake

pre-GNUmakefile: how NrrdIO sources are created from the Teem
sources. Requires that TEEM_SRC_ROOT be set, and uses the following two files.

tail.pl, unteem.pl: used to make small modifications to the source files to
convert them from Teem to NrrdIO sources

mangle.pl: used to generate a #include file for name-mangling the external
symbols in the NrrdIO library, to avoid possible problems with programs
that link with both NrrdIO and the rest of Teem.

preamble.c: the preamble describing the non-copyleft licensing of NrrdIO.

qnanhibit.c: discover a variable which, like endianness, is architecture
dependent and which is required for building NrrdIO (as well as Teem), but
unlike endianness, is completely obscure and unheard of.

encodingBzip2.c, formatEPS.c, formatPNG.c, formatPNM.c, formatText.c,
formatVTK.c: These files create stubs for functionality which is fully present
in Teem, but which has been removed from NrrdIO in the interest of simplicity.
The filenames are in fact unfortunately misleading, but they should be
understood as listing the functionality that is MISSING in NrrdIO.

All other files: copied/modified from the air, biff, and nrrd libraries of
Teem.
```

A.2.13 OpenJPEG

```
/*
 * Copyright (c) 2002-2012, Communications and Remote Sensing Laboratory,
```

```
 * Universite catholique de Louvain (UCL), Belgium
 * Copyright (c) 2002-2012, Professor Benoit Macq
 * Copyright (c) 2003-2012, Antonin Descampe
 * Copyright (c) 2003-2009, Francois-Olivier Devaux
 * Copyright (c) 2005, Herve Drolon, FreeImage Team
 * Copyright (c) 2002-2003, Yannick Verschueren
 * Copyright (c) 2001-2003, David Janssens
 * Copyright (c) 2011-2012, Centre National d'Etudes Spatiales (CNES), France
 * Copyright (c) 2012, CS Systemes d'Information, France
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS 'AS IS'
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */
```

A.2.14 PNG

```
This copy of the libpng notices is provided for your convenience.  In case of
any discrepancy between this copy and the notices in the file png.h that is
included in the libpng distribution, the latter shall prevail.

COPYRIGHT NOTICE, DISCLAIMER, and LICENSE:

If you modify libpng you may insert additional notices immediately following
```

```
this sentence.

This code is released under the libpng license.

libpng versions 1.2.6, August 15, 2004, through 1.6.6, September 16, 2013, are
Copyright (c) 2004, 2006-2013 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-1.2.5
with the following individual added to the list of Contributing Authors

   Cosmin Truta

libpng versions 1.0.7, July 1, 2000, through 1.2.5 - October 3, 2002, are
Copyright (c) 2000-2002 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-1.0.6
with the following individuals added to the list of Contributing Authors

   Simon-Pierre Cadieux
   Eric S. Raymond
   Gilles Vollant

and with the following additions to the disclaimer:

   There is no warranty against interference with your enjoyment of the
   library or against infringement.  There is no warranty that our
   efforts or the library will fulfill any of your particular purposes
   or needs.  This library is provided with all faults, and the entire
   risk of satisfactory quality, performance, accuracy, and effort is with
   the user.

libpng versions 0.97, January 1998, through 1.0.6, March 20, 2000, are
Copyright (c) 1998, 1999 Glenn Randers-Pehrson, and are
distributed according to the same disclaimer and license as libpng-0.96,
with the following individuals added to the list of Contributing Authors:

   Tom Lane
   Glenn Randers-Pehrson
   Willem van Schaik

libpng versions 0.89, June 1996, through 0.96, May 1997, are
Copyright (c) 1996, 1997 Andreas Dilger
Distributed according to the same disclaimer and license as libpng-0.88,
with the following individuals added to the list of Contributing Authors:
```

```
   John Bowler
   Kevin Bracey
   Sam Bushell
   Magnus Holmgren
   Greg Roelofs
   Tom Tanner

libpng versions 0.5, May 1995, through 0.88, January 1996, are
Copyright (c) 1995, 1996 Guy Eric Schalnat, Group 42, Inc.

For the purposes of this copyright and license, "Contributing Authors"
is defined as the following set of individuals:

   Andreas Dilger
   Dave Martindale
   Guy Eric Schalnat
   Paul Schmidt
   Tim Wegner

The PNG Reference Library is supplied "AS IS".  The Contributing Authors
and Group 42, Inc. disclaim all warranties, expressed or implied,
including, without limitation, the warranties of merchantability and of
fitness for any purpose.  The Contributing Authors and Group 42, Inc.
assume no liability for direct, indirect, incidental, special, exemplary,
or consequential damages, which may result from the use of the PNG
Reference Library, even if advised of the possibility of such damage.

Permission is hereby granted to use, copy, modify, and distribute this
source code, or portions hereof, for any purpose, without fee, subject
to the following restrictions:

1. The origin of this source code must not be misrepresented.

2. Altered versions must be plainly marked as such and must not
   be misrepresented as being the original source.

3. This Copyright notice may not be removed or altered from any
   source or altered source distribution.

The Contributing Authors and Group 42, Inc. specifically permit, without
fee, and encourage the use of this source code as a component to
supporting the PNG file format in commercial products.  If you use this
source code in a product, acknowledgment is not required but would be
```

```
appreciated.


A "png_get_copyright" function is available, for convenient use in "about"
boxes and the like:

   printf("%s",png_get_copyright(NULL));

Also, the PNG logo (in PNG format, of course) is supplied in the
files "pngbar.png" and "pngbar.jpg (88x31) and "pngnow.png" (98x31).

Libpng is OSI Certified Open Source Software.  OSI Certified Open Source is a
certification mark of the Open Source Initiative.

Glenn Randers-Pehrson
glennrp at users.sourceforge.net
September 16, 2013
```

A.2.15 TIFF

```
Copyright (c) 1988-1997 Sam Leffler
Copyright (c) 1991-1997 Silicon Graphics, Inc.

Permission to use, copy, modify, distribute, and sell this software and
its documentation for any purpose is hereby granted without fee, provided
that (i) the above copyright notices and this permission notice appear in
all copies of the software and related documentation, and (ii) the names of
Sam Leffler and Silicon Graphics may not be used in any advertising or
publicity relating to the software without the specific, prior written
permission of Sam Leffler and Silicon Graphics.

THE SOFTWARE IS PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND,
EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

IN NO EVENT SHALL SAM LEFFLER OR SILICON GRAPHICS BE LIABLE FOR
ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND,
OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF
LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

A.2.16 VNL

```
#ifndef vxl_copyright_h_
#define vxl_copyright_h_

// <begin copyright notice>
// ---------------------------------------------------------------------------
//
//                Copyright (c) 2000-2003 TargetJr Consortium
//               GE Corporate Research and Development (GE CRD)
//                             1 Research Circle
//                            Niskayuna, NY 12309
//                            All Rights Reserved
//              Reproduction rights limited as described below.
//
//      Permission to use, copy, modify, distribute, and sell this software
//      and its documentation for any purpose is hereby granted without fee,
//      provided that (i) the above copyright notice and this permission
//      notice appear in all copies of the software and related documentation,
//      (ii) the name TargetJr Consortium (represented by GE CRD), may not be
//      used in any advertising or publicity relating to the software without
//      the specific, prior written permission of GE CRD, and (iii) any
//      modifications are clearly marked and summarized in a change history
//      log.
//
//      THE SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND,
//      EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
//      WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
//      IN NO EVENT SHALL THE TARGETJR CONSORTIUM BE LIABLE FOR ANY SPECIAL,
//      INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND OR ANY
//      DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
//      WHETHER OR NOT ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR ON
//      ANY THEORY OF LIABILITY ARISING OUT OF OR IN CONNECTION WITH THE
//      USE OR PERFORMANCE OF THIS SOFTWARE.
//
// ---------------------------------------------------------------------------
// <end copyright notice>

#endif // vxl_copyright_h_
```

A.2.17 ZLIB

```
Acknowledgments:

  The deflate format used by zlib was defined by Phil Katz. The deflate
  and zlib specifications were written by L. Peter Deutsch. Thanks to all the
  people who reported problems and suggested various improvements in zlib;
  they are too numerous to cite here.

Copyright notice:

 (C) 1995-2004 Jean-loup Gailly and Mark Adler

  This software is provided 'as-is', without any express or implied
  warranty.  In no event will the authors be held liable for any damages
  arising from the use of this software.

  Permission is granted to anyone to use this software for any purpose,
  including commercial applications, and to alter it and redistribute it
  freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
     claim that you wrote the original software. If you use this software
     in a product, an acknowledgment in the product documentation would be
     appreciated but is not required.
  2. Altered source versions must be plainly marked as such, and must not be
     misrepresented as being the original software.
  3. This notice may not be removed or altered from any source distribution.

  Jean-loup Gailly        Mark Adler
  jloup@gzip.org          madler@alumni.caltech.edu

If you use the zlib library in a product, we would appreciate *not*
receiving lengthy legal documents to sign. The sources are provided
for free but without warranty of any kind.  The library has been
entirely written by Jean-loup Gailly and Mark Adler; it does not
include third-party code.

If you redistribute modified sources, we would appreciate that you include
in the file ChangeLog history information documenting your changes. Please
read the FAQ for more information on the distribution of modified source
versions.
```

BIBLIOGRAPHY

[1] M. H. Austern. *Generic Programming and the STL:*. Professional Computing Series. Addison-Wesley, 1999. 3.2.1

[2] K.R. Castleman. *Digital Image Processing*. Prentice Hall, Upper Saddle River, NJ, 1996. 6.4.1, 6.4.2

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 3.2.6, 4.3.9, 8.6

[4] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Addison-Wesley, Reading, MA, 1993. 6.4.1, 6.4.1, 6.4.2

[5] H. Gray. *Gray's Anatomy*. Merchant Book Company, sixteenth edition, 2003. 4.1.5

[6] H. Lodish, A. Berk, S. Zipursky, P. Matsudaira, D. Baltimore, and J. Darnell. *Molecular Cell Biology*. W. H. Freeman and Company, 2000. 4.1.5

[7] D. Malacara. *Color Vision and Colorimetry: Theory and Applications*. SPIE PRESS, 2002. 4.1.5, 4.1.5

[8] D. Musser and A. Saini. *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley, 1996. 3.2.1

[9] G. Wyszecki. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley-Interscience, 2000. 4.1.5, 4.1.5

INDEX