IGSTK: The Book
DRAFT VERSION

Kevin Cleary
and the *Insight Software Consortium*

December 6, 2006

http://www.igstk.org
Email: igstk-developers@public.kitware.com

A Great Project, needs a great Quote.

Kevin will think about this one. . .

Abstract

The Image Guided Surgery Toolkit (IGSTK) is an open-source software toolkit for performing . . .

IGSTK is implemented in C++. It is cross-platform, using a build environment known as CMake to manage the compilation process in a platform-independent way.

Because IGSTK is an open-source project, developers from around the world can use, debug, maintain, and extend the software. IGSTK uses a model of software development referred to as Extreme Programming. Extreme Programming collapses the usual software creation methodology into a simultaneous and iterative process of design-implement-test-release. The key features of Extreme Programming are communication and testing. Communication among the members of the IGSTK community is what helps manage the rapid evolution of the software. Testing is what keeps the software stable. In IGSTK, an extensive testing process (using a system known as Dart) is in place that measures the quality on a daily basis. The IGSTK Testing Dashboard is posted continuously, reflecting the quality of the software at any moment.

This book is a guide to using and developing with IGSTK.

Contributors

The Image Guided Surgery Toolkit (IGSTK) has been created by the efforts of many talented individuals and prestigious organizations.

This book lists a few of these contributors in the following paragraphs. Not all developers of IGSTK are credited here, so please visit the Web pages at http://www.igstk.org/HTML/About.htm for the names of additional contributors, as well as checking the CVS source logs for code contributions.

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

Kevin Cleary

Luis Ibáñez

David Gobi

Patrick Cheng

Kevin Gary

Julien Jomier

Andinet Enquobahrie

Hui Zhang

M. Brian Black

Hee-su Kim

Stephen Aylward

Rick Avila

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Overview and Design

CHAPTER

ONE

Introduction

Welcome to the *IGSTK: The Book* for *Image-Guided Surgery*.

IGSTK is an open-source[1] software toolkit designed to enable biomedical researchers to rapidly prototype and create new applications for image-guided surgery. The purpose of this software guide is to help you learn how to use IGSTK so that you can more easily create your application. The material is explained using a number of examples that we encourage you to compile and run.

IGSTK is built on top of three open source software packages:

- The Insight Segmentation and Registration Toolkit (ITK)
- The Visualization Toolkit (VTK)
- The Fast Light Toolkit (FLTK) for the user interface, although other GUI toolkits can be accommodated

Some familiarity with these open source packages will be required to effectively use IGSTK. In addition, IGSTK uses CMake to configure the build process in multiple platforms.

IGSTK is an open-source software system. What this means is that the user community has great impact on the future evolution of the software. Users can make significant contributions to IGSTK by providing bug reports, bug fixes, test cases, new classes, and other feedback. You are encouraged to contribute your ideas to the community through the user mailing list which can be located through `www.igstk.org`.

As opposed to popular belief, open-source software projects cannot simply be modified by anybody. The official version of most open source packages are maintained by a small core of very skilled developers. Users of those packages have the freedom to download the source code and to modify it in their own machines if they like, but they can not simply put those modifications back into the official version of the package without passing throught the scrutiny of the core developers. Since IGSTK is intended to be used in the operating room its developers

[1] Distributed under a BSD-like license

makes emphasis on robustness motivated by the directive of protecting patient safety. In this context, IGSTK is a system that is open for discussion and willing to receive contributions from the community, but that will adapt and throughly tests those contributions to make sure that they satisfy the quality standards required for safety-critical applications.

1.1 Rationale and Background

The creation of the toolkit was motivated by the following scenario:

> Imagine that you are a biomedical researcher and you want to develop an image-guided surgery application. This application should include the ability to display DICOM medical images, functionality for registration and segmentation, and an interface to the AURORA electromagnetic tracking system. You have a clinical partner who is anxious to test your completed system, so you begin your software design and hope to have a prototype within a few months. But progress is slow, as you have to develop all your own code and understand the nuances of DICOM, segmentation, registration, and the AURORA interface. After a year or so, you complete your prototype and proudly show it to your clinical partner. But your clinical partner now wants to make several changes and add some more features, so you are back to the drawing board for another lengthy development cycle.
>
> Now imagine the same scenario using the IGSTK toolkit. You still need to develop a specification for your application, but implementation is greatly simplified as standard components for reading DICOM images, image display, segmentation, registration, and an AURORA interface are provided. You start by reading the toolkit documentation and looking at the example applications. You then either put together a set of components to build your application or modify one of the example applications. Because the toolkit is open-source and uses BSD[2] licensing, you have access to all the source code and are free to incorporate the code in your program whether is an academic or commercial application. You are able to fairly quickly put together a prototype. When your clinical partner requests changes, this is also easily done. Your completed application is a success. You license it to a big company and retire on the island of Fiji.

While the example above is fictitious, most biomedical researcher can probably identify with this scenario. The reality is that software development is a large portion of the work in many research labs today. Most software projects are started from scratch and a great deal of time and effort is spent “re-inventing the wheel”.

This was the situation at Georgetown University Medical Center, where one of the authors (KC) has been leading a group of researchers in developing image-guided systems for abdominal interventions for the past five years. After several years of developing new software applications,

[2]BSD originally stood for Berkeley Source Distribution and the BSD License was the license that the BSD software (a version of Unix) was distributed under. This license is used by many open-source implementations today

the research group has been attempting to standardize their software process based on the open-source software packages VTK (Visualization Toolkit) and ITK (Insight Segmentation and Registration Toolkit). While these packages provided an excellent start, the group noted that further progress could be made by developing a set of components specifically for image-guided applications. A partnership was formed with Kitware Incorporated, a leading open-source software company. A small business proposal was submitted to the National Institutes of Health and a grant award was received to develop the toolkit. Several other collaborators, including UNC, Atamai Inc., and the University of Arizona later joined the project. This book and the associated software is the result of that effort.

1.2 Organization

This book is divided into three parts, each of which is further divided into several chapters. Part I is a general introduction to IGSTK, with the next few chapters on installation, architecture, requirements, and software development process. This part will give the readers a general overview about the IGSTK project, how it started, its design philosophy and development process, and also a chapter showing users how to install and start using this toolkit. Part II guides user through each single component in the toolkit such as state machine, tracker, spatial object, registration etc. These chapters will explain how we design and implement these components and how you can use them. Part III and IV are dedicate to help user learn to build applications from simple "Hello World" to more complex ones based on clinical procedures.

1.3 Software Organization

The following sections describe the directory contents, summarize the software functionality in each directory, and locate the documentation and data.

1.3.1 Obtaining the Software

There are three different ways to access the IGSTK source code (see Section 1.4 on page 6).

1. from periodic releases available on the IGSTK Web site,
2. from direct access to the CVS source code repository.

Official releases are available a few times a year and announced on the ITK Web pages and mailing lists. However, they may not provide the latest and greatest features of the toolkit. In general, the periodic releases and CD-ROM releases are the same, except that the CD release typically contains additional resources and data. CVS access provides immediate access to the latest toolkit additions, but on any given day the source code may not be stable as compared to the official releases—i.e., the code may not compile, it may crash, or it might even produce incorrect results.

This software guide assumes that you are working with the official ITK version 2.4 release (available on the ITK Web site). If you are a new user, we highly recommend that you use the released version of the software. It is stable, consistent, and better tested than the code available from the CVS repository. Later, as you gain experience with ITK, you may wish to work from the CVS repository. However, if you do so, please be aware of the ITK quality testing dashboard. The Insight Toolkit is heavily tested using the open-source DART regression testing system (`http://public.kitware.com/dashboard.php`). Before updating the CVS repository, make sure that the dashboard is *green* indicating stable code. If not green it is likely that your software update is unstable. (Learn more about the ITK quality dashboard in Section **??** on page **??**.)

1.4 Downloading IGSTK

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

`http://www.itk.org/HTML/Download.php`

In order to track the kind of applications for which ITK is being used, you will be asked to complete a form prior to downloading the software. The information you provide in this form will help developers to get a better idea of the interests and skills of the toolkit users. It also assists in future funding requests to sponsoring agencies.

Once you fill out this form you will have access to the download page where two options for obtaining the software will be found. (This page can be book marked to facilitate subsequent visits to the download site without having to complete any form again.) You can get the tarball of a stable release or you can get the development version through CVS. The release version is stable and dependable but may lack the latest features of the toolkit. The CVS version will have the latest additions but is inherently unstable and may contain components with work in progress. The following sections describe the details of each one of these two alternatives.

1.4.1 Downloading Packaged Releases

Please read the `GettingStarted.txt`[3] document first. It will give you an overview of the download and installation processes. Then choose the tarball that better fits your system. The options are `.zip` and `.tgz` files. The first type is better suited for MS-Windows while the second one is the preferred format for UNIX systems.

Once you unzip or untar the file a directory called `Insight` will be created in your disk and you will be ready for starting the configuration process described in Section **??** on page **??**.

[3]http://www.itk.org/HTML/GettingStarted.txt

1.4.2 Downloading from CVS

The Concurrent Versions System (CVS) is a tool for software version control [5]. Generally only developers should be using CVS, so here we assume that you know what CVS is and how to use it. For more information about CVS please see Section **??** on page **??**. (Note: please make sure that you access the software via CVS only when the ITK Quality Dashboard indicates that the code is stable. Learn more about the Quality Dashboard at **??** on page **??**.)

Access ITK via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight login
(respond with password "insight")

cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co Insight
```

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

```
cvs update -d -P
```

Once you obtain the software you are ready to configure and compile it (see Section **??** on page **??**). First, however, we recommend that you join the mailing list and read the following sections describing the organization of the software.

1.4.3 Join the Mailing List

It is strongly recommended that you join the users mailing list. This is one of the primary resources for guidance and help regarding the use of the toolkit. You can subscribe to the users list online at

`http://www.itk.org/HTML/MailingLists.htm`

The insight-users mailing list is also the best mechanism for expressing your opinions about the toolkit and to let developers know about features that you find useful, desirable or even unnecessary. ITK developers are committed to creating a self-sustaining open-source ITK community. Feedback from users is fundamental to achieving this goal.

1.4.4 Directory Structure

To begin your ITK odyssey, you will first need to know something about ITK's software organization and directory structure. Even if you are installing pre-compiled binaries, it is helpful to know enough to navigate through the code base to find examples, code, and documentation.

ITK is organized into several different modules, or CVS checkouts. If you are using an official release or CD release, you will see three important modules: the `Insight`, `InsightDocuments` and `InsightApplications` modules. The source code, examples and applications are found in the `Insight` module; documentation, tutorials, and material related to the design and marketing of ITK are found in `InsightDocuments`; and fairly complex applications using ITK (and other systems such as VTK, Qt, and FLTK) are available from `InsightApplications`. Usually you will work with the `Insight` module unless you are a developer, are teaching a course, or are looking at the details of various design documents. The `InsightApplications` module should only be downloaded and compiled once the `Insight` module is functioning properly.

The `Insight` module contains the following subdirectories:

- `Insight/Auxiliary`—code that interfaces packages to ITK.
- `Insight/Code`—the heart of the software; the location of the majority of the source code.
- `Insight/Documentation`—a compact subset of documentation to get users started with ITK.
- `Insight/Examples`—a suite of simple, well-documented examples used by this guide and to illustrate important ITK concepts.
- `Insight/Testing`—a large number of small programs used to test ITK. These examples tend to be minimally documented but may be useful to demonstrate various system concepts. These tests are used by DART to produce the ITK Quality Dashboard (see Section **??** on page **??**.)
- `Insight/Utilities`—supporting software for the ITK source code. For example, DART and Doxygen support, as well as libraries such as `png` and `zlib`.
- `Insight/Validation`—a series of validation case studies including the source code used to produce the results.
- `Insight/Wrapping`—support for the CABLE wrapping tool. CABLE is used by ITK to build interfaces between the C++ library and various interpreted languages (currently Tcl and Python are supported).

The source code directory structure—found in `Insight/Code`—is important to understand since other directory structures (such as the `Testing` and `Wrapping` directories) shadow the structure of the `Insight/Code` directory.

- `Insight/Code/Common`—core classes, macro definitions, typedefs, and other software constructs central to ITK.
- `Insight/Code/Numerics`—mathematical library and supporting classes. (Note: ITK's mathematical library is based on the VXL/VNL software package `http://vxl.sourceforge.net`.)
- `Insight/Code/BasicFilters`—basic image processing filters.

- `Insight/Code/IO`—classes that support the reading and writing of data.
- `Insight/Code/Algorithms`—the location of most segmentation and registration algorithms.
- `Insight/Code/SpatialObject`—classes that represent and organize data using spatial relationships (e.g., the leg bone is connected to the hip bone, etc.)
- `Insight/Code/Patented`—any patented algorithms are placed here. Using this code in commercial application requires a patent license.
- `Insight/Code/Local`—an empty directory used by developers and users to experiment with new code.

The `InsightDocuments` module contains the following subdirectories:

- `InsightDocuments/CourseWare`—material related to teaching ITK.
- `InsightDocuments/Developer`—historical documents covering the design and creation of ITK including progress reports and design documents.
- `InsightDocuments/Latex`—LaTeX styles to produce this work as well as other documents.
- `InsightDocuments/Marketing`—marketing flyers and literature used to succinctly describe ITK.
- `InsightDocuments/Papers`—papers related to the many algorithms, data representations, and software tools used in ITK.
- `InsightDocuments/SoftwareGuide`—LaTeX files used to create this guide. (Note that the code found in `Insight/Examples` is used in conjunction with these LaTeX files.)
- `InsightDocuments/Validation`—validation case studies using ITK.
- `InsightDocuments/Web`—the source HTML and other material used to produce the Web pages found at `http://www.itk.org`.

Similar to the `Insight` module, access to the `InsightDocuments` module is also available via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co InsightDocuments
```

The `InsightApplications` module contains large, relatively complex examples of ITK usage. See the web pages at `http://www.itk.org/HTML/Applications.htm` for a description. Some of these applications require GUI toolkits such as Qt and FLTK or other packages such as VTK (*The Visualization Toolkit* `http://www.vtk.org`). Do not attempt to compile and build this module until you have successfully built the core `Insight` module.

Similar to the `Insight` and `InsightDocuments` module, access to the `InsightApplications` module is also available via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d:pserver:anonymous@www.itk.org:/cvsroot/Insight \
  co InsightApplications
```

1.4.5 Documentation

Besides this text, there are other documentation resources that you should be aware of.

Doxygen Documentation. The Doxygen documentation is an essential resource when working with ITK. These extensive Web pages describe in detail every class and method in the system. The documentation also contains inheritance and collaboration diagrams, listing of event invocations, and data members. The documentation is heavily hyper-linked to other classes and to the source code. The Doxygen documentation is available on the companion CD, or on-line at `http://www.itk.org`. Make sure that you have the right documentation for your version of the source code.

Header Files. Each ITK class is implemented with a .h and .cxx/.txx file (.txx 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. (Indeed, Doxygen uses the header documentation to produces its output.)

1.4.6 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/HTML/Data.htm`
`ftp://public.kitware.com/pub/itk/Data/`.

1.5 The Insight Community and Support

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.

- Users may actively report bugs, defects in the system API, and/or submit feature requests. Currently the best way to do this is through the ITK users mailing list.
- Developers may contribute classes or improve existing classes. If you are a developer, you may request permission to join the ITK developers mailing list. Please do so by

sending email to will.schroeder "at" kitware.com. To become a developer you need to demonstrate both a level of competence as well as trustworthiness. You may wish to begin by submitting fixes to the ITK users mailing list.

- 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 at `http://www.kitware.com`. Kitware also offers short ITK courses either at a site of your choice or periodically at Kitware.
- 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. Watch the ITK web pages or check in the `InsightDocuments/CourseWare` directory for more information.

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 Ibanez 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. NLM funding of Insight Toolkit development is continuing through 2003, with additional application and maintenance support anticipated beyond 2003. 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.

CHAPTER

TWO

Getting Started

This chapter will guide you though downloading, installing, and writing the first simple IGSTK program.

2.1 Downloading IGSTK

IGSTK is an open source software toolkit freely available to all individuals and institutions for both academia research and commercial usages. The detailed disclaimer can be found on the following IGSTK copy right page. Please read this page carefully before you proceed.

`http://www.igstk.org/copyright.htm`

There are different ways to obtain the IGSTK source code. You can get the tarball of stable release version for your operating system from the following website:

`http://www.igstk.org/download.htm`

If you want to stay on top of the most recent development of this toolkit, you can get a copy of the source code from the CVS repository. The CVS version will have the latest feature added to the toolkit but is inherently unstable and may contain components in working progress. The source code repository is being hosted by Kitware, The following sections describe how to access the CVS repository.

2.1.1 Instruction on command line CVS client user

The Concurrent Versions System (CVS) is a powerful open-source tool for source code maintenance. It comes standard with nearly all Unix and Unix like systems, MAC OS X, and Cygwin on Windows system. There is no need to install extra software, user should be able to use the CVS command line client. CVSNT is a command line CVS client for windows, you need to install it on the windows machine before you can use it in command window.

1. First, in your terminal, shell, or command line window, *'cd'* into a directory where you want to put the source code.

2. Type the following command:

   ```
   cvs -d :pserver:anonymous@public.kitware.com:/cvsroot/IGSTK login
   ```

   answer by'igstk'

   ```
   cvs -d :pserver:anonymous@public.kitware.com:/cvsroot/IGSTK co IGSTK
   ```

   Now you will have a new directory called `'IGSTK'` in your current directory. you can later *'cd'* into this directory and use the following CVS command to get the latest copy of the source code:

   ```
   cvs update -dAP
   ```

   CAUTION: The CVS version might be unstable, please refer to the nightly dashboard of IGSTK.

   http://public.kitware.com/dashboard.php?name=igstk

   You can browse through the dashboard for the past 15 days too see on which day the dashboard is more stable, no errors, warnings, failing tests, generally speaking, the dashboard looks more 'green'. To get a snapshot of the source code for a specific day, for example 'April 15th, 2006', use the following CVS command:

   ```
   cvs update -D 2006-04-15
   ```

2.1.2 Instruction on Windows GUI CVS client user

There are a number of CVS GUI clients for windows system freely available, such as WinCVS,

Luis: fill out a form ? bookmark the page?. It seems that this paragraph was copied from the web page. We may want to rephrase it for the book. Once you fill out this form you will have access to the download page where two options for obtaining the software will be found. (This page can be book marked to facilitate subsequent visits to the download site without having to complete any form again.) You can get the tarball of a stable release or you can get the development version through CVS. The release version is stable and dependable but may lack the latest features of the toolkit. The CVS version will have the latest additions but is inherently unstable and may contain components with work in progress. The following sections describe the details of each one of these two alternatives.

2.1.3 Downloading Packaged Releases

Luis: and introduction section is missing here. We should talk about the fact that IGSTK depends on ITK and VTK, and optionally on FLTK. Then we can proceed with the explanations on how to download each one of those toolkits.

Please read the `GettingStarted.txt`[1] document first. It will give you an overview of the download and installation processes. Then choose the tarball that better fits your system. The options are `.zip` and `.tgz` files. The first type is better suited for MS-Windows while the second one is the preferred format for UNIX systems.

Once you unzip or untar the file a directory called `Insight` will be created in your disk and you will be ready for starting the configuration process described in Section **??** on page **??**.

2.1.4 Downloading from CVS

The Concurrent Versions System (CVS) is a tool for software version control [5]. Generally only developers should be using CVS, so here we assume that you know what CVS is and how to use it. For more information about CVS please see Section **??** on page **??**. (Note: please make sure that you access the software via CVS only when the ITK Quality Dashboard indicates that the code is stable. Learn more about the Quality Dashboard at **??** on page **??**.)

Access ITK via CVS using the following commands (under UNIX and Cygwin):

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight login (respond with
password "insight")
```

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co Insight
```

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

```
cvs update -d -P
```

`http://www.igstk.org/download.htm`

1. Download periodic stable release.
2. Using CVS to get the latest code and stay updated

2.2 Software Organization

The following sections describe the structure of this toolkit and How to get this software and its documentation and data.

[1]http://www.itk.org/HTML/GettingStarted.txt

Like most other open source projects, the source code of IGSTK is managed and developed by a group of distributed developers. To coordinate this development process and control the quality of this software, we use two centralized CVS repositories to manage the source code, one called *"IGSTK"* and the other called *"IGSTKSandbox"*. *IGSTK* is the main repository for the stable release of the source code, and it might have some minor bug fixes in between releases. *IGSTKSandbox* is the active source code pool. Source code in this repository are under rapid development and might experience lots of refactoring or changes in API.

The standard approach for writing code in IGSTK is as follows: First, based on the requirements for each iteration. We start developing new components/classes in the *IGSTKSandbox*, or copying existing ones from main repository to sandbox for feature extension. After going through cross platform test, code coverage improvement, and code review, the code meets both functional requirements and software engineering standards(platform compatibility, code coverage, and coding style), and reach its maturity. Then the new components/classes will be moved into the man repository, also the extended features will be merged back into the original classes in the main repository. This whole process is called one iteration. The main repository is more rigorous in accepting changes. It requires a bug number from bug tracker (`http://public.kitware.com/Bug/index.php`) before it can accept the code changes. Thus bug should be logged in bug tracker first, before the fix can be committed into the main repository. This two repositories strategy leverage between the need of having a stable and reliable release and the rapid development of the toolkit(For detail on this process, please refer to Chapter 5).

2.2.1 Directory Structure

2.2.2 Documentation

2.2.3 Data

2.2.4 IGSTK Community and Support

The Insight Toolkit is an open-source software system. What this means is that the community of ITK users and developers has great impact on the evolution of the software. Users and developers can make significant contributions to ITK by providing bug reports, bug fixes, tests, new classes, and other feedback. Please feel free to contribute your ideas to the community (the ITK user mailing list is the preferred method; a developer's mailing list is also available).

Mailing lists are used by developers and users to communicate information about the projects. It is strongly recommended that you join the IGSTK mailing list. This is one of the primary resources for guidance and help regarding the use of the toolkit. You can subscribe to the list online at

`http://www.igstk.org/contact.htm`

Currently, two mailing lists exist: the developers and users.

- The users list is open to the public. Use this list to post general questions, postbug reports, or offer suggestions to improve IGSTK. Email: igstk-users@public.kitware.com
- The developers list is for developers. Use this list if you are interested in developing and contributing your own classes. Approval is required to join the list. Email: igstk-developers@public.kitware.com

Both mail lists are archived and searchable through Kitware search page(`http://www.kitware.com/search.html`). This will return hits from the IGSTK mailing list as well a the related projects (CMake, VTK and ITK) mailing lists.

2.2.5 Additional Resources

For more information about Image-Guided Surgery Toolkit (IGSTK), please refer to the following website.

- IGSTK Homepage. `http://www.igstk.org`
- IGSTK Wiki. `http://public.kitware.com/IGSTKWIKI`
- IGSTK Dashboard. `http://public.kitware.com/dashboard.php?name=igstk`
- IGSTK Bug Tracker. `http://public.kitware.com/dashboard.php?name=igstk`
- ...

2.3 Installation

This chapter gives instruction on how to install IGSTK and other libraries IGSTK depends on in your system.

2.3.1 Prerequisite

First, you must download, configure and build three toolkits that are required in order to build IGSTK. These three toolkits are VTK, ITK and FLTK.

- VTK provides visualization functionalities
- ITK provides image processing, segmentation and registration
- FLTK provides Graphical User Interface (GUI) functionalities

It is very important to make sure that you use the appropriate versions of those toolkits since that will ensure that your ge the code is coherent with the current version of IGSTK.

IGSTK relies on other toolkits. This page specifies the versions of those other toolkits that must be used in order to build the current version of IGSTK. *What is the current version of IGSTK? Rlease 7, 8, or 9*

- ITK CVS from January 27, 2006
- VTK 5.0 or CVS version
- FLTK. A snapshot of this toolkit is available on IGSTK wiki papge *How to direct the reader to this complicated link? this zip file with the snapshot of FLTK 1.1 Fltk-1.1-12-16-05.zip*
- CMake 2.2

Note 1: FLTK 1.1 already has a CMakeLists.txt file, therefore it can be configured with CMake.

Note 2: How to obtain ITK through CVS:

```
cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight login answer with
password : insight

cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co Insight cd Insight

cvs update -D 2006-01-27
```

Note 3 : How to obtain VTK 5.0

Downloadable versions of VTK 5.0 are available

http://www.vtk.org/get-software.php#latest

2.3.2 Install IGSTK on Windows System

2.3.3 Install IGSTK on Unix System

2.4 Hello World in IGSTK

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

To add a graphical user interface to the application, we use FLTK. FLTK is a a light weight cross-platform GUI toolkit. FLTK stores a description of an interface in files with extension .fl. The FLTK tool *fluid* takes this file and uses it for generating C++ code in two files. One header file with extension .h, and an implementation file with extension .cxx. In order to use that GUI from the main program of our application we must include the header file generated by fluid. This is done in the following line.

```
#include "HelloWorldGUI.h"
```

The geometrical description of the Cylinder and the Sphere in the scene are managed by SpatialObjects. For this purpose we need the two classes `igstk::EllipsoidObject` and `igstk::CylinderObject`. Their two header files are included below.

```
#include "igstkEllipsoidObject.h"
#include "igstkCylinderObject.h"
```

The visual representation of SpatialObjects in the visualization window is created using SpatialObject Representation classes. Every SpatialObject has one or several representation objects associated with it. We include now the header files of the `igstk::EllipsoidObjectRepresentation` and `igstk::CylinderObjectRepresentation`.

```
#include "igstkEllipsoidObjectRepresentation.h"
#include "igstkCylinderObjectRepresentation.h"
```

As stated above, the tracker in this minimal application is represented by a `igstk::MouseTracker`. This class provides the same interface of a real tracking device but with the convenience of running based on the movement of the mouse in the screen. The header file of this class is included below.

```
#include "igstkMouseTracker.h"
```

Since image guided surgery applications are used in a critical environment, it is quite important to be able to trace the behavior of the application during the intervention. For this purpose IGSTK uses a `igstk::Logger` class and some helpers. The logger is a class that receives messages from IGSTK classes and forward those messages to LoggerOutput classes. Typical logger output classes are the standard output, a file and a popup window. The Logger classes and their helpers are taken from the Insight Toolkit (ITK).

```
#include "itkLogger.h"
#include "itkStdStreamLogOutput.h"
```

We are now ready for writing the code of the actual application. Of couse we start with the classical `main()` function.

```
int main(int , char** )
{
```

The first IGSTK command to be invoked in an application is the one that initialize the parameters of the clock. Timing is critical for all the operations performed in an IGS application. Timing signals make possible to synchronize the operation of different components and to ensure that the scene that is rendered on the screen actually displays a consistent state of the environment on the operating room.

```
igstk::RealTimeClock::Initialize();
```

First, we instantiate the GUI application.

```
HelloWorldGUI * m_GUI = new HelloWorldGUI();
```

Next, we instantiate the ellipsoidal spatial object that we will be attaching to the tracker.

```
igstk::EllipsoidObject::Pointer ellipsoid = igstk::EllipsoidObject::New();
```

The ellipsoid radius can be set to one in all dimensions (X,Y and Z) using the SetRadius member function as follows.

```
ellipsoid->SetRadius(1,1,1);
```

To visualize the ellipsoid spatial object, an object representation class is created and the ellipsoid spatial object is added to it.

```
igstk::EllipsoidObjectRepresentation::Pointer
      ellipsoidRepresentation = igstk::EllipsoidObjectRepresentation::New();
ellipsoidRepresentation->RequestSetEllipsoidObject( ellipsoid );
ellipsoidRepresentation->SetColor(0.0,1.0,0.0);
ellipsoidRepresentation->SetOpacity(1.0);
```

Similarly, a cylinder spatial object and cylinder spatial object representation object are instantiated as follows.

```
igstk::CylinderObject::Pointer cylinder = igstk::CylinderObject::New();
cylinder->SetRadius(0.1);
cylinder->SetHeight(3);

igstk::CylinderObjectRepresentation::Pointer
        cylinderRepresentation = igstk::CylinderObjectRepresentation::New();
cylinderRepresentation->RequestSetCylinderObject( cylinder );
cylinderRepresentation->SetColor(1.0,0.0,0.0);
cylinderRepresentation->SetOpacity(1.0);
```

Next, the spatial objects are added to the view, and the camera position of is reset to observe all objects in the scene.

```
m_GUI->Display->RequestAddObject( ellipsoidRepresentation );
m_GUI->Display->RequestAddObject( cylinderRepresentation );
m_GUI->Display->RequestResetCamera();
m_GUI->Display->Update();
```

Function `RequestEnableInteractions()` allows the user to interactively manipulate (rotate, pan, zoomm etc.) the camera. For `igstk::View2D` class, `vtkInteractorStyleImage` is used; For `igstk::View3D` class, `vtkInteractorStyleTrackballCamera` is used. In IGSTK, the keyboard events are disabled, so it doesn't support the original VTK key-mouse-combined interactions. In summary the mouse events are as follows: Left button click triggers pick event; Left button hold rotates the camera, in `igstk::View2D`, camera direction is always perpendicular to image plane, so there is no rotational movement available for `igstk::View2D`; Middle mouse button pans the camera; Right mouse button dollys the camera.

```
m_GUI->Display->RequestEnableInteractions();
```

The following code instantiate a new mouse tracker and initialize it. The scale factor is just a number to scale down the movement of the tracked object in the scene.

```
igstk::MouseTracker::Pointer tracker = igstk::MouseTracker::New();
tracker->Open();
tracker->Initialize();
tracker->SetScaleFactor( 100.0 );
```

Now we attach previously created spatial object to the tracker and set the tracker to monitor the mouse events from the user interface. The tool port and tool number is naming convention from NDI trackers. *Reference to tracker chapter* object in the scene.

```
const unsigned int toolPort = 0;
const unsigned int toolNumber = 0;
tracker->AttachObjectToTrackerTool( toolPort, toolNumber, ellipsoid );
m_GUI->SetTracker( tracker );
```

Now we setup a logger. We will direct the log output to both the standard output (std::cout) and a file (log.txt). *need reference to logger chapter about priority level*

```
itk::Logger::Pointer logger = itk::Logger::New();
itk::StdStreamLogOutput::Pointer logOutput = itk::StdStreamLogOutput::New();
itk::StdStreamLogOutput::Pointer fileOutput = itk::StdStreamLogOutput::New();

logOutput->SetStream( std::cout );
logger->AddLogOutput( logOutput );
logger->SetPriorityLevel( itk::Logger::DEBUG );

std::ofstream ofs( "log.txt" );
fileOutput->SetStream( ofs );
logger->AddLogOutput( fileOutput );
```

By connecting the logger to the Display and the Tracker, messages from the these components will be redirected to the logger.

```
m_GUI->Display->SetLogger( logger );
tracker->SetLogger( logger );
```

Next, we set the refresh frequency of the display window. After we call the `RequestStart()` function, the pulse generator inside the display window will start ticking, and call the display to update itself 60 times per second.

```
m_GUI->Display->RequestSetRefreshRate( 60 );
m_GUI->Display->RequestStart();
```

All application should includes the following code. This is the main event loop of the application. First it checks if the application is aborted by user, if not, it calls for the `igstk::PulseGenerator` to check its time out.

```
while( !m_GUI->HasQuitted() )
  {
  Fl::wait(0.001);
  igstk::PulseGenerator::CheckTimeouts();
  }
```

Finally, before exiting the application, the tracker is properly closed and other clean up procedures are executed.

```
tracker->StopTracking();
tracker->Close();
delete m_GUI;
ofs.close();
return EXIT_SUCCESS;
```

Add the application wizard here?

CHAPTER

THREE

Architecture

K2 General Comments: - Lots of spelling errors, ran it through a spellcheck - Several grammatical errors, tried to fix obvious ones - Uses passive voice a lot, only changed a few of these - I have put several comment blocks prefixed with K2 throughout. - My main general comments are 1) some things are talked about or referred to before concept is explained (this chapter is early in the book), and 2) I get a lot of information/lecture on the importance of safety-driven programming, but an unlevel explanation of IGSTK's architecture and how it addresses these issues.

3.1 General Background

> *"As to diseases, make a habit of two things - to help, or at least, to do no harm."*
> Hippocrates, Epidemics Book 1, Section XI.

The fundamental characteristic of the IGSTK toolkit is that it is software intended to be used in the operating room. Applications using IGSTK will provide informative graphic displays to a clinician with the purpose of facilitating the execution of a surgical procedure. In that context, the main consideration driving the design of the IGSTK toolkit is to do as much as possible to protect the patient from harm.

Safety was the first consideration during the design, implementation and testing decisions of the toolkit. Whenever the development team found software requirements conflicting with patient safety this latter took precedence.

The IGSTK architecture is designed to try to take advantage of every possible software mechanism that could prevent errors from happening when the application were running in the surgery room. These mechanisms include

- Defensive Programming
- Safety by Design
- Software Verification

- High Code coverage
- State Machine Programming

Some of these techniques, such as Defensive Programming and High Code Coverage, are closer to the actual software writing process while the others are closer to the high level design process. Among these techniques, the use of State Machines stands out as the primary mechanism by which to ensure patient safety. The reasons for using State Machines in critical application design is that they offer the opportunity for performing formal validation of the software, they make possible to guarantee that the application is never in an error state, and they make explicit their behavior face to any interactions with the user or with other pieces of software. State machines are an accepted pattern in software design for safety-critical applications, in particular real-time systems and embedded applications [2, 3].

Implementing the software as a collection of components is another architectural decision motivated by the commitment to reduce or eliminate patient risk. When software is written in small components it is easier to fully verify the behavior of each component by testing it throughly. Every component was designed with a very compact Application Programmer Interface (API) that prevented the introduction of arbitrary features and kept the complexity of the software at a level where it can be throughly tested.

One of the postulates held during the design process is that in the context of safety-critical applications

> *Freedom is dangerous,*
> *Flexibility is bad,*
> *Generality brings risk.*

Although this may sound like a statement from George Orwell's novel "1984", it is indeed critical for raising the quality of the software, facilitating maintenance, making possible to have 100% code coverage and preventing the complexity of the software from increasing to the point where it can not be entirely tested.

Feature creep is a well known disease affecting most software projects, from the library level to the end user application level. Developers can rarely resist the temptation of adding a cool feature to the software, sometimes for as little justification as because the opportunity presented itself at their fingertips. Developers of toolkits tend to do as much as possible for offering options to application developers. This tendency has to be reversed in the context of a safety-critical software because every new feature added to the code results in a combinatorial explosion of states that can never be tested nor documented. Of course, a well controlled software development process can also prevent the insertion of arbitrary features by enforcing the discussion of requirements, code reviews and traceability. The software development practices followed in IGSTK are described in detail in Chapter 5.

Further motivation for the emphasis on safety came from the development team's awareness of the fact that clinical conditions are already tense, even before software is introduced in that environment. Adding the risk of software to an already unstable clinical situation is something that must be done with careful consideration. The following two sections provide some of the background of what is publicly accepted as the evidence of continuous danger in a clinical environment, and what is know about the frailties and vulnerabilities of software development endeavors.

3.2 Medical Errors

K2 You have two ways of citing the report - with a URL in a footnote and as a citation. Which is it? Luis: Good point. The footnote has now been removed.

The following quote was taken from the report *"To Err is Human"* prepared by the U.S. Institute of Medicine[1] in 2001 [10].

> *Two large studies, one conducted in Colorado and Utah and the other in New York, found that adverse events occurred in 2.9 and 3.7 percent of hospitalizations, respectively. In Colorado and Utah hospitals, 6.6 percent of adverse events led to death, as compared with 13.6 percent in New York hospitals. In both of these studies, over half of these adverse events resulted from medical errors and could have been prevented.*
>
> *When extrapolated to the over 33.6 million admissions to U.S. hospitals in 1997, the results of the study in Colorado and Utah imply that at least 44,000 Americans die each year as a result of medical errors. The results of the New York Study suggest the number may be as high as 98,000. Even when using the lower estimate, deaths due to medical errors exceed the number attributable to the 8th-leading cause of death. More people die in a given year as a result of medical errors than from motor vehicle accidents (43,458), breast cancer (42,297), or AIDS (16,516).*

This report is a striking revelation of how much risk is inherent to the practice of health care delivery. The report is particularly unsettling when considered under the light that those statistics were gathered only from the medical errors that are *officially reported*. The numbers do not include, of course, the medical error that are not reported to hospital officials, and much less, those errors that go unnoticed by nurses and medical practitioners.

The report also refers to the economic cost of medical errors

> *Total national costs (lost income, lost household production, disability, health care costs) are estimated to be between $37.6 billion and $50 billion for adverse events and between $17 billion and $29 billion for preventable adverse events. Health care costs account for over one-half of the total costs. Even when using the lower estimates, the total national costs associated with adverse events and preventable adverse events represent approximately 4 percent and 2 percent, respectively, of national health expenditures in 1996. In 1992, the direct and indirect costs of*

[1] www.iom.edu

> *adverse events were slightly higher than the direct and indirect costs of caring for people with HIV and AIDS.*

The notion that medical errors are expensive in human, social and economical terms was kept in mind during the design and development process of the toolkit. It became relevant every time that arguments were raised in favor of relaxing the safety rules of the toolkit with the purpose of making easier or more convenient the work of developers. In such instances it was useful to remember that the cost of toolkit and application developers is insignificant compared to the tragic consequences that a potential adverse event may have in a patient when produced in the operating room as a consequence of a software deficiency.

In the analysis of "*Why errors happen?*" the report emphasizes the following fact

> *When large systems fail, it is due to multiple faults that occur together in an unanticipated interaction, creating a chain of events in which the faults grow and evolve. Their accumulation results in an accident. "An accident is an event that involves damage to a defined system that disrupts the ongoing or future output of that system."*

This passage of the report provided background motivation for developing IGSTK as a toolkit with minimal functionalities and with well defined interactions between components. When compared to other toolkits such as VTK and ITK, the application developer will notice that IGSTK is very compact and that it provides a very restricted set of functionalities. Such characteristics, that could have been perceived as a weakness in a general software toolkit, are indeed the strength of IGSTK because they make the toolkit safe and reliable enough for being used in a clinical setting.

The dreadful consequences of medical errors and their high rate of occurrence are important for understanding the architectural design decisions made in the toolkit as described in the rest of this chapter.

The lessons to remember from this section are

- Medical errors are very common.
- Medical errors are irreversible.
- Medical errors are expensive.
- Good design practices help to prevent medical errors.

3.3 Layered Architecture

Figure 3.1 illustrates the layered architecture of IGSTK starting from the operating system layer as a foundation. The Insight Toolkit ITK is used for providing all the image analysis functionalities in IGSTK as well as a good portion of the infrastructure classes. The Visualization Toolkit

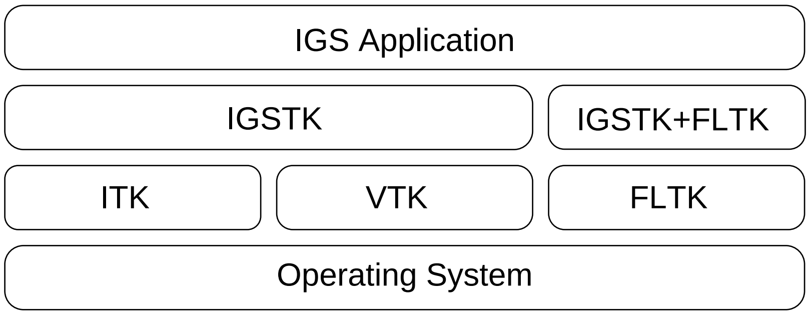


Figure 3.1: IGSTK Layered Architecture.

is used for supporting the display of image an geometrical models as well as managing a good portion of the user interaction. The Fast Light Toolkit FLTK is optionally used for proving GUI functionalities. Although most of the examples in this book are based on FLTK, it is possible to use IGSTK with other GUI libraries such as Qt and MFC. IGSTK internally makes use of ITK and VTK classes but does not expose any of them in its API[2].

Image Guided Surgery applications will be built on the top of IGSTK. The application code can only have access to IGSTK classes, not to the ITK or VTK classes that are used underneath. The purpose of this encapsulation of toolkit functionalities is to improve the safety of the final application by preventing developer to directly manipulate objects without passing first through the many safeguards that have been included in IGSTK.

3.4 Software Quality

The practice of software development and software engineering is farther from being a science than what most of us would like to think. Unfortunately, software design and development involves such high levels of complexity that most of it is still an art rather than a science. This section illustrates how fragile and error prone is the process of software development, and what good practices may help to make it safer and more reliable.

K2 I have several comments on this quality section. - With it being back-to-back with the previous section, we have several consecutive pages of being almost ”preachy” to the reader, instead of telling them about architecture. - It implies up above (correctly) that better process yields better quality. If so, may some of this motivation should be in the process chapter? - the first subsection on ”Software Quality Statistics” equates quality with testing, which is accepted in software engineering as a failed endeavor. The accepted axiom is ”the more defects you find and fix, the more that remain that you do not know about”. Despite our aggressive unit testing, I would not want to imply that quality equals testing. We do a number of other activities - code reviews, requirements review every week, requirements management, clinician feedback,

[2] Application Programmer Interface

coding standards, etc. that do as much if not more to ensure quality. - I think the combination of this section and the previous section overstate the situation of modern software development as being in dire straights. True, there is a lot of bad code out there, and often end-users accept it as a cost of doing business, and we can say that. But data show that the general trend in software intensive systems, and especially in mission critical systems, is toward a declining defect density in delivered source code.

3.5 The Main Components

The main components of the toolkit are the following

- `igstk::Tracker`
- `igstk::View`
- `igstk::SpatialObject`
- `igstk::SpatialObjectRepresentation`
- `igstk::ImageReader`

Some of the main components have derived classes for managing specific implementation issues. In those cases the base class is implemented as an abstract class, which means that it is not intended to be instantiated by the application developer. Instead, the developer must use one of the derived classes. This is done as part of the safetybydesign approach in IGSTK. The idea is that derived classes allow to further restrict the expected behavior of a particular class, and in that way preclude the misuse of the class by adding specific tests that ensure that the interaction of the class with other pieces of software are limited to a well defined set of use cases. A typical example of this implementation of safetybydesign is the family of the `igstk::ImageReader` classes. Where an abstract `igstk::DICOMImageReader` class implements the basic actions required for reading any DICOM files, while its derived classes the `igstk::CTImageReader` and `igstk::MRImageReader` deal with specific modalities. Since these classes know what modality they are expecting, they can perform extra verification at the moment of reading an image. In the event that the image provided to the reader do not match the expected modality, then the reader will report the failure of the action and it will remain in a safe state declaring that no image has been read. The DICOMImageReader class can not be instantiated by an application developer, so there is no way of creating *Generic* readers that will introduce **ambiguity**, and therefore **uncertainty**, and therefore **risk** into the application.

K2 presenting a long laundry list of class names does not provide the reader with any information as to the purpose and relationships of the classes. Our other papers have a component diagram. Then within the component diagram you could have class diagrams that describe the classes that realize that component's behavior. In short, the bulleted lists in this section would be better served with diagrams.

Other classes for supporting infrastructure include

- `igstk::RealTimeClock`
- `igstk::PulseGenerator`
- `igstk::TimeStamp`
- `igstk::Transform`
- `igstk::StateMachine`
- `igstk::StateMachineInput`
- `igstk::StateMachineState`
- `igstk::SerialCommunication`

The `igstk::DICOMImageReader` is further specialized in the two classes

- `igstk::CTImageReader`
- `igstk::MRImageReader`

The family of SpatialObjects form the hierarchy

- `igstk::SpatialObject`
- `igstk::GroupObject`
- `igstk::AxesObject`
- `igstk::BoxObject`
- `igstk::ConeObject`
- `igstk::CylinderObject`
- `igstk::EllipsoidObject`
- `igstk::ImageSpatialObject`
- `igstk::CTImageSpatialObject`
- `igstk::MRImageSpatialObject`
- `igstk::USImageSpatialObject`
- `igstk::MeshObject`
- `igstk::TubeObject`
- `igstk::TubeGroupObject`

- igstk::UltrasoundProbeObject

These classes have their corresponding Representation Objects

- igstk::ObjectRepresentation
- igstk::AxesObjectRepresentation
- igstk::BoxObjectRepresentation
- igstk::ConeObjectRepresentation
- igstk::CylinderObjectRepresentation
- igstk::EllipsoidObjectRepresentation
- igstk::ImageSpatialObjectRepresentation
- igstk::CTImageSpatialObjectRepresentation
- igstk::MRImageSpatialObjectRepresentation
- igstk::USImageSpatialObjectRepresentation
- igstk::MeshObjectRepresentation
- igstk::TubeObjectRepresentation
- igstk::UltrasoundProbeObjectRepresentation

K2 A diagram appears in my PDF right here of a Timing Collaboration, which apparently goes with the next section. It appears before any reference or discussion.

3.6 Timing

In action, timing is everything.
Force doesn't matter.
Weight doesn't matter.
Even being morally right does not matter.
All that matters is timing.
Deng Ming-Dao,Everyday Tao

The primary use of IGSTK in a surgical application will involve presenting information to the surgeon in the form of a graphical display. This display will typically include some elements that are physically visible to the surgeon in the operating room, combined with other elements that are only visible in the display. The surgeon will make a decision and take actions based on the relative position of those elements as presented in the display. It is therefore of the utmost

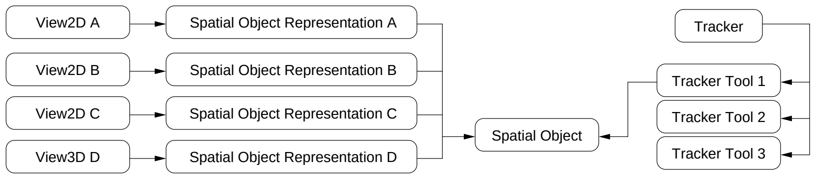

View2D A
View2D B
View2D C
View3D D
Spatial Object Representation A
Spatial Object Representation B
Spatial Object Representation C
Spatial Object Representation D
Spatial Object
Tracker
Tracker Tool 1
Tracker Tool 2
Tracker Tool 3

Figure 3.2: Timing Collaboration between the main IGSTK components.

importance that the positions and orientations of all the objects in the display correspond to a consistent view in time of the surgical scenario.

When a display is presented to the surgeon, it is implicitly claiming that this is the view of the operating room at a very recent time. Since many of the objects in the scene are in continuous movement, whether because they are inside of the patient or because the surgeon is controlling them, the accuracy of the position is related to the consistency of time for each object. In other words, when the graphic display shows where the surgical needle was located at 9:06 am, it should appear along with the location of the patient's liver at 9:06am. The toolkit design takes measures for preventing the accidental display of the position of the needle where it was at 9:06am along with the patient's liver where it was at 8:54am. Synchronicity of the objects in the scene is extremely important because it is from their relative position that the surgeon will derive the most useful information for proceeding with the intervention.

The collaboration between the main component of the toolkit is illustrated in Figure 3.2. Tracker and View classes have their own pulse generators that will keep them updating at a rate specified by the application developer. A Tracker device can usually drive multiple tracked tools, that in IGSTK are managed by the `igstk::TrackerTool` class. Once information about Transform updates is passed from the Tracker into the TrackerTool, it is then pushed into the SpatialObject. One SpatialObject can be attached to one an only one TrackerTool. A TrackerTool can only be attached to a single SpatialObject.

K2 The push and pull semantics of Spatial Objects and Tracker Tools does not match between the TimingCollaboration diagram and the TimingArchitectureSequenceDiagram. In the former the tool pushes the transform to the SO, and the text describes this. In the latter, the SO pulls information from the tool

Also, the sequence diagram is very awkward because it implies a linear sequence of events between the tracker operations and the left-side. These things actually happen in 2 threads, one internal to the tracker - so in actual time they may be interleaved. In practice you would expect the right-side (tracker) to update more times that the left-side (view). Threads are tricky to convey in UML sequence diagrams, but it can be done.

Also, the syntax ”xxx : class” in a box on the sequence diagram means there is a object whose unique identity is ”xxx” of type ”class”. Therefore you should not have two boxes with ”id: PulseGenerator”. In fact, you should not all of these objects using ”id:”. I would just remove ”id” and use ”:View”, ”:SpatialObject” etc., but use ”viewPG : PulseGenerator” and ”trackerPG : PulseGenerator” for the two Pulse Generator objects in the diagram.

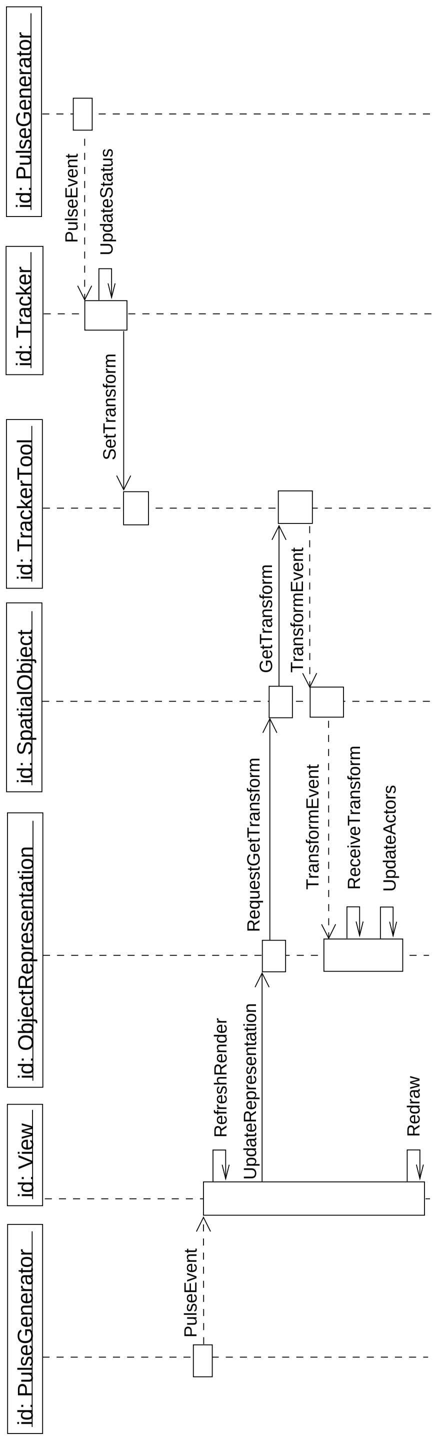


Figure 3.3: Timing Collaboration between the main IGSTK components.

K2 The explanation in the next few paragraphs is pretty detailed and describes things using concepts (events, observers) not presented yet. I wonder if this would work better as an example in a later chapter

A UML sequence diagram of the timing interactions is presented in Figure 3.3. The diagram can be better understood by dividing it in two sections. The division line must be drawn in the SpatialObject class. The section to the right of the SpatialObject class describes the flow of information and events from the PulseGenerator class that is associated with a Tracker, up to the actual SpatialObject class. The direction of information flow in this section of the diagram is from right to left, starting with the PulseGenerator class at the right side of the diagram. The Tracker internally has its own PulseGenerator whose rate for generating pulses is set by the application developer. At every pulse, the Tracker class is going to query the actual hardware tracker device and will get from it information about the position of the tracked instruments in the operating room. Since most tracker devices can track multiple instruments, the Tracker class uses the TrackerTool classes as the helpers that are associated to each one of the physical tracked surgical instruments. When the Tracker class received the updated information from the hardware, it stores the new Transforms, that describe the most recent position of the instruments, into their corresponding TrackerTool objects.

Communication between the Tracker object and its associated PulseGenerator is performed through Events and Observers. The PulseGenerator produces PulseEvents at the rate selected by the application developer. The Tracker class has an internal Observer that is connected to the PulseGenerator. When the Observer receives a PulseEvent, it invokes the method `UpdateStatus()` in the Tracker. This method queries the hardware, recovers the most recent transformations and then push them into the corresponding TrackerTool classes by invoking their `SetTransform()` method. At that point, the TrackerTool is updated and the refreshing cycle of the Tracker concludes.

DG: The second-to-last sentence should be rephrased: "This method recovers the most recent transformations sent by the tracking hardware and then pushes them into the corresponding TrackerTool objects by invoking their `SetTransform()` methods." The reason for this rephrasing is that the actual querying of the hardware is done in the tracker's extra thread (the pushing of the transforms from the tracker buffer to the TrackerTools still occurs in the main thread).

The Transforms that are passed from the Tracker class, all the way up to the TrackerTool are marked with a TimeStamp that indicates at what time the Transform started to be valid, and at what time the Transform will expire. Since the Tracker provides a periodic flow of Transforms, the expiration mechanism allows other components downstream to know when a new Transform should be available and when to stop using an old transform. A very similar mechanism is used in the TCP/IP protocol in order to drop packages once they have completed a number of redirections on the network. In networking lingo this is known as the "Time to Live" or TTL.

If a SpatialObjectRepresentation finds that the Transform of its associated object has expired by the time it needs to render its appearance in the scene, then it can decide to not show that particular instrument, or to flag it in blinking mode, or in a special color. The purpose of this change in representation will be to warn the surgeon about the fact that the current position of that object is not known at this point. Such an event may be the consequence of someone blocking the line of sight of an optical tracker, or an accidental disconnect of a physical tracking tool.

The section of the diagram at the right side of the TrackerTool represents another cycle of information refreshing, this time in two stages. First, request for updating information are moved from left to right, starting in the PulseGenerator associated to the View class, and moving towards the TrackerTool class. Second, the information flows from right to left by carrying the current Transform known by the TrackerTool up to the SpatialObjectRepresentation class.

If we follow these two flows we will encounter the following details. The View class has its own PulseGenerator that is set to produce a particular rendering rate by the application developer. At every pulse of the PulseGenerator, a PulseEvent is sent to an Observer inside the View class. The callback of this Observer invokes the method `RefreshRender()` in the View class. This method visits all the SpatialObjectRepresentation classes that have been registered with this View and invokes on each of them the methods `UpdateRepresentation()` and `UpdatePosition()`. Note that only the cycle of the `UpdatePosition()` method is presented in this diagram for the sake of simplicity. When the UpdatePosition method is invoked in the SpatialObjectRepresentation object, it triggers a call to the `RequestGetTransform()` method on the SpatialObject. This in its turn triggers a call to the `GetTransform()` method of the TrackerTool object that has been associated with this particular SpatialObject. In response to this invocation, the TrackerTool object invokes a TransformModified event for which the SpatialObject has already connected an internal Observer. The reception of the TransformEvent in the SpatialObject triggers the `ReceiveTransform()` method, that then calls the `UpdateActors` method. This last one is the method that decided how to modify the graphical representation of the objects as presented to the surgeon, in order to indicate whether the current position of the object in the surgical scene is valid or not. Once the View class has finished triggering the refresh of each one of its registered SpatialObjectRepresentation objects, it will proceed to actually redraw the scene by triggering a redraw of its internal FLTK window. This redraw will trigger a Render action on the VTK pipelines that is maintained inside the View and the SpatialObjectRepresentation classes.

The cycle of updates in the right side of the diagram happen in an independent thread that is created for the Tracker class. In this way, the rate of the Tracker updates and the refreshing rate of the View can be maintained in a decoupled fashion. This of course is only possible as long as the refreshing rates for both components are far from the maximum possible refresh rate. If the application developer selects refreshing rates that are too high, then aliasing effects will happen in the temporal domain between the cycle driven by the PulseGenerator of the Tracker (right side of the diagram) and the cycle driven by the PulseGenerator of the View class (left side of the diagram). *K2 Tracker above should be TackerTool ?*

DG: The updates in the right side of the diagram do not occur in a different thread: they occur in the same thread as all other IGSTK events. The part of the Tracker class that accepts pulses from the PulseGenerator runs in the main IGSTK thread (in fact it has to, since the PulseGenerator events are always sent from the main IGSTK thread). When a Pulse occurs, the Tracker will extract data for a particular time point from its internal data buffer and push that data to the tools, and the only role of the tracker's extra thread is keep the data in the buffer up-to-date by

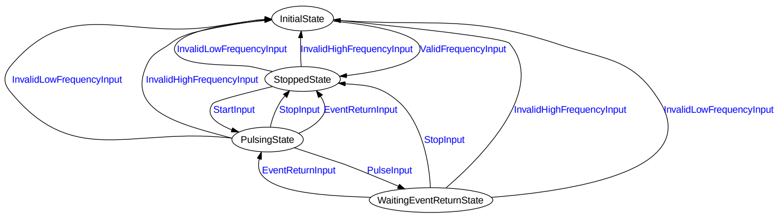


Figure 3.4: State Machine Diagram of the PulseGenerator class.

listening on the communication link for new data from the tracking system and placing the new data into the buffer. This extra thread does not utilize the PulseGenerator.

The Timing infrastructure is based on providing self-contained behavior to the individual components of the toolkit. This prevents the need for a master class for controlling all the activities at the top level of the application.

3.6.1 Pulse Generator Implementation

Figure 3.4 illustrates the State Machine of the `igstk::PulseGenerator` class.

K2 This figure appears before any reference to it. Also, no discussion is provided whatsoever

3.7 State Machine Architecture

K2 We have to be careful - we use the term formal validation at least twice in this chapter, yet we do not do it nor do we have any intention of doing it.

I have not checked yet, but I hope this section dovetails with the later detailed section on state machines in a later chapter.

State Machines make it possible to introduce determinism and formal validation into the software infrastructure of the toolkit.

3.7.1 Safe States

The approach following in IGSTK is the one for “walking in a mined field”. This means that before executing every operation, IGSTK verifies that the operation can be completed successfully. In this way, no component is ever placed in an error state. When one component ”Requests” another to do something, the second component goes into an ”Attempting” state and from there it triggers all the functions that will verify that such request can be satisfied without going into an error state. If any error condition is found, then an error-notifying event is sent from the

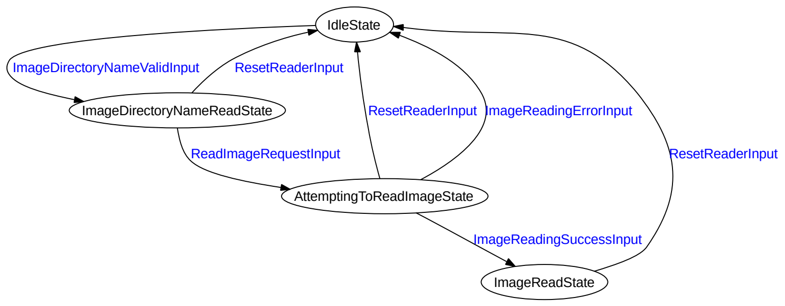


Figure 3.5: Example of the State Machine Attempting Pattern in the DICOMImageReader component.

second component. If, on the other hand, the operation can be completed successfully, then an Event with a positive notification is sent from the second component.

Figure 3.5 illustrates the use of this pattern of interaction in the context of the `igstk::DICOMImageReader`.

3.7.2 Public versus Private API

In order to enforce safety-by-design, the only methods that are public are those that ”Request” operations into the components. In that context, there is no way to force a component to do anything. Every component is kindly asked to “try” to do something, with the understanding that the request may be satisfied or denied according to the current state of the StateMachine that drives such component.

3.7.3 Communication Protocols

Never return data whose value is to be checked for validity. The old FORTRAN style of invoking a function and on its returned value check whether its operation was successful or not is a poor scheme for safety-critical applications. In IGSTK a more secure protocol for notification of success or failure of queries was implemented. In summary what this protocol defines is that all information returned as the answer to a query is to be passed in the form of Events with payloads. When the response to a query for data is that the data is not available or that the data is not valid then the response to the query is sent in the form of a specific event that encodes that error condition. When, on the other hand, the information is valid, then the response is sent in the form of an event with payload, where the payload is the object that carries the information requested.

Events, Inputs and Transduction

When information is being passed back from the components that provide services to the components that requested such services, the data is encapsulated in the form of an event with payload. The component that requested the information should have an internal Observer expecting such an event in order to translate it into an Input code for its internal State Machine. The process of converting Events into StateMachine Inputs is called here “Transduction”. Since the process of Event to Input transduction is almost the same, a set of helper Macros were defined in the toolkit in order to facilitate development and to enforce uniformity on the source code.

A typical TransductionMacro expects the toolkit developer to define the type of the Event to be received and the type of the Input to be passed to the State Machine as a result.

3.7.4 Helper Macros

- igstkDeclareInputMacro
- igstkDeclareStateMacro
- igstkAddInputMacro
- igstkAddStateMacro
- igstkAddTransitionMacro

[6] [1] [2] [3]

CHAPTER

FOUR

Requirements

4.1 Introduction

The degree of expertise required to develop effective applications in the image-guided surgery domain is significant. As a result, development processes require a close connection between subject matter experts (SMEs) and the software engineers/developers. Waterfall and spiral development approaches tend to incorporate “top-down” processes that assume that complete requirements can be defined early in development. However, in this domain, we have discovered that development processes must more tightly integrate the iterative involvement of medical professionals and SMEs. As such, agile development approaches are more inline with this domain. Approaches to requirements engineering and management are not well-defined within standard agile development venues. In our work, we introduce a customized extension to standard agile development techniques through the introduction of an *agile* requirements management approach.

4.2 What is an IGSTK Requirement

Requirements typically define system constraints with an emphasis on the functionality that affects the end users. Since IGSTK is a component-based toolkit, there are three types of end users, *medical software developers* who use the components to compose new applications, *radiologists* who will use applications derived from IGSTK, and *software developers* that will contribute components to the IGSTK framework. In all cases, requirements must meet the rigor necessary to assure the reliability of the target applications.

4.2.1 Types of Requirements

IGSTK requirements can be classified into three areas that represent the three different end users of the system, although it is inevitable that some overlap will exist. These three types of requirements, shown in Figure 4.1, are *architecture requirements*, *style guidelines requirements*,

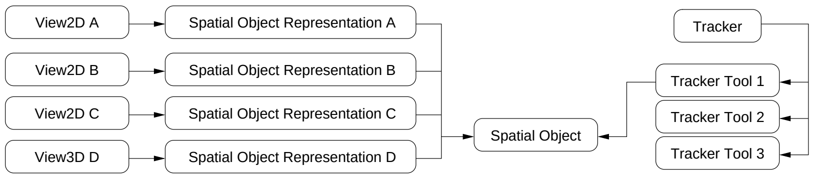

View2D A
Spatial Object Representation A
View2D B
Spatial Object Representation B
View2D C
Spatial Object Representation C
View3D D
Spatial Object Representation D
Spatial Object
Tracker
Tracker Tool 1
Tracker Tool 2
Tracker Tool 3

Figure 4.1: Classification of Requirements in IGSTK.

and *application requirements*, respectively. Architecture requirements define how the new systems should be composed from IGSTK components. Style guidelines include the constraints on the developers who develop new software for IGSTK. Application requirements are standard requirements that define how the target system will execute with respect to its used in the medical environment. These classifications were derived from earlier work which defines a product line development process called Component-Based Product Line Analysis and Development (C-PLAD) [REF????]. Interestingly, as applications are built from the underlying components, the application requirements extend the component-level requirements. The classification of requirements and their relationship to other types of requirements are illustrated in Figure 4.1 Figure 1.

4.2.2 Requirements Hierarchy

Requirements are classified by several image-guide surgery-related areas. The hierarchy by which requirements are classified is shown in Table 4.1. All requirements sections are split into two parts, functional requirements and nonfunctional requirements *Brian: hmmm may remove the notion of functional/nonfunctional, I think we mixed this up a bit*.

4.2.3 Snapshot of Requirements at Publication Time

Considering the fact that IGSTK is an open-source toolkit, requirements are ever-evolving. *Luis: being Open Source is not related to the fact the requirements are always evolving. The changing nature of requirements is mostly related to the fact that IGSTK is a toolkit and therefore intended to be a service layer. The dynamic nature of a toolkit is the result that users may apply the toolkit to different applications and every new application brings new requirements in the form of desirable new features, or modification of existing features. A closed source toolkit would have the same evolving pressure, the difference with Open Source is that the public will notice the changes in the code, while in close source, users do not know what is changing in the software and have no way of verifying its correctness*. Table 2 shows the requirements defined for IGSTK at the time of the publication of this book in outline format. The purpose of this chapter is describe the notion of requirements and how to access them; therefore we do not describe each requirement in detail. *Brian: IF we keep this, then probably should place this in an appendix*

Table 4.1: Requirements Hierarchy

1.	Interface Requirements
2.	User Requirements
3.	Infrastructure Requirements
3.1	Performance
3.2	Testing and Evaluation
3.3	Implementation and Transition
3.4	Quality Assurance and Reliability
3.5	Configuration Management
3.6	Communication
3.7	Logging
3.8	Timing/Latency/Calibration
3.9	Error-handling Notification
4.	Tracker Requirements
5.	Viewer/ Image IO
6.	Spatial Objects
7.	Landmark Registration Component
8.	State Machine

4.2.4 Lightweight Requirements Management Process

Defined Requirements Management Process

The process for requirements management is significantly integrated with code management. Although IGSTK development has not been a pure agile process, as discussed above, our project does see requirements for development as coming from the “bottom-up”. Developers introduce new requirements for further capabilities as components are being developed. The IGSTK project has employed a new collaborative process for reviewing, implementing, validating, and archiving these requirements, and it is integrated with application development. This process is illustrated as a UML state diagram in Figure 2.

In the initial requirements phase, general requirements for tracking devices (localizers) were discovered. As the components for these initial requirements were developed, we discovered that additional requirements existed (i.e. perhaps specific validation requirements). Once a developer identifies new potential requirements (Conceptualized box in Figure 1), the developer will post a text description (Defined) on the shared web site (Wiki). At the same time, the initial code that fulfills the requirements is entered into a sandbox repository (i.e. a development configuration management area). The requirement would then undergo an iterative review sub-process where the team members would review, discuss, and potentially modify the requirement. Based on the teams decision, the requirements can be rejected/aborted or accepted. Rejected requirements are archived on the Wiki (Logged) so that they can be reopened later, if necessary. A unique approach in the IGSTK project is the use of an on-line open source bug

Table 4.2: Snapshot of IGSTK Requirements.

1.	Interface Requirements
1.1.1	IGSTK components shall be limited to 8 exposed methods.
1.1.2	Interactions with IGSTK components that require more than request/response protocol must be explicitly documented in the component API specification.
2.	High-Level User Requirements.
2.1	Development Environment-Specific.
2.1.1	IGSTK components shall execute a high-level task autonomously requiring minimal interaction.
2.1.2	IGSTK components shall not expose underlying class structure in header files.
2.2	Application-Specific
2.2.1	At a minimum, IGSTK shall support general components for tracker capabilities, viewer capabilities, segmentation, and registration.
2.2.2	The IGSTK platform shall include a minimum of two application examples.
3.	Infrastructure and Nonfunctional Requirements
3.1	Performance Requirements
3.1.1	IGSTK components that manage events shall include threading capability.
3.2	Test and Evaluation Requirements
3.2.1	A test case shall be included for each IGSTK component.
3.2.2	Each IGSTK components must have 100% testing coverage (explain this).
3.3	Implementation and Transition Requirements
3.3.1	IGSTK components shall enable compilation via CMake.
3.3.2	Each IGSTK components shall be documented with corresponding Doxygen files
3.4	Quality Assurance and Reliability Requirements
3.4.1	Each IGSTK component must be controlled by the IGSTK internal state machine code.
3.5	Configuration Management Requirements
3.5.1	Each configuration management action must be supported by a requirement number.
3.6	Communication Requirements
3.6.1	The IGSTK platform shall contain a general communication component that acts as a proxy to all communication ports to and from the application such as the serial/parallel ports or wireless connections.
3.7	Logging Requirements
3.7.1	The IGSTK platform shall contain a general, centralized logging capability that can be used for all IGSTK components.
3.7.2	The logging capability shall support timestamps for the audited information.
3.7.3	The logging capability shall enable logging for multiple components concurrently and support auditing into multiple files.
3.8	Timing and Latency Requirements
3.8.1	Add synchronization requirements.
3.9	Error-Handling and Notification Requirements
3.9.1	Add error-handling and notification requirements.
4.	Tracker Requirements
4.1	Functional Requirements
4.1.1	The tracker component shall have the ability to initialize the tracker hard-

tracker (PHP BugTracker) to store requirements. This approach is particularly effective as defect reports and resulting actions are also stored in the bug tracker. The accepted requirements are entered into the bug tracker and marked as “open”. Once the supporting software is implemented and its functionality is confirmed, the requirement is marked as “verified”. As the nightly builds takes place, all verified requirements are automatically extracted into Latex and PDF files, and are archived. Custom scripts were developed for this purpose.

Figure 2. The Requirements Management Process as a UML state diagram.

Conceptualizing Requirements through Activity Modeling

A barrier to conceptualizing requirements is the disparity of knowledge between software engineers and medical professionals. These two groups tend to speak in totally different terms. In IGSTK, the best way, that was determined for software engineers to collaborate with radiologists, was by considering scenarios that describe the application of the system. Table 1 shows a sample sample scenario of the use of IGSTK applications for guidewire placement. A stepwise, temporal review of this scenario is illustrated in Figure 3.

Table 1. Sample Scenario for Guideware Placement.

The activity diagram view is an effective collaboration medium for the radiologists and the software engineers. Although the activity diagram explains one particular application, requirements were conceived for many components included in IGSTK (e.g. tracking, visualization, and registration). In additions, requirements can be inferred from this activity diagram for all three requirements classifications as defined in the earlier section.

Figure 3. Activity Diagram Illustrating Guidewire Tracking Scenario.

Integrating Requirements Management and Bug Tracking

The lightweight requirements management process follows two scenarios. One scenario follows a formal approach for introducing new software whereas the second scenario is inline with quick changes or modifications. Step-by-step details of the two processes are shown in Table 3 and Table 4. *Brian: Add more description*

Accessing and Contributing to IGSTK Requirements

Developers wishing to enhance the core IGSTK framework may discover a new system constraint that is not previously defined. IGSTK encourages developers to follow the requirements management process and identify the requirement for a new software change first. Secondly, the new software should be associated to the newly discovered requirement. As a first step the developer should be familiar with the current requirements, because perhaps the change will help to modify the software to better represent that current requirement. In this case, the new software should reference the existing requirement.

Table 4.3: Sample Scenario for Guideware Placement

1.	Interventional radiologist (IVR) positions fiducials on patient
2.	IVR uses CT or MRI imaging to obtain a digital representation of the patient
3.	IVR initializes image-guided surgery (IGS) software application.
4.	IVR loads patients digital image (DICOM) into the IGS software application.
5.	IVR confirms that tracking hardware is recognized by IGS software application.
6.	IVR initiates tracking using IGS software application.
7.	IVR performs initial configuration of the software display as pertinent to the procedure.
8.	IVR performs registration.
9.	IVR enables image overlay.
10.	IVR performs visual evaluation of the resulting registration.
11.	IVR loads 3-D display.
12.	IVR finalizes software display for the procedure.
13.	IVR records visual display and saves as pre-operation view
14.	IVR initializes needle tool for tracking.
15.	IVR aligns needle tool (tracking-enabled) for target puncture.
16.	IVR simultaneously inspects alignment and entry angle using IGS software.
17.	IVR completes needle placement.
18.	IVR records visual display and saves as post-operation view.
19.	IVR documents the procedure using events captured by IGS software application and fuses pre- and post-operative images for analysis.

Table 4.4: Formal Requirements Development Process.

1.	Agile developer identifies the need for a new component or enhancement that is extensive enough to justify a requirement.
2.	The WIKI page should be separated into two sections. Open requirements and Iteration-directed requirements. Each section should be separated into the Requirements Taxonomy headers as shown in: http://public.kitware.com/IGSTKWIKI/images/2/2b/REQ-Taxonomy.pdf
3.	The developer should write the drafted requirement and place correctly (i.e. pertinent section) on the WIKI.
4.	The team will consider requirements both open and iteration-directed during weekly telecons.
5.	When a requirement is discussed and approved in a preliminary state, then the requirements lead will move the requirement the PHP Bug tracker
A.	The requirement should be classified (i.e. Severity Field) as either"Style Guideline", "Design Guideline", or "Functional Requirement"
B.	The requirement should be put in the assigned status ("New"/"Unconfirmed"/"Assigned")
C.	The number should be picked based on REQ XX.XX.XX format (e.g. REQ 06.02.10) (This is consistent with the taxonomy)
D.	The requirement number should be appended in bold to the requirement text in the WIKI
6.	If the requirement is tagged for an iteration, then the software code should be developed that corresponds to the requirement.
7.	Once the code is developed is should be moved to the Sandbox.
8.	As the code is tested and reviewed, this may require changes to the requirements
A.	Changes to the requirement text should be entered to the WIKI and discussed during the telecon.
B.	Once a change is agreed upon, again the Requirements Lead makes the correction to the bug tracker by adding a comment
C.	The changed requirement should be stated as NEW_REQ_TEXT: ¡New requirements text¿ (i.e. NEW_REQ_TEXT: The tracker component shall report the maximum refresh rate and time latency on demand.)
9.	Once the code is properly tested and ready for release, the requirement status is moved to "Verified" concurrently with iteration completion

Table 4.5: Fast-tracked Requirements Process.

1.	Agile developer identifies the need for a new component or enhancement that is extensive enough to justify a requirement.
2.	The WIKI page should be separated into two sections. Open requirements and Iteration-directed requirements. Each section should be separated into the Requirements Taxonomy headers as shown in: http://public.kitware.com/IGSTKWIKI/images/2/2b/REQ-Taxonomy.pdf
3.	The developer should write the drafted requirement and place correctly as an open requirement on the WIKI.
4.	The team will consider requirement as an open requirement at the next weekly telecons.
5.	When a requirement is discussed and approved in a preliminary state, then the requirements lead will move the requirement the PHP Bug tracker
A.	The requirement should be classified (i.e. Severity Field) as either ”Style Guideline”, ”Design Guideline”, or ”Functional Requirement”
B.	The requirement should be put in the assigned status (”New”/”Unconfirmed”/”Assigned”)
C.	The number should be picked based on REQ XX.XX.XX format (e.g. REQ 06.02.10) (This is consistent with the taxonomy)
D.	The requirement number should be appended in bold to the requirement text in the WIKI
6.	As an open requirement, the software code should be developed that corresponds to the requirement.
7.	Once the code is developed it should be moved to the Sandbox.
8.	Since fast-tracked software changes tend to be somewhat smaller in scope the code should be approved during a telecon
A.	Any changes to the requirement text should be entered to the WIKI as discussed during the telecon.
9.	Code should be moved to the main repository and requirements text updated in the bug tracker.

IGSTK requirements can be viewed, modified, and enhanced from your web browser. Requirements are captured in the IGSTK BugTracker located at http://www.itk.org/Bug/ . New users can freely register using their current email address. Once registered, the user can access IGSTK requirements by pressing the “Query Bugs” hyperlink in the top middle column. The new page presents a query page for filtering information from the bug tracker. Requirements represent a subset of all information contained in the bug tracker. To facilitate the query, the user should use the advance query page by pressing the hyperlink at the bottom of the page, “Go to the advanced query page”. The advanced query page can be populated as in Figure 4.

Figure 4. Query Configuration to Retrieve IGSTK Requirements.

Brian: Should add a short sniplet of requirements and perhaps final thoughts here.

CHAPTER

FIVE

Software Development Process

"If you can't describe what you are doing as a process, you don't know what you're doing." W Edwards Deming.

IGSTK is intended for use in the operating room. This type of mission-critical software requires a robust software development process that ensures the quality of the software produced. A *robust software development process* for IGSTK means well-defined, well-understood, and well-executed. It does not necessarily mean a heavyweight process definition that imposes large documentation products or rigid constraints that restrict the manner in which developers may innovate. However, the safety requirements of the mission critical domain do mandate that some controls be put in place. IGSTK does this by defining and adhering to a set of best practices for software development. This chapter presents these best practices and then describes the specific tools and techniques that are used to realize these practices.

5.1 IGSTK Best Practices

IGTK developers adhere to the following set of best practices.

1. Recognize that people are the most important mechanism available for ensuring high-quality software. The IGSTK team consists of developers with considerable expertise in the application domain, supporting software, and tools. Their collective judgment outweighs process mandates.

2. Facilitate constant communication. To prevent distributed team members who are working on decoupled components from becoming too isolated, IGSTK members participate in a weekly teleconference and meet in person twice per year. IGSTK also employs a mailing list , instant messaging, and a wiki for online collaboration.

3. Produce iterative releases. IGSTKs development cycle includes twice-yearly external releases. We considered six months too long a horizon to manage development, so internal

releases are broken down into approximately two-month iterations. At the end of an iteration, team members perform quality reviews and move code considered stable to the main repository.

4. Manage source code from a quality perspective. IGSTK defines different configuration management policies to satisfy different quality criteria. Codelines with separate policies, for example a main repository and a sandbox, lets developers collaborate on code that might not yet meet stringent requirements. Exploiting configuration management approaches early in a project helps document and track quality progress.

5. Focus on 100 percent code and path coverage code coverage at the component level. Unit tests ensure complete code coverage across all platforms. We are also developing customized visualization and validation tool machines to guarantee that correctness properties within all IGSTK state machines are verified with every nightly build. In addition, dynamic analysis tools prevent memory leaks and access violations.

6. Emphasize continuous builds and testing. IGSTK uses the open source DART (http://public.kitware.com/Dart/HTML/Index.shtml) tool to produce a nightly dashboard of build and unit test results across all supported platforms.

7. Support the development process with robust tools. In addition to Dart, IGSTK employs the CMake (http://www.cmake.org) open source cross-platform build solution, KWStyle (http://public.kitware.com/KWStyle) for source code style checking, Doxygen (http://www.stack.nl/ dimitri/doxygen) , an open source documentation system, phpBugTracker (http://phpbt.sourceforge.net) a bug tracking system, and CVS a source code version control system. Best practices for coding and documentation posted on the wiki augment these tools.

8. Manage requirements iteratively in lockstep with code management. As requirements evolve and the code matures, adopting flexible yet defined processes for managing requirements becomes necessary.

9. Focus on meeting exactly the current set of requirements. Traceability is needed in safety-critical domains, particularly in surgical applications that need to satisfy government regulations, and implies heavy process structures with large documents and invasive tools. IGSTK addresses this problem with continuous requirements review, lightweight tools, and codeline policies.

10. Evolve the development process. Through constant communication, IGSTK members recognize when to approach the complexities they face within the current process framework, when tweaks are required, or when to adopt entirely new practices.

IGSTK is *agile*. The approach followed by the team and recommended for component and application developers is to employ *lightweight* methods. Traditional approaches that introduce rigid process steps with volumes of documentation may actually lead to a decrease in quality and safety for projects such as IGSTK due to the distributed collaborative nature of open source software development. It is not beneficial to create strict process controls that can not, nor will not, be adhered to in such an environment. The resulting process would lead to inconsistent documentation and poor execution of the defined process, leading to false hope that by having such a strict process definition quality will be achieved.

IGSTK uses the best practices presented above as a means to achieve quality and safety by executing these practices throughout the software development process. The emphasis is on agile execution; if execution of a defined process is good, IGSTK executes these practices constantly. Tools, reviews, communication, builds, testing, release management are all performed in a highly iterative continuous manner to ensure the quality and safety of the software produced.

5.2 Developer Practices

5.2.1 Code Conventions

Understandability of source code is a critical aspect in the maintenance of a software system. Its importance is magnified in open source projects such as IGSTK that rely on a large distributed development community to evolve the framework and construct applications. Defining enforceable coding standards and conventions is a powerful technique for ensuring the maintainability of an open source codebase. IGSTK employs a combination of tools, practices, and conventions to ensure understandability and maintainability of the source code.

IGSTK source code conventions are included in Appendix A. These include stylistic conventions, file organization, and best practices for developers (use of exceptions, macros, STL, etc.). All IGSTK component and application developers are strongly encouraged to review these conventions and adhere to them to the greatest extent possible. In this section we highlight some of the best practices and then discuss the use of the Doxygen and KWStyle tools.

The code conventions in Appendix A includes recommended best practices within the source code that all IGSTK component and application developers should follow. IGSTK discourages the use of generics (C++ templates), and encourages the use of the Standard Template Library (STL) but not at the API level. IGSTK relies on strong type checking to ensure API contracts, and the unnecessary use of templates may lead to type-related runtime errors.

IGSTK advocates the use “smart pointers” to manage object references for objects with a significant memory footprint. Memory management can be an area where it is particularly hard to detect and correct defects. The use of smart pointers, while adding some overhead to application execution, reduces the opportunity for memory-related defects. IGSTK also advocates const correctness. As stated in the Appendix, “A safe approach is to start considering everything as const and making classes and methods non-const only when a justification exists. Const verification is done by the compiler and prevents inappropriate and unsafe use of the classes and methods.” IGSTK has created macros that assist with using smart pointers and for enforcing setter/getter contracts.

The Doxygen documentation tool generates external documentation from commented source code. Doxygen automatically extracts comments delimited in a special way to generate the external documentation. The IGSTK style guide, Appendix A, section 10 defines Doxygen documentation conventions for classes and methods. These comments are only extracted from C++ header (.h) files. IGSTK developers are required to keep comments up to date with source code changes.

Coding stylistic conventions can often be the hardest standards for developers to consistently adhere to over a lengthy period of time. As code grows and evolves, enforcement of stylistic conventions via developer diligence and code reviews is tedious to maintain. The KWStyle tool performs static code checks on over 20 stylistic properties of C++ source code. IGSTK has codified stylistic rules in KWStyle and integrates KWStyle analysis into the nightly Dart dashboard. The KWStyle tool removes the time-consuming tedious process of checking for stylistic conformance and ensures code style does not degrade over time. KWStyle is open source freely available for download. You may also demo the tool via the web using the *Check my File* option off the KWStyle homepage.

Source code is best understood and maintained when it looks as if it was written by a single developer. This is especially true when considering open source. Readers of the source code can more easily understand the intent of the code when they can apply a single mental model when "internally parsing" it. Misinterpretation of source code intent may decrease code quality and application safety. Developers could add new features in the wrong place, or patch existing code in an unsafe manner, or invoke services in an improper manner. Tools such as Doxygen, KWStyle, and Dart can help developers adhere to conventions, but in the end developers must accept responsibility for creating readable, understandable, and maintainable source code. IGSTK core component source code reviews (see 5.2.2 below) always include detailed checks that these conventions are followed.

5.2.2 Code Reviews

Source code reviews are well-known as one of the best, if not the best, method to ensure quality software. An IGSTK code review is an informal review facilitated by the managed communication methods described below. IGSTK code reviews are integrated into the configuration management policies governing the source code repository and the iterative development cycle of IGSTK. Developers constructing applications using IGSTK are strongly encouraged to employ a code review process, leveraging other team members or the IGSTK community.

IGSTK code reviews do not have the formality of a software inspection (see [4, 14]), *Luis:Fagan76 and Wiegers02 are to be added to IGSTKDocuments/Latex/IGSTK.bib* but are not so informal as to lack a record of defects found and fixed. IGSTK code reviews are performed by at least two reviewers, at least one of which should have deep knowledge of the functional domain of the introduced code. Reviewers use the IGSTK coding standards document, the requirements repository, the Wiki, and tools such as KWStyle and DART to facilitate reviewing the code. Defects found by the reviewer are first posted to the Wiki to give an opportunity to the original developer to make fixes. Applying the principle of collective code ownership, the reviewer or other members of the development community may also perform fixes. Usually this process is fluid in the sense that the reviewers and the original developer communicate (via email, Wiki, or phone calls) to ensure a common understanding of the defect and the proper fix.

Developers endeavor to fix defects as soon as possible in the current version of the source code, to prevent lingering defects that may propagate to other branches and releases of the software. In those cases where it is decided there is not an immediate solution, defects may be entered into the defect tracking repository. In this way, all defects found by reviewers are addressed by the team before the code is released.

As important as how the code review takes place is *when* the review takes place. All source code must complete a code review process before being included in an IGSTK release.

Code reviews are an important, arguably the most important, quality technique that can be applied to software development. Code reviews complement automated unit tests and code analysis tools by adding a dimension of expert evaluation to the source code. Further, code reviews reinforce the principle of collective code ownership and common source code understanding. IGSTK component developers are required to perform code reviews on every line of source code included in a framework release; IGSTK application developers are strongly encouraged to do the same.

5.2.3 Managed Communication

IGSTK was developed by a team of developers geographically distributed on a wide scale, and is intended to support the global community of interest in image-guided software for surgical applications. The IGSTK community makes use of websites, Wikis , and mailing lists to support the community. The main IGSTK website (http://www.igstk.org) has the latest news and information about IGSTK releases and other developments. The IGSTK Wiki (http://public.kitware.com/IGSTKWIKI/index.php) provides an online interactive forum for IGSTK developers. Modifications to the Wiki are restricted to authorized users.

The IGSTK community supports two mailing lists, one for developers and one for users. Participants on the developers mailing list are core IGSTK developers. The users list supports developers who intend to build applications on top of IGSTK. If you choose to download IGSTK and build an application, we strongly recommend joining the users' mailing list to interact with the IGSTK community. You may join and view mailing list archives at `http://public.kitware.com/mailman/listinfo/igstk-users`.

One the foundational principles of open source development is *community*. The *open* in open source refers not just to the code but to the community. IGSTK intends to promote and support community involvement for the toolkit, and the principle vehicles for this are the communication tools described here. We encourage you to become a participant in the IGSTK community.

5.2.4 Configuration Management

Luis: the term "Configuration Management" maps better to what CMake does. We probably should call this section "Source Code Control" or something along those line.

It is not always obvious how to much software has pervasively invaded our modern environment. Most electric and electronic appliances contain microchips that require dedicated software at different levels. The complexity of the software correlates with the complexity of the device; in particular with the number of different tasks that the device is supposed to perform as well as the combination of such task whether simultaneously or in sequence.

As an illustration of how much software surround us, here are some of the typical amounts of lines of code in the software of modern devices.

- A Laundry appliance has a couple thousand lines of code.
- The Joint Strike Fighter F-35 has six million lines of code.[1]
- The Boeing 777 has 2.5 million lines of *newly developed* code. Approximately six times more than any previous Boeing commercial airplane. When including commercial off-the-shelf and optional software, the total is more than 4 million lines of code.[2]
- A modern automobile has 35 million lines of code.
- The operating system, Windows XP has 40 million lines of code.

As a comparison, here are the amount of lines of code in IGSTK and the toolkits it depends on. The units are thousands of lines of code "KLOC".

- IGSTK: 30 KLOC, 11 of which are for testing
- ITK: 562 KLOC, 90 of which are for testing
- VTK: 868 KLOC, 70 of which are for testing
- FLTK: 107 KLOC, 9 of which are for testing

Tracking the modifications made on this large amount of code requires the systematic use of a configuration management system.

Configuration Management (CM) is important for enforcing and maintaining the quality of software products. Most developers are familiar with using CM tools in projects large and small, but this use is often restricted to versioning files (or collections of files associated with a job) for the purposes of evolving and maintaining software. A common scenario is for a developer, when addressing a defect entered in the defect tracking repository, to *checkout* the associated files, create a fix, locally test the fix, and check the files back in to the code repository. If any changes were made to the same files by other developers, the CM system will typically inform the developer and provide various pessimistic or optimistic strategies for resolving the issue. Another common scenario is for a developer to be working on a new feature for the next release, and to create new source files or modify existing ones. Again, if changes are required to existing files, these new versions are checked in to the code repository under the purview of the CM system.

[1] http://www.afa.org/magazine/april2003/0403F35.asp
[2] http://www.stsc.hill.af.mil/crosstalk/1996/01/Boein777.asp

IGSTK uses the popular and well-known Concurrent Version System (CVS) tool for version control of software products. The commands needed for obtaining IGSTK and supporting software packages such as ITK from CVS are described in Chapter 2. IGS Application evelopers are encouraged to use anonymous CVS access to download IGSTK. You may use the *export* cvs function to obtain a copy, or do a regular *cvs checkout* if you wish to maintain local repository information for the purposes of tracking histories, differences, and new versions of the software you have downloaded. Detailed online references for CVS may be found at the CVS websites for the CVS book (http://cvsbook.red-bean.com) and the CVS wiki (http://ximbiot.com/cvs/wiki/index.php?title=Main_Page).

IGSTK recognizes CM as an important quality process, one that must be supported and integrated throughout the entire software lifecycle. IGSTK uses CVS to support defect fixes and the addition of new features as described by the above scenarios. However, IGSTK goes further by supporting best practices in CM such as minimizing branch creation to reduce product version proliferation, and establishing multiple *codelines* and distinct *codeline policies* for source code at different stages in the development process. A *branch*, in CM terms (and specifically in CVS) is a named (or *tagged*) set of source code files that are then modified in a copy independent from the codeline from which the branch originated. Branches are a useful tool for supporting development on a codebase that is undergoing multiple types of changes but is under the same quality constraints. For example, the main codeline (or *trunk* is often reserved for new feature evolution, while a branch is created from a stable release of the code for the purposes of supporting defect fixes on the code while not disturbing new feature development. At a defined point in the lifecycle of the new release, defect fixes may be *merged* back into the main codeline. This activity is usually controlled by a Change Control Board (CCB) comprised of representative of all stakeholders of the codeline. Another reason to branch might be to support custom features for a particular customer or project that are not targeted for the main product line. When a branch is no longer needed (because the release or custom project is no longer supported) it is deprecated. While branches are useful for these scenarios, branch proliferation can be problematic to manage. A single change anywhere in the codebase now has to be reviewed by more stakeholders to determine if that change applies to them. In effect, it creates multiple release versions of the software, resulting in additional complexity for requirements change management and testing processes. Therefore generally accepted axioms are to branch as late as possible and as seldom as possible to avoid this complexity. IGSTK, during its development, did not create branches, though it did employ separate codelines.*Luis: We actually branched IGSTK at the release of every Iteration. Branching is standard practice for a release. Bug fixes that go in to a release branch are manually applied also to the main trunk.*

A *codeline* is a repository of source code distinct from other repositories. The main codebase, or trunk, is a codeline. A codeline has an associated set of rules that govern where, when, and what developers may commit changes to the codebase. The *where* determines what branch of a codeline to commit a modification. The *when* determines at what point in the lifecycle of that codebase the modification may be submitted. The *what* defines the criteria by which modifications are allowed, possibly including *who* may submit such modifications. The when, what, and who of codeline management is critical is applying different quality criteria to different source code files. For example, the main codeline policy in IGSTK states that only developers on the product development team may commit changes, and may only do so after a complete unit test and code review of the source code. An experimental codeline, where developers are working collaboratively in an evolutionary manner on a high-risk feature, may define lower quality standards such as informal walkthroughs and lesser stylistic checks in order to facilitate rapid development. However, note that if an experimental feature were to be targeted for the main product line, the code could not be moved to that line without upgrading the quality checks to meet the quality policies defined for the main codeline. In this way, using separate codelines better enforces quality standards than simple branch-and-merge.

Since IGSTK included the development of several innovative features as well as an innovative architecture, it used a multiple codeline approach known as *sandboxing*. A sandbox is a separate codeline with lesser quality policies where developers working collaboratively could prototype high-risk code. For these mini-projects that were deemed successful, the code was then subjected to more rigorous quality policies so it could be moved into the main IGSTK codeline. The quality policies on the IGSTK main codeline include adherence to IGSTK style guidelines, unit tests present for all behaviors, complete code coverage (every source line executed), no dynamic analysis (memory management) defects, cross-platform verification, traceability of the source code back to requirements, and a complete code review according to IGSTK's defined code review practices.

Going forward, IGSTK will support periodic releases, and support tagging and branching in CVS as appropriate to support maintenance of those releases. These policies will be posted on the IGSTK wiki. Future component developers for the toolkit will be expected to adhere to the quality criteria described for the main product line. Application developers are encouraged to follow IGSTK's CM patterns when developing and supporting applications that depend on IGSTK. Only through the application of consistent quality policies can quality and safety of software for surgical environments be achieved. For further reading on best practices for CM, we recommend [15]. *Luis: Perforce1998 must be added to the Insight.bib file*

Finally, we emphasize one related practice in IGSTK that is often not considered as a CM-type problem. Unlike many component-based systems being created today, IGSTK specifically avoids run-time configuration of framework behaviors via configuration files. While many component-based systems, such as web-based applications or embedded systems, use external configuration files to set run-time behavior and component "wiring", IGSTK avoids this in favor of compiling a single pre-configured binary version. In this way, clinicians do not have to worry about misconfigured software deployments in the operating environment. This is an example of another configuration management best practice, as it ensures that the behaviors deployed in a given environment have been enforced by the compiler and build process in general, including the testing framework. In effect, instead of many possible runtime versions of the software in an operating environment, there is exactly one version deployed, and it is the version targeted for that environment. In the next section we elaborate on the IGSTK build process and how it helps enforce safety.

5.2.5 Build and Release Management Processes

IGSTK has an internal release cycle of approximately every two to three months and an external public release cycle planned for twice a year. Availability of new public releases will be driven by the energy of the community and the need for new features. As it is the expressed purpose of IGSTK to remain a small and safe toolkit for a dedicated purpose, new enhancement requests will be reviewed based on necessity, not desirability. Please check the IGSTK website for continuing updates on upcoming planned releases of IGSTK.

Internal releases are available off the IGSTK wiki. Though not "official" public releases, these versions are made available so that community developers have access to the latest internally stable versions of the software. Although considered "internal" releases, these are still subject to a rigorous internal process:

1. The sandbox repository is frozen and tagged.*Luis:The main CVS is also tagged at this stage*
2. Code review is scheduled for new classes. Two reviewers are assigned to review each class.
3. Reviewers review code according to established guidelines. A code review checklist is available for this purpose.
4. Authors fix code.
5. Reviewed and fixed code will be moved from the sandbox to the main CVS repository.
6. Pending bugs and dynamic analysis (memory leak) issues are fixed and code coverage is increased.
7. The main CVS and sandbox repository gets tagged.
8. Downloadable tarballs are generated and uploaded to the Wiki.

The main difference between internal and external releases is that external releases represent the completion of a collection of functionality targeted by the IGSTK community to constitute a major release.

Instructions for downloading and building IGSTK (and VTK/ITK upon which IGSTK relies) are provided in Chapter 2.

5.2.6 Continuous Testing using DART

Software Quality Statistics

Software bugs, or errors, are so prevalent and so detrimental that a study entitled "The Economic Impacts of Inadequate Infrastructure for Software Testing"[3] produced for the U.S. Department of Commerce's National Institute of Standards and Technology (NIST) found in 2002 that

[3]http://www.nist.gov/director/prog-ofc/report02-3.pdf

> *The national cost estimate of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion. This represents about 0.2 to 0.6 percent of the U.S.'s $10 trillion dollar gross domestic product (GDP). Over half of these costs are borne by software users in the form of error avoidance and mitigation activities. The remaining costs are borne by software developers and reflect the additional testing resources that are consumed due to inadequate testing tools and methods.*
>
> *Although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. These are the savings associated with finding an increased percentage (but not 100 percent) of errors closer to the development stages in which they are introduced. Currently, over half of all errors are not found until "downstream" in the development process or during post-sale software use.*
>
> *Software is error-ridden in part because of its growing complexity. The size of software products is no longer measured in thousands of lines of code, but in millions. Software developers already spend approximately 80 percent of development costs on identifying and correcting defects, and yet few products of any type other than software are shipped with such high levels of errors. Other factors contributing to quality problems include marketing strategies, limited liability by software vendors, and decreasing returns on testing and debugging, according to the study. At the core of these issues is difficulty in defining and measuring software quality.*

It is important to note that this report is gathering statistics from application domains that are not considered to be be mission-critical. That is, the cost referred in this study do not include software error events such as the Mars Polar Lander crash in 1999 ($165 million), the Arian V explosion in 1996 ($500 million), the breakdown of the radio system linking air traffic controllers in southern California in 2004 leaving 800 planes in the air without airport guidance, the crash of the Lufthansa Airbus A320 in Warsaw in 1993, the failure of the Cryosat Russian rocket in 2005, the software fault in anti-lock brakes that forced the recall of $39,000$ trucks and tractors and $6,000$ school buses in 2000 or the software bug that motivated the recall of $160,000$ Toyota Prius hybrid cars in 2005.

> *Quantifying the impact of inadequate testing on mission critical software was beyond the scope of this report. Mission critical software refers to software where there is extremely high cost to failure, such as loss of life. Including software failures associated with airbags or anti-lock brakes would increase the national impact estimates.*
>
> *Finally, the costs of software errors and bugs to residential households is not included in the national cost estimates. As the use of computers in residential households to facilitate transactions and provide services and entertainment increases, software bugs and errors will increasingly affect household production and leisure. Whereas these software problems do not directly affect economic metrics such as*

GDP, they do affect social welfare and continue to limit the adoption of new computer applications.

IGSTK relies on extensive automated unit testing to ensure all lines of code are verified along some execution path. Automation is provided by CMake and DART integration. DART is a regression testing system that supports web-based report generation for a variety of test types. The web reports generated by DART creates an online *dashboard* that the entire team uses on a daily basis to understand exactly what the quality level of the source code is at that instant in time. A sample IGSTK dashboard screenshot is shown in Figures 5.1 and 5.2.

ITK and VTK developers and users should already be familiar with the DART dashboard concept; the philosophy of complete continuous regression testing in IGSTK was adapted from ITK. Readers familiar with Chapter 14 of the ITK Software Guide [9] will recognize the following descriptions of test types, slightly abridged from that chapter for IGSTK:

1. Compilation. All source and test code is compiled and linked. Any resulting errors and warnings are reported on the dashboard.

2. Regression. Some IGSTK tests require comparing test output against a valid baseline image. If the images match then the test passes. The comparison must be performed carefully since many graphics systems (e.g., OpenGL) produce slightly different results on different platforms. IGSTK also performs regression tests on Tracker-related operations.

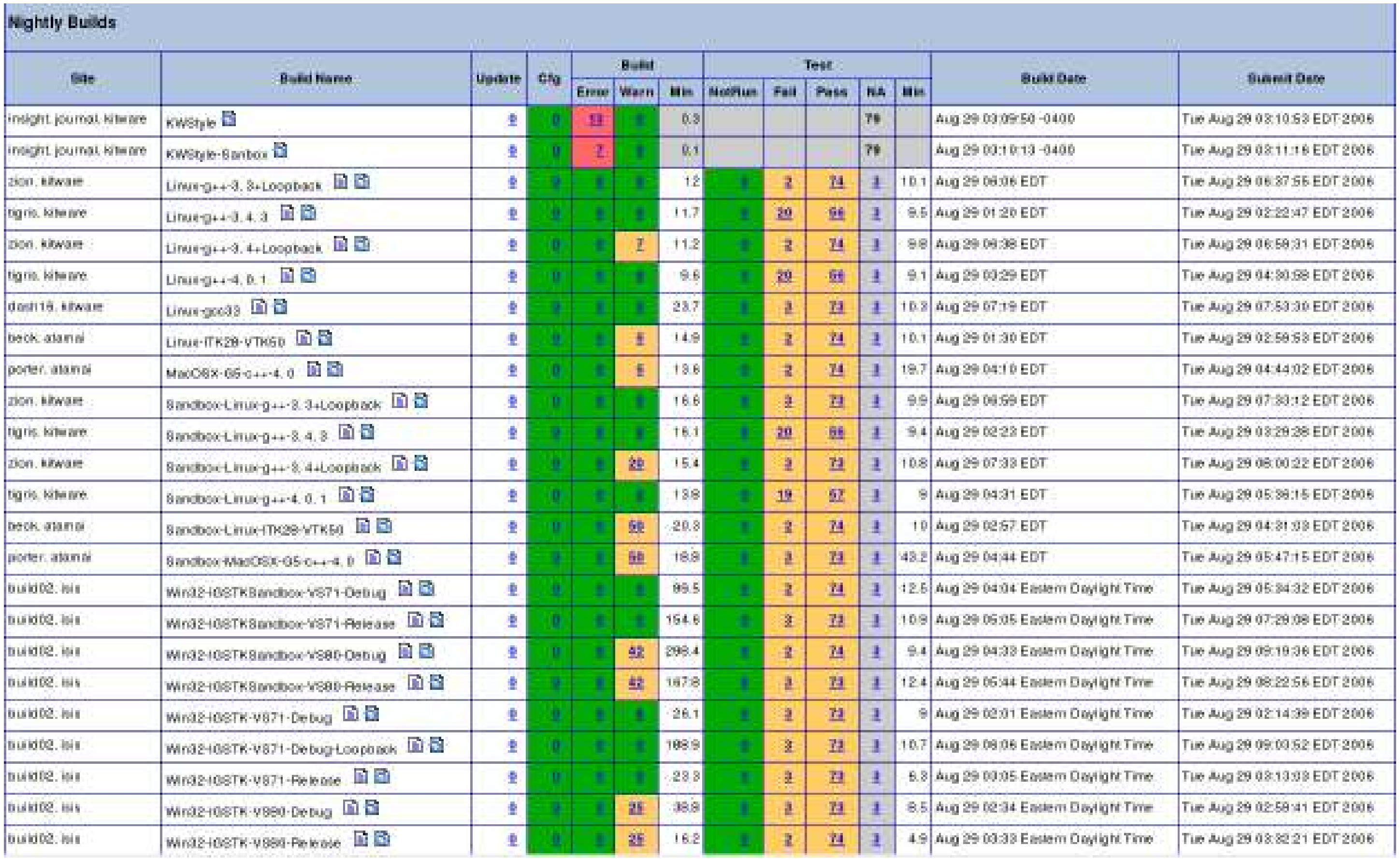
3. Memory. Problems relating to memory such as leaks, uninitialized memory reads, and reads/ writes beyond allocated space can cause unexpected results and program crashes. IGSTK checks for run-time memory errors using Valgrind (http://valgrind.org), a freely available open source debugging platform for Linux.

4. PrintSelf. IGSTK follows the ITK practice of having each class implement a PrintSelf method to print out all instance variables. The CMake-configured unit test driver checks to make sure that this is the case.

5. Unit. Each class in IGSTK must have a corresponding unit test where all 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.

6. Coverage. IGSTK unit tests should ensure that each and every line of code is executed at least once by the suite of available unit tests. This is commonly referred to as *node coverage*, and is the most important type of coverage for IGSTK as the state machine architecture ensures that conditionals are kept to an absolute minimum, thereby reducing the need for edge and full path coverage. For safety purposes, the goal is 100 percent coverage for source code committed to the main codeline. In practice the IGSTK dashboard has usually been at 90 percent coverage or greater.

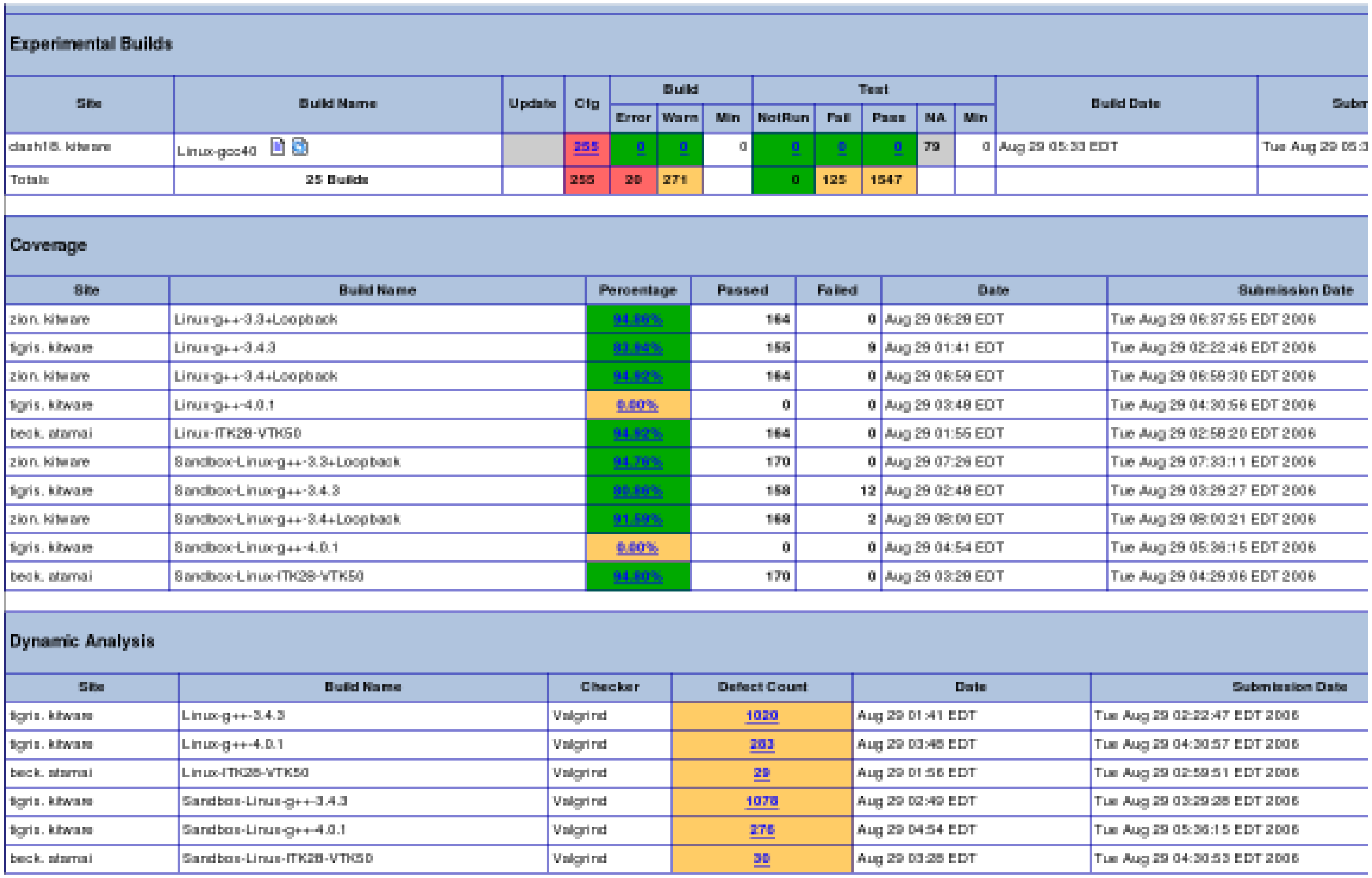
7. Style checking. The KWStyle tool described earlier in this chapter also provides a table-formatted display of stylistic violations.



Nightly Builds

Site	Build Name	Update	Cfg	Build			Test					Build Date	Submit Date
				Error	Warn	Min	NotRun	Fail	Pass	NA	Min		
insight. journal. kitware	KWStyle	0	0	13	0	0.3				79		Aug 29 03:09:50 -0400	Tue Aug 29 03:10:53 EDT 2006
insight. journal. kitware	KWStyle-Sanbox	0	0	7	0	0.1				79		Aug 29 03:10:13 -0400	Tue Aug 29 03:11:16 EDT 2006
zion. kitware	Linux-g++-3. 3+Loopback	0	0	0	0	12	0	2	74	3	10.1	Aug 29 06:06 EDT	Tue Aug 29 06:37:55 EDT 2006
tigris. kitware	Linux-g++-3. 4. 3	0	0	0	0	11.7	0	20	56	3	9.5	Aug 29 01:20 EDT	Tue Aug 29 02:22:47 EDT 2006
zion. kitware	Linux-g++-3. 4+Loopback	0	0	0	7	11.2	0	2	74	3	9.8	Aug 29 06:38 EDT	Tue Aug 29 06:59:31 EDT 2006
tigris. kitware	Linux-g++-4. 0. 1	0	0	0	0	9.6	0	20	56	3	9.1	Aug 29 03:29 EDT	Tue Aug 29 04:30:58 EDT 2006
dash16. kitware	Linux-gcc33	0	0	0	0	23.7	0	3	73	3	10.3	Aug 29 07:19 EDT	Tue Aug 29 07:53:30 EDT 2006
beck. atamai	Linux-ITK28-VTK50	0	0	0	6	14.9	0	2	74	3	10.1	Aug 29 01:30 EDT	Tue Aug 29 02:59:53 EDT 2006
porter. atamai	MacOSX-G5-c++-4. 0	0	0	0	6	13.6	0	2	74	3	19.7	Aug 29 04:10 EDT	Tue Aug 29 04:44:02 EDT 2006
zion. kitware	Sandbox-Linux-g++-3. 3+Loopback	0	0	0	0	16.6	0	3	73	3	9.9	Aug 29 06:59 EDT	Tue Aug 29 07:30:12 EDT 2006
tigris. kitware	Sandbox-Linux-g++-3. 4. 3	0	0	0	0	16.1	0	20	56	3	9.4	Aug 29 02:23 EDT	Tue Aug 29 03:29:28 EDT 2006
zion. kitware	Sandbox-Linux-g++-3. 4+Loopback	0	0	0	20	15.4	0	3	73	3	10.8	Aug 29 07:33 EDT	Tue Aug 29 08:00:22 EDT 2006
tigris. kitware	Sandbox-Linux-g++-4. 0. 1	0	0	0	0	13.8	0	19	57	3	9	Aug 29 04:31 EDT	Tue Aug 29 05:38:15 EDT 2006
beck. atamai	Sandbox-Linux-ITK28-VTK50	0	0	0	50	20.3	0	2	74	3	10	Aug 29 02:57 EDT	Tue Aug 29 04:31:33 EDT 2006
porter. atamai	Sandbox-MacOSX-G5-c++-4. 0	0	0	0	50	18.8	0	3	73	3	43.2	Aug 29 04:44 EDT	Tue Aug 29 05:47:15 EDT 2006
build02. isis	Win32-IGSTKSandbox-VS71-Debug	0	0	0	0	99.5	0	2	74	3	12.5	Aug 29 04:04 Eastern Daylight Time	Tue Aug 29 05:34:32 EDT 2006
build02. isis	Win32-IGSTKSandbox-VS71-Release	0	0	0	0	154.6	0	3	73	3	10.9	Aug 29 05:05 Eastern Daylight Time	Tue Aug 29 07:29:08 EDT 2006
build02. isis	Win32-IGSTKSandbox-VS80-Debug	0	0	0	42	298.4	0	2	74	3	9.4	Aug 29 04:33 Eastern Daylight Time	Tue Aug 29 09:19:36 EDT 2006
build02. isis	Win32-IGSTKSandbox-VS80-Release	0	0	0	42	187.8	0	3	73	3	12.4	Aug 29 05:44 Eastern Daylight Time	Tue Aug 29 08:22:56 EDT 2006
build02. isis	Win32-IGSTK-VS71-Debug	0	0	0	0	26.1	0	3	73	3	9	Aug 29 02:01 Eastern Daylight Time	Tue Aug 29 02:14:39 EDT 2006
build02. isis	Win32-IGSTK-VS71-Debug-Loopback	0	0	0	0	188.9	0	3	73	3	10.7	Aug 29 08:06 Eastern Daylight Time	Tue Aug 29 09:03:52 EDT 2006
build02. isis	Win32-IGSTK-VS71-Release	0	0	0	0	23.3	0	3	73	3	6.3	Aug 29 03:05 Eastern Daylight Time	Tue Aug 29 03:13:03 EDT 2006
build02. isis	Win32-IGSTK-VS80-Debug	0	0	0	26	38.8	0	3	73	3	8.5	Aug 29 02:34 Eastern Daylight Time	Tue Aug 29 02:58:41 EDT 2006
build02. isis	Win32-IGSTK-VS80-Release	0	0	0	26	16.2	0	2	74	3	4.9	Aug 29 03:33 Eastern Daylight Time	Tue Aug 29 03:32:21 EDT 2006

Figure 5.1: Dart Dashboard Screenshot: Nightly Builds.



Experimental Builds

Site	Build Name	Update	Cfg	Build			Test					Build Date	Subr
				Error	Warn	Min	NotRun	Fail	Pass	NA	Min		
dash18. kitware	Linux-gcc40		255	0	0	0	0	0	0	79	0	Aug 29 05:33 EDT	Tue Aug 29 05:3
Totals	25 Builds		255	20	271		0	125	1547				

Coverage

Site	Build Name	Percentage	Passed	Failed	Date	Submission Date
zion. kitware	Linux-g++-3.3+Loopback	94.86%	164	0	Aug 29 06:28 EDT	Tue Aug 29 06:37:55 EDT 2006
tigris. kitware	Linux-g++-3.4.3	93.94%	156	9	Aug 29 01:41 EDT	Tue Aug 29 02:22:46 EDT 2006
zion. kitware	Linux-g++-3.4+Loopback	94.82%	164	0	Aug 29 06:59 EDT	Tue Aug 29 06:59:30 EDT 2006
tigris. kitware	Linux-g++-4.0.1	0.00%	0	0	Aug 29 03:48 EDT	Tue Aug 29 04:30:56 EDT 2006
beck. atamai	Linux-ITK28-VTK50	94.82%	164	0	Aug 29 01:56 EDT	Tue Aug 29 02:58:20 EDT 2006
zion. kitware	Sandbox-Linux-g++-3.3+Loopback	94.78%	170	0	Aug 29 07:28 EDT	Tue Aug 29 07:33:11 EDT 2006
tigris. kitware	Sandbox-Linux-g++-3.4.3	88.66%	158	12	Aug 29 02:48 EDT	Tue Aug 29 03:29:27 EDT 2006
zion. kitware	Sandbox-Linux-g++-3.4+Loopback	91.59%	168	2	Aug 29 08:00 EDT	Tue Aug 29 08:00:21 EDT 2006
tigris. kitware	Sandbox-Linux-g++-4.0.1	0.00%	0	0	Aug 29 04:54 EDT	Tue Aug 29 05:36:15 EDT 2006
beck. atamai	Sandbox-Linux-ITK28-VTK50	94.80%	170	0	Aug 29 03:28 EDT	Tue Aug 29 04:29:06 EDT 2006

Dynamic Analysis

Site	Build Name	Checker	Defect Count	Date	Submission Date
tigris. kitware	Linux-g++-3.4.3	Valgrind	1020	Aug 29 01:41 EDT	Tue Aug 29 02:22:47 EDT 2006
tigris. kitware	Linux-g++-4.0.1	Valgrind	283	Aug 29 03:48 EDT	Tue Aug 29 04:30:57 EDT 2006
beck. atamai	Linux-ITK28-VTK50	Valgrind	29	Aug 29 01:56 EDT	Tue Aug 29 02:59:51 EDT 2006
tigris. kitware	Sandbox-Linux-g++-3.4.3	Valgrind	1078	Aug 29 02:49 EDT	Tue Aug 29 03:29:28 EDT 2006
tigris. kitware	Sandbox-Linux-g++-4.0.1	Valgrind	276	Aug 29 04:54 EDT	Tue Aug 29 05:36:15 EDT 2006
beck. atamai	Sandbox-Linux-ITK28-VTK50	Valgrind	30	Aug 29 03:28 EDT	Tue Aug 29 04:30:53 EDT 2006

Figure 5.2: Dart Dashboard Screenshot: Continuous Builds, Code Coverage, Memory Testing.

These test types are augmented by a set of demo applications that exercise IGSTK functionality, and a state machine validation tool to ensure proper construction and execution of component state machines (see Appendix B).

Each weekly teleconference of IGSTK developers includes a review of Dashboard status and action items to bring any quality parameters back inline if they are out of bounds. The focus is on continually and aggressively maintaining safety and quality, and the combination of CTest (from CMake) and DART dashboard provides the best way to do this. DART is freely available; IGSTK application developers are encouraged to contribute CTest submissions to the IGSTK DART dashboard, as well as to setup their own DART server and run cross-platform unit tests of their own applications on a nightly basis. Application developers may want to keep their dashboards private, this can easily be taken care of when they setup their own DART dashboard. The IGSTK community is available to assist with this process.

Finally, IGSTK believes in defect tracking , and uses a customized implementation of the open source PHP BugTracker (http://phpbt.sourceforge.net). The IGSTK team addresses defects as soon as they are found, but any defects that remain unresolved for longer than a short window of time are entered into the defect tracking repository. The defect is revisited at least on every internal release boundary.

5.3 Software Development Process Summary

IGSTK employs an agile philosophy for the development of safety-critical software. While the lightweight nature of agile methods might give some cause for concern, we believe that vigilance in the application of the process is the most important aspect for creating quality. A process is not a good process if it is not followed by the developers.

The best practices advocated at the beginning of this chapter emphasize people as the key aspect of any process. However, this does not mean developers are free to simply discard the process when they deem convenient. As an open source, community-oriented project, IGSTK welcomes developer and user involvement, and asks that the community adhere to these principles in order to ensure the continued high-quality and safety goals of IGSTK. The IGSTK development community welcomes not only questions on the design aspects embodied on the source code, but also on the proper execution of agile and test-driven methods to attain these goals.

Part II

Components

CHAPTER

SIX

StateMachine

6.1 General Background

State Machines are a fundamental concept in computer programming. They were introduced by Alan Turing in 1936 as a formalism for supporting his work on determining whether the execution of an algorithm will ever stop or not. This problem is also known as the *Entscheidungsproblem* problem. A State Machine is defined by a set of states, a set of inputs and a set of transitions from one state to another. A Finite State Machine (FSM) is a state machine where the number of states is finite, and a Deterministic State Machine (DSM) is one where a given input presented to a given state will always led to a unique state [3, 11].

In practice all computers are state machines, unfortunately their number of possible states is so large that they can barely be considered to be FSMs. An alternative way of looking at this large number of states is to assume that some of those states are not modeled in the FSM itself and therefore they become elements of randomness on the behavior of the state machine. In this interpretation, the state machine appears as a Non-Deterministic State Machine due to our ignorance or lack of awareness of the non-modeled states. This apparently non-deterministic scenario is probably the one that better describes a typical modern computer software, and it is the kind that should be avoided in safety-critical applications.

6.2 Motivation

The fundamental motivation for introducing the use of State Machines in IGSTK is to improve the safety and robustness of the library, and by this mean, to protect patients from harm.

Computer programs, in particular those that are modeled using Object Oriented programming are naturally well-suited to be described in terms of state machines. Unfortunately, the lack of formality in traditional programming techniques leads to programs that are equivalent to under-defined state machines, where the states are poorly structured, and the transitions between states are rarely stated explicitly. Such relaxed programming practices produce programs that behave erratically and unpredictably. Those are exactly the kind of behaviors that are unacceptable in a safety-critical application such as image-guided surgery.

The sake of reliability and robustness in IGSTK lead the development team to opt for the use of the State Machine model at the very early stages of the project. State Machines are an excellent way of limiting the number of possible behaviors, ensuring that a program will always be in a valid condition, and that all possible behaviors have been considered in advance by the developer team in order to anticipate appropriate responses. A well defined architecture based on state machines makes possible to guarantee repeatability and deterministic behavior [7, 2]

The introduction of state machines makes possible to improve the safety and robustness of the source code by enforcing the following characteristics

- Deterministic Behavior
- Preclude wrong Use
- Robustness to misuse
- Managing Complexity
- Traceability
- Suitability for Testing
- Consistent Documentation

6.2.1 Deterministic Behavior

Deterministic behavior is the characteristic of a program for providing exactly the same response when exposed to exactly the same type of input. The main reason why determinism is a desirable feature is that it gives sense to the effort of testing intensively the software. In the absence deterministic behavior the process of software testing would becomes pointless because non-deterministic programs could behave fine during the testing stages and still may fail during the execution of a surgical intervention. The whole purpose of performing testing is to be able to build confidence on the quality of the software and to reasonably expect that if the software behave correctly during the testing sessions, it will do as well when it is used for guiding the execution of a surgical procedure.

6.2.2 Preclude Wrong Use

The introduction of state machines in IGSTK made possible to have a layer of logic that separates the users of a class from the actual actions that the class can perform. In traditional object oriented programming, a class provides a set of public methods that can be called from any other piece of software, and a set of private methods that only the class can call internally. Usually the private methods are those that can only be called when specific conditions are met inside the class, and therefore only the class itself is qualified for deciding whether such private methods should be called or not. Unfortunately, it is commonly the case that many of the remaining public methods still have a number of assumptions regarding the order and the conditions in which they should be invoked. Developers, being aware of such conditions, tend to test the classes by following a set of non-explicit rules that make sure that the conditions are satisfied. User's of the software, however, are usually unaware of the implementation details and they will attempt to use the software in ways that were not anticipated by developers. The fact when a user of the software invoke any of the public methods, the class obliges blindly to the request, makes the software fragile face to users that are interacting naively with the system.

Adding a state machine permits to separate the request from users of the class, from the actions executed by the class. This means, that instead of blindly responding to every invocation of a method, now a class simply accept function calls as a "*request*" from the user. This request is translated into an input to the state machine of the class, and depending on the current state, it results in the execution of different types of actions. If in the current state all the conditions for satisfying the user's request are met, then the state machine will trigger the execution of the action requested by the user. If the current state does not satisfies the necessary conditions then the state machine may reply with an error notification via an Event (see Chapter 7) or simply ignore the request.

Only the methods that correspond to "requests" are made public, while the methods that actually execute actions are made private and can only be called by the state machine. By preventing direct access to the methods that trigger actions, the state machine layer, prevent the users from making wrong use of the class.

6.2.3 Robustness to misuse

Despite the fact that the state machine prevents access to the execution methods, it is still possible to misuse the class by sending incoherent requests. The protection against this potential misuse is to carefully program the logic of the state machine in such a way that, the states in which specific requests are acceptable, are well defined. For example, if a class offers a set of five potential requests that can be called by users, but still requires those requests to be made in a particular order, then the set of states and transitions in the state machine must model the appropriate order in which the requests are valid. In this way, when a request is received at a moment when the state machine is not ready for it, then the transitions will result in simply ignoring the request and sending an error notification back to the user. In no case, the state machine will go into an erroneous state.

6.2.4 Managing Complexity

It is clear very early in the development cycle that simplicity of the classes API is a must for supporting robustness and reliability. In the context of surgical guidance, we must consider flexibility and abundance of features to be undesirable, because each one of them brings more opportunities for thing to go wrong during the surgical intervention.

The introduction of state machines make possible to manage complexity by forcing the logic of the application to become explicit. Unfortunately, programmers accustomed to traditional techniques might easily develop the wrong impression that state machines introduce complexity in an application. This happens because by making explicit the logic of the application, programmers are made aware for the first time of all the intricacies of the code. Previously it was easy for programmers to deny or to ignore the existance of these details because they were spread all across the code in the form of *latent logic* in hundreds of "if" statements here and there, as well as in the conditionals of "for", "while", and "switch" statements.

Once the complexity is made explicit and it becomes visible in a single centralized location of the class, then it is possible to have a realistic approach for managing it and keeping it under control. Classes based on state machines typically offer a very small set of possible requests, as well as a small set of possible actions to be performed. A look at the transition matrix of a state machine gives a very direct and realistic evaluation of the level of complexity contained in a particular class.

6.2.5 Traceability

When a class is exercised at run-time, the use of an internal state machine makes very easy to trace and understand the logic path followed by the class during its execution. In traditional programming techniques developers are forced to do the equivalent job by keeping track of the values of any variables that indirectly maintain the state of the class.

The implementation of state machines in IGSTK classes facilitates to send to the `igstk::Logger` information about the intputs, states and transitions taken at every step of its execution. Such information facilitates the task of testing and debugging the class during development, and also make easier to monitor the behavior of a full IGS application after it has been deployed.

6.2.6 Suitability for Testing

From the point of view of testing, State Machines make possible to exercise full coverage, not only in the sense of number of lines executed during the testing cycle, but also in the sense of all possible execution paths of the code, at least at a single-class level.

The goal of reaching 100% code coverage becomes realistic when state machines are used. Otherwise, the logic of the program diffuses over many lines of code and it is very hard to attempt to follow every possible combination of circumstances. When state machines are used, performing a 100% code coverage becomes a matter of exercising all possible transition in the transtion matrix.

The significance of the testing suite of an application is only as large as the code coverage associated with it. For instance, it is of no use to claim that an application passes "all" its tests, if the tests exercise only 15% of the application's code. In honest terms such testing suite should be claiming that the applications passed only 15% of the tests, and the remaining test have not even been performed.

Test suites with low code coverage have the dangerous effect of providing a false sense of security. Since developers get the impression that the software is in a good state, while in reality they simply do not know the full state of the software they are developing.

6.2.7 Consistent Documentation

State machines in IGSTK are capable of exporting its internal programming in formats that are suitable for generating diagrams. In particular, they can export their list of possible states, possible inputs, and the content of the transition matrix. This funcionality makes possible to automatically generate "State Diagrams" [3] that correspond to what is *actually* encoded in the state machine as opposed to what the developers *intended* to encode. Thanks to this feature, developers can look at this diagrams and use them as support documentation for understanding the behavior of the code.

The state diagrams can be exported in the format of the "dot" application from Graphviz[1], and in LTSA format[2].

6.3 Implementation

A generic State Machine class is available in the toolkit and provides the abstraction of the set of states, the set of inputs and the set of transitions. Each IGSTK component instantiate internally its own state machine and at construction time programs the full behavior of the State Machine. This organization makes possible to anticipate how the classes will work when their methods are invoked in any order. It is rare to find object oriented classes that will behave correctly or at least without run-time failures when their methods are invoked in random order.

6.3.1 C++ Features

A number of C++-Language features have been used in order to enforce the safety and integrity of the State Machine. For example, the methods that actually perform actions are all declared private and can only be invoked by the State Machine itself. Encapsulation and enforcement of const-correctness are also used at great lengths in order to reduce the risks of misusing the code.

Transitions in a State Machine result in actions being taken. Some State Machine paradigms execute the actions when leaving the old state, while some others do it when entering the new state.

[1]http://www.graphviz.org
[2] http://www.doc.ic.ac.uk/ltsa

6.3.2 Integration inside a Class

State machines are hiddend inside IGSTK classes. Every major class in the toolkits has an internal state machine that manages the logic of the class. The State Machine is templated over the type of its owner class with the purpose of making its types and states to be unique for this particular class. In this way, there is no possibility of erroneously sending to State Machine of type "A" the inputs that were intended for state machine of type "B". The type checking functionality of the C++ compiler will help to ensure that such errors will be detected during compilation time and will never be possible at run time.

Once the state machine is instantiated as a type. It defines a number of traits, in particular, those are types for representing States, Inputs and Transitions.

The class owning the state machine passes its "this" pointer to the StateMachine so that the state machine can invoke methods of the owner class. Additionaly the instantiated state machine type is declared as `friend` of the owner class, so that `private` methods of the owner class can be called by the state machine.

With this integration, it is expected that the owner class will have methods such as

- `public:  RequestActionX()`
- `private:  ActionXProcessing()`

Where *ActionX* can be replaced with the actual description of the action to be executed. All the *Request* methods are available in the public section of the owner class, while all the *Processing* methods are sequestered in the private section of the owner class.

The *Request* methods will analyze the user's petition and will translate it into an input that will be passed to the internal state machine. Depending on its state, the state machine will decide which one, if any, of the owner's class *Processing* methods to call as the action to be performed during its state transition.

6.3.3 State Machine API

The state machine offers to its owner class a set of public methods intended for programming, executing and querying the logic of the state machine.

The following method are used for programming the state machine

- `AddState()`
- `AddInput()`
- `AddTransition()`
- `SetInitialState()`

- `SetReadyToRun()`

These methods must be called in the constructor of the IGSTK class that owns the state machine. In order to simplify the writing of the code and enforcing style consistecy, a set equivalent C++ macro were defined. They are:

- `igstkAddStateMacro()`
- `igstkAddInputMacro()`
- `igstkAddTransitionMacro()`
- `igstkSetInitialStateMacro()`

The following methods are used for passing an input to the state machine and for triggering the execution of the transition.

- `PushInput()`
- `PushInputBoolean()`
- `ProcessInputs()`

Inputs passes to the state machine with the PushInput and PushInputBoolean methods are stored internally in a queue and they are later processed, in the same order of insertion, by the Process-Inputs methods. Pushing inputs in the queue does not trigger the execution of the state machine. Again, for enforcing the consistency of coding style, a macro was created for pushing inputs into the state machine, this is the `igstkPushInputMacro`.

The following public methods are intended to export the description of the state machine to Graphviz, LTSA and SCXML respectively.

- `ExportDescription()`
- `ExportDescriptionToLTS()`
- `ExportDescriptionToSCXML()`

6.4 Usage

State machines are not exposed to the users of the IGSTK toolkit. They have been hidden in the private section of the classes. Users of the toolkit communicate with the state machines only via “Requests” methods in the public section of the class API.

In order to maintain the state machine as the only repository of logic in the code, it is fundamental to ensure its full encapsulation inside the class that uses it. For instance, the state of the state machine is never exposed, not even to the class itself, in order to prevent developers from adding code that will test the state of the state machine and use it in conditional statements (e.g. "if"). Such attempts would have resulted in a leakage of the logic from the state machine into the uncontrolled section of the class code.

CHAPTER

SEVEN

Events

"You only need to know things on a need-to-know basis."
"Yes, Prime Minister", movie, 1986.

7.1 General Background

In the context of Object Oriented Programming, *"Encapsulation"* is known as the property of classes for not exposing the details of their implementation to other classes that interact with them. Encapsulation of medium size components is one of the techniques used in IGSTK for addressing the concern for improving safety and robustness. The effectiveness of Encapsulation is further improved when combined with decoupling among the different software components.

In practical terms, this means that one class should need to know as little as possible about any other class. Of course the ideal case is when a class does not need to know anything about the other classes that will interact with it. This seems to be contradictory with the need of classes for passing information between them during their interactions. The use of the Event and Observer pattern [6, 3] makes possible to solve this contradiction by establishing a bride for passing information between two objects while still preserving their encapsulation and their full decoupling.

The use of Events is largely widespread among software packages. They are ubiquitous in GUI libraries, from the X11 Windows system to the more recent Microsoft Foundation Classes MFC and Qt libraries. Events and observers have also been used successfully in the Visualization Toolkit VTK and the Insight Toolkit ITK.

7.2 Motivation

There are multiple motivations for using Events in IGSTK. They are summarized in the following list.

- Decoupling of collaborating classes.
- Containment of API modifications.
- Improvement of code reuse.

Decoupling classes that need to collaborate one with another is probably the strongest motivation in the context of safety-critical applications for which IGSTK was designed. The principle of decoupling is that the less one class “A” knows about the internal implementation of another class “ B”, the less this first class depends on assumptions about “B”, and therefore, the more robust it will be to the eventual changes of “B”s implementation. Decoupling also makes easier to perform unit testing of individual classes, by making possible to focus on the contractual interface of the class, as opposed to consider specific features of the collaborating classes.

Containment of API modifications refers to the continuous maintenance of software. It is a common mistake to assume that software is written once and that the task is completed. In practice, software is continuously evolving, according to the needs of the user community. Software that is not designed to facilitate this evolution become easily degraded when changes start to happen. One of the aspects that can commonly change in a object oriented programming library is the public interface of specific classes. This means that the methods of classes may be renamed, removed, or they may change the type of their arguments. When such changes take place, all the other pieces of software that were calling these functions must be modified accordingly. Failing to update all other calls to those methods is a very common source of bugs, with the detrimental aggravation that it is cumulative over time. When method calls are replaced with Events as the mechanism for passing information between classes, then the API modifications become more localized and do not affect many other pieces of software.

Code reuse is also improved by introducing Events and Observers as the mechanism for passing information between classes. In particular, it becomes easier to connect two classes to make them work together, because they only need to know about the types of the data passed between them, instead of having to know about each other.

7.3 Implementation

7.3.1 Relationship with ITK

Events are implemented in IGSTK following the model of the Insight Toolkit ITK. The Observers of such events are also closely related to the ITK Command class. Figure 7.1 illustrates these relationships.

ITK Events are implemented as a class hierarchy. The `itk::AnyEvent` class is at the base of this hierarchy while the `itk::UserEvent` is defined as one of the branches. The purpose of the `itk::UserEvent` is to provide ITK users with a place to attach their own events hierarchy. This ITK feature is exploited in IGSTK by defining the `igstk::IGSTKEvent` as the base class for all events used in IGSTK, and making it to derive from the `itk::UserEvent`.

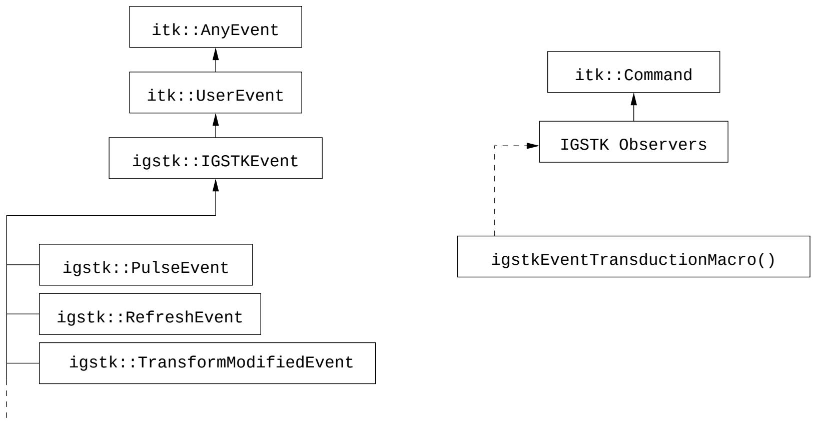


Figure 7.1: Events Class Hierarchy.

Figure 7.1 shows a small subset of the many events that derive from the IGSTKEvent. In particular the `igstk::PulseEvent` used by the `igstk::PulseGenerator` class, the `igstk:RefreshEvent` used by the `igstk::View` class, and the `igstk::TransformModifiedEvent` used by the `igstk::SpatialObject` class.

7.3.2 Events with Payload

Two main categories of events have been defined in IGSTK. The first category is composed of events that only carry information in their type. The second category is composed of events that carry information in their type and also carry instances of other objects as payload. In this regard IGSTK Events behave similar to C++ exceptions.

Events that only have a type are used to notify other classes about the occurrence of a specific incident, for example the expiration time of a timer, or a failure to open a file. Events that carry payload have the additional capability of transmitting detailed data between two classes without requiring these classes to know about the other. In order to facilitate the creation of events with payload, a set of Macros and Templated classes are available in the files.

- `IGSTK/Source/igstkMacro.h`
- `IGSTK/Source/igstkEvents.h`

The most important of these macros is the `igstkEventMacro` that creates a new event class in a single line of code by just specifying the name of the new event and its superclass. The `igstkLoadedEventMacro` creates a new event with payload by specifying the event name, the superclass and the type of the payload. The `igstkLoadedObjectEventMacro` is intended for the case where the payload carried by the event derives from the `igstk::Object` and therefore uses SmartPointers. These Macros simplify the code and enforce uniformity across the entire toolkit.

Events with payload created by the `igstkLoadedEventMacro` have the following standard methods

- `PayloadType &Get() const`
- `void Set( const PayloadType &)`

while the events created by the `igstkLoadedObjectEventMacro` have the following standard methods

- `PayloadType * Get() const`
- `void Set( PayloadType * )`

and internally store a Smart Pointer to a `PayloadType` object.

Events with payload are particuarly important for the implementation of the Request/Observe pattern described in Section 7.4.2.

7.3.3 Events and State Machines

Events can also be integrated in the behavior of State Machines. This is facilitated by the `igstkEventTransductionMacro` that when used inside an IGSTK class will create in it an observer as a member variable and will also defined methods that will make possible for that class to connect its internal observer to the class that is expected to generate the event.

When the event is received at run-time, the code in the transduction macro translates the incoming event into an input to the State Machine in the class that owns the observer. In this way, events can be managed as equivalent to State Machine inputs, so that their message-passing capabilities are complemented with the decision-making capabilities of the State Machines.

Figure 7.2 illustrates the mechanims of the transduction macro. Class "B" is intended to observe events "E" send from class "A". Upon reception of any event "E", class "B" is expected to perform an action. When the code of the Transduction macro is put into class "B", it will instantiate an Observer and it will create the `CallbackEventInput` method using the name of Event "E" and the name of the State Machine Input to which we want to translate the event; so that the full method name is still unique. This new method is a member method of class "B" and it will be called by the Observer whenever it receives an Event of type "E". The Callback-EventInput method passes a specific input to the internal State Machine of class "B" by calling its `PushInput` method. The State Machine will use its current state and the current input in order to decide on the next transition to perform. The selected transition may trigger the execution of an action method that, in IGSTK by convention, is a method named with the sufix *Processing*. Such methods are private and are intended to be called only by the State Machine. In this way, an Event "E" is seamlessly integrated as equivalent to an input to the State Machine of class "B".

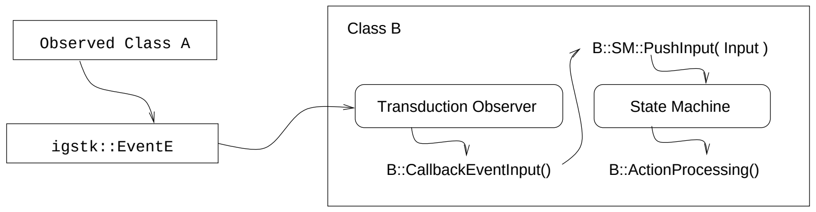


Figure 7.2: Events/Input Transduction.

7.3.4 Observers

IGSTK does not have observers *per se*. Instead, it relies on deriving this functionality from the ITK Command class. Given that IGSTK Events derive from ITK events, it is possible to Observe IGSTK events by using classes that derive from the ITK Command class. IGSTK classes use internally the following options for defining observers based on ITK classes

- Creating a new class that derives from `itk::Command`
- Using the `itk::SimpleMemberCommand<>`
- Using the `itk::ReceptorMemberCommand<>`

The first option is used when a lot of flexibility is required by the IGSTK class internal implementation. The second option is used in the `igstk::Tracker` and `igstk::View` classes in order to observe the pulse events generated by their respective internal `igstk::PulseGenerator` classes. The last option is used in the State Machine and in the transduction macros discussed in Section 7.3.3.

These details are only of interest for those developers that want to modify or maintain the source code of IGSTK. Developers of IGS applications based on IGSTK only need to be concerned about defining Observers when they query information from particular IGSTK classes, mostly for the purpose of monitoring the behavior of those classes. The communication between IGS Application code and the toolkit classes is described more in detail in Section 7.4.2 where the use of the *Request/Observe* pattern is explained.

7.4 Usage

7.4.1 Internal Usage

Most of the time, communications between IGSTK classes is done internally through events. Developers of IGS applications do not need, in general, to deal with the events that are being emitted from IGSTK classes. It is useful however to know how information is flowing between

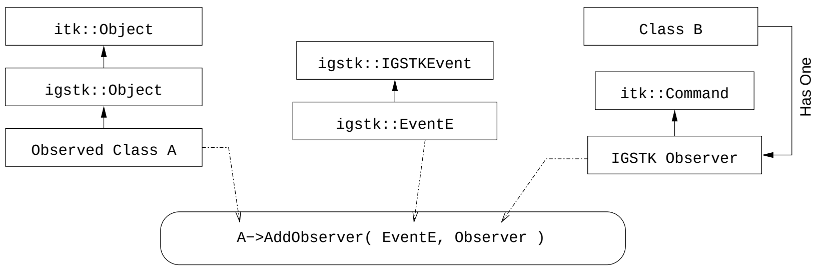


Figure 7.3: Events Usage.

the different IGSTK components. We present here some details about how the mechanism works.

First of all, in order to pass an Event "E" from a class "A" to a class "B" the following requirements must be satisfied

- Class "A" must derive from `igstk::Object`.
- The Event "E" must derive from the `IGSTKEvent` class.
- Class "B" must have defined an observer deriving from the `itk::Command`.
- Class "B" must add its observer to Class "A" using the `AddObserver` method of class "A" and specifying that is interested in Event "E" or one of its super-classes.

Figure 7.3 illustrates these requirements. The left side the Figure shows the hierarchy of the class that is being observed, class "A". This class must derive directly or indirectly from the `igstk::Object`. It is from this class that the Event "E" will be send. The middle of the Figure illustrates the hierarchy of the Event "E", which must derive directly or indirectly from the `IGSTKEvent`. The right side of the Figure illustrates the collaboration between class "B" that wants to know about Event "E" and its helper class, the IGSTK Observer, that itself derives from the `itk::Command`. Class "B" must have as a member variable an IGSTK Observer. This observer is programmed to invoke a member method of "B" whenever an Event "E" is sent form class "A". Finally, the bottom of the figure shows the invocation to the `AddObserver()` method from class "A". This invocation informs class "A" that the Observer is interested on being notified whenever the Event "E" occurs. Since the IGSTK Observer is a generic class, class "A" does not need to know about the existence of class "B", and class "B" does not need to know about the existence of class "A". In this way, decoupling between "A" and "B" is completely achieved. Of course, the application developer will be responsible for calling the `AddObserver` method from the application's code.

7.4.2 External Usage: The Request/Observe Pattern

The motivation for using the *Request/Observe* method in IGSTK is to improve the robustness and safety of the toolkit as well as the IGS applications that are based on it.

The Request/Observe pattern is intended to deal with the scenario where an IGSTK class is capable of returning a data object, for instance a Transform, but the value of this data object is only valid in some of the states of the class and not in others. Typical solutions used for this scenario in other software packages are to have a “Get()” method that returns the data object and combine it with one of the two following options

- Define a specific value of the data object to indicate when the object is invalid
- Add a boolean “IsValid” method that indicates when the data object is valid
- Return the data object by reference simultaneously with a boolean verification

In the context of a safety-critical applications, the first option has the drawback that every time that the “Get()” method is called by the customer of the class, they have to check whether the data object is valid or not. This results in code plagued with “if” conditions that are difficult to debug, hard to maintain and where it is very difficult to enforce testing with 100% code coverage. Avoiding proliferation of “if” conditions was in fact one of the main reasons for introducing State Machines into the toolkit.

The second method is also known as a *pre-condition* and it is common in *contract-based programming*. There are two drawbacks to this method. The first is that it relies on IGS applications developers to actually call the “IsValid()” method before calling the “Get()” method. In C++ there is no way to enforce that the “IsValid()” method must be called before the “Get()”. The second drawback is that in a multi-threaded environment there is no way to ensure that if the “IsValid()” method returned true at any given point, the value of the transform will not become invalid before the “Get()” method is called. Therefore this combination may result in the “Get()” method returning an invalid data object that is now not checked by the recipient code.

In the third option a single function will simultaneously return a boolean and the data object that may or may not be valid. The validity of the data object is indicated by the boolean that is simulaneouly returned by the function. This is equivalent to combining the “IsValid()” and “Get” functions into a single atomic operation. This option still have the drawback that the customer’s code need to be plagued with “if” conditions that are detrimental to the robustness and safety of the code.

As an alternative to these three typical options, IGSTK intoduced a Request/Observe pattern in which no “if” conditions are required when two pieces of code based on State Machines are used. The principle of the Request/Observe method is to split the transaction of information into a set of state, inputs and transitions in the State Machines of the two classes involved in the transaction of the data object. Figure 7.4 shows the sequence diagram of the transaction between two classes. On the left side of the figure, the “Customer” class is the one that need to receive the data object. On the right side of the figure, the “Provider” class is the one that produces the data object. The "Customer" in this case is a class that internally has already defined an Observer, presumably by using the `igstkEventTransductionMacro` discussed in Section 7.3.3.

Time progresses from top to bottom of the diagram. The first step on the interaction is for the "Customer" to connect its internal observers to the "Provider" class. Two types of events are expected in this transaction, therefore two Observers must be connected to the "Provider", one for each event. One of the events to be Observed is a event with payload that is capable of carrying the specific data object. The other event is the one that may notify the the data object is invalid. This connections of Observers are done only once regardless of how many times in the future the "Customer" may need to query the "Provider" for the data object.

The next step is for the "Customer" to call a method in the "Provider" in order to ask it for sending the data object if it is available. This is done in the form of a "RequestDataObject()" method. This action is presumably the result of an action triggered in the "Customer" by a transition of its internal State Machine. This is indicated in the diagram by the arc labeled as "State Transition X". The request is not answered immediately by the "Provider". Instead, it results in a specific input being passed to the internal State Machine of the "Provider". Depending on the current state of the "Provider", that input may result in different transtions and associated actions in the State Machine of the "Provider". Two typical occurences are illustrated in the diagram. The first is "State Transition A" where the "Provider" happens to be able to return a valid data object at this moment. In this case, the data object is loaded into an Event and send by using the InvokeEvent method. The second occurence is illustrated by "State Transition B", and corresponds to the case where the "Provider" does not have a valid data object at this point. In this case, an Event that indicate the non-validity of the requested value is send using the InvokeEvent method.

In the case of "State Transition A", when the "Provider" is capable of sending a valid data object, the invocation of the Event will trigger the Observer of the "Customer" who will translate this event into an input to its state machine. Depending on the current state of the "Customer" its State Machine may make a "State Trasition Y" with an associated action that is capable of processing the data brought by the Event, in such case the "Customer" will extract the payload by using the "Get" method of the event with payload that was discussed in Section 7.3.2. If by the time that the Event gets to the "Customer", its State Machine is in a state the can not process the data object, or is not interested in processing the data object anymore, then the data is simply ignored.

In the case of "State Transition B", when the "Provider" is unable of producing a valid data object and invokes an ErrorEvent, the Error event is received by the Observer in the "Customer" class and it is translated into an input to its State Machine. Depending on the current state of the "Customer" the State Machine will perform a transition that is the appropriate response to the lack of a valid data object.

As can be seen from this diagram, the Request/Observe pattern permits to have a very explicit transaction in which the reactions to both valid and invalid data objects are carefully analyzed. Since this mechanism directly takes cares of routing the responses of both classes the the condition in which the data object is invalid, it permits to ensure that none of the classes will be

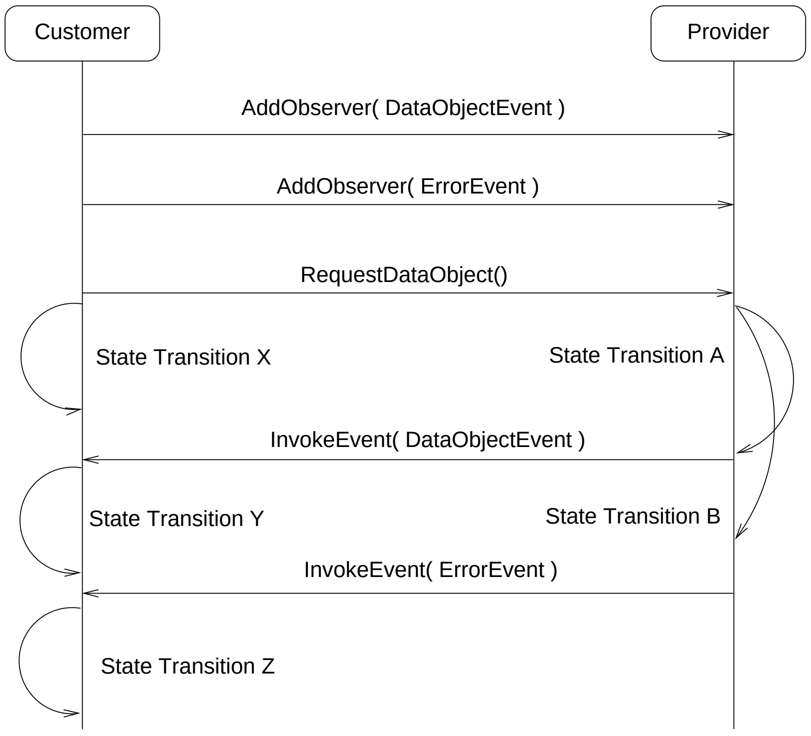


Figure 7.4: Request/Observe Pattern.

put in an invalid state during the transaction. This is a great advantge compared with the typical fragile approach in which developers expect a valid object to be returned, and consider the invalid case just as a rare event that will put the customer into an invalid condition. In IGSTK, by anticipating explicitly the potential error situations and wiring the appropriate responses in the logic of State Machines, a solid step is taken towards improving robustness and safety.

7.5 Conclusion

Events are an extremely valuable programming technique that is used in many different places in IGSTK. In order to take advantage of the strengths of Events and State Machines, IGS application developers should strive for designing their own applications based on the same principles. In that way the safety and robustness of IGSTK design will permeate the final complete IGS application.

Of course, attempting to introduce IGSTK code into an application with more traditional design will give the developers the false impression of IGSTK being utterly complex. That perception should be taken as an indication that the application developer is not building the software in a way that is compatible with the structure of IGSTK.

CHAPTER

EIGHT

Tracker

"The trouble with measurement is its seeming simplicity."
Unknown.

The goal of image-guided surgery is to provide the surgeon with the necessary tools to correlate features and locations in a medical image, with those same features and locations on (and inside of) the patient's body during surgery. This process can be purely visual, for example when a surgeon uses x-ray fluoroscopy to look inside the patient during the surgery, but the term "image-guided surgery" usually implies a process called stereotaxis, which involves three actions: 1) assigning a coordinate system to the image, 2) registering the patient to this coordinate system, so that each location in the image can be mapped to a location inside the patient, and 3) providing the surgeon with the means of getting to a particular location inside the patient. In IGSTK and in most other modern image-guided surgery systems, the latter is achieved by tracking the positions of the surgeon's tools and displaying the tools superimposed on the images of the patient, so that the surgeon can see the positions of the tools relative to the target location.

Tracking of the surgical tools is done by devices known as *tracking systems* (also *localizers* or *position measurement systems*). These devices measure the (x, y, z) coordinates of each tool as well as three angles to describe the tool orientation, and they report this information several times per second to make it possible to track the motion of the tools. Several models of tracking systems are available on the market. The ones that are supported by the current version of IGSTK are the NDI POLARIS family of optical tracking systems, the NDI AURORA magnetic tracking system, and the Ascension Flock of Birds magnetic tracking system.

8.1 The Role of the Tracker Component in IGSTK

The IGSTK Tracker component communicates with a tracking system to get position measurements, and it then makes these measurements available to other IGSTK components. Given the tracker component's straightforward role, you would expect its public interface to consist of only a small number of methods, and that is in fact the case. There is a fair degree of complexity

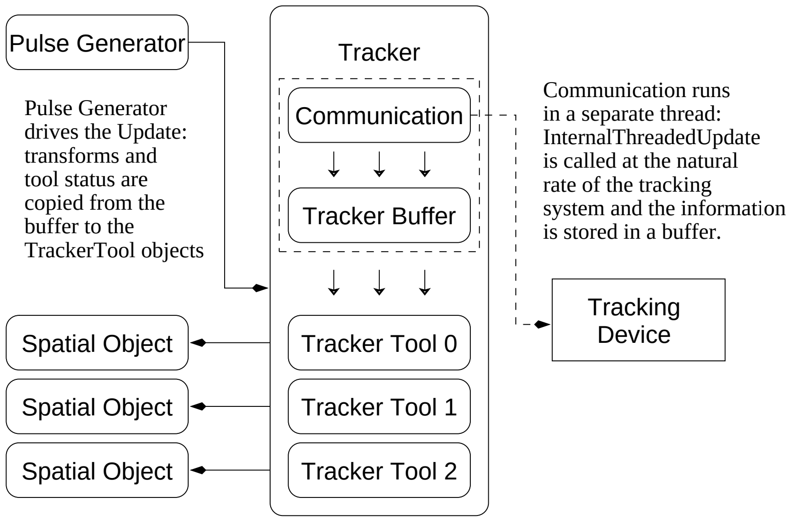

Pulse Generator
Pulse Generator drives the Update: transforms and tool status are copied from the buffer to the TrackerTool objects
Tracker
Communication
Tracker Buffer
Communication runs in a separate thread: InternalThreadedUpdate is called at the natural rate of the tracking system and the information is stored in a buffer.
Tracking Device
Spatial Object
Tracker Tool 0
Spatial Object
Tracker Tool 1
Spatial Object
Tracker Tool 2

Figure 8.1: The IGSTK Tracker Component.

in this component to handle accurate timing, data calibration, and recovery from hardware communication errors, but these things are hidden beneath the public interface as far as is possible.

8.2 Structure of the Tracking Component

The `igstk::Tracker` object houses a number of `igstk::TrackerTool` objects, one for each of the surgical tools that will potentially be tracked. Each of these TrackerTool objects store position and status information about a particular tool, just as the Tracker object contains status information for the tracking system as a whole.

It is important for the reader to understand the dichotomy that is present here: the Tracker and TrackerTool objects are part of the software of IGSTK, but the tracking system is a physical device located in the operating room, and the tracked tools are things that you can hold in your hand. The updating of the tool position and status information in the software occurs because there is a communication link such as a serial cable or network connection between the computer and the tracking system. The Tracker is the only IGSTK component that has direct access to the tracking system via this communication link.

The IGSTK software components run as a synchronous system, where each component executes in turn. When a component sends an event to another component, the other component controls the program execution until it passes control back to the originator. At the top level, actions are driven by `igstk::PulseGenerator` events , which are timer events generated in the application's main event loop. The tracking device (the physical device itself) has its own internal update cycle which is completely independent of anything going on in our software, therefore it was decided that the IGSTK tracker component should spawn a separate thread to communicate with the tracking device. This internal thread is not driven by a PulseGenerator, but instead listens constantly on the communication link for data records from the device, and collects data records at whichever rate they are sent by the device.

These data records are not used immediately as they are received, instead they are buffered until the application is ready for them, i.e. until the PulseGenerator generates a pulse to indicate that it is time for the tool positions within the IGSTK application to be updated. This ensures that the information only changes at times when other IGSTK components are expecting it to change.

8.2.1 Communication

Collecting data from the tracking system is one of the two fundamental duties of the Tracker (the other fundamental duty, of course, is relaying this data to the application). Early in the development if IGSTK, it was decided that communication between the tracker component and the tracking system should be handled by an `igstk::Communication` class that met specific requirements:

1. each port that is used by the application to communicate with a device will have a communication object that acts as a proxy for that port
2. the communication object must not freeze if the connection is broken; instead, it must report a suitable error
3. the data stream that flows through the port will be logged to create a complete record of of all communication that takes place
4. the data stream can be played back for testing purposes

The second requirement is important because most communication ports (including network sockets and RS232 serial ports) operate in *blocking* mode by default, which means that if the data stream is broken, then any attempt to read from the port will cause program execution to freeze until the connection is restored. This situation is unacceptable in a surgical application. The port is always switched to *non-blocking* mode by the communication object before any data is sent or received.

The third and forth requirements facilitate the testing of the tracker component and the tracking system. During the development of an application, the data stream must be examined repeatedly during the debugging process. Furthermore, if the tracking system suffers any faults or malfunctions, access to the data stream is very useful for diagnosis of the system.

Finally, the playback of a prerecorded data stream can be used to simulate the presence of a tracking device during testing. This allows components that depend on the Tracker component to be tested even in the absence of a tracking system or a human being to move the tools around.

This is fundamental to one of the cornerstones of IGSTK: the automated nightly testing of all components.

8.2.2 Threading

As stated in Section 8.2, the communication with the tracking system occurs in a dedicated thread. The purpose of the tracking thread is to maintain a constant link with the tracking system: the thread can respond to information from the tracking system while the application is busy with other things. The use of multi-threading is desirable for two reasons: safety and performance.

Safety

Many tracking systems have some method of signaling to the surgeon that certain events have occurred or that certain unsafe situations have arisen. These signals are usually verbal (the device will beep or will talk) or visual (an indicator light on a tool will blink). Some devices can even provide *haptic* (tactile) feedback, where the device will physically resist any the motion of the surgeon's hand which would lead to the cutting of sensitive tissues such as major nerves or arteries.

Any such feedback must be instantaneous to be effective. If the Tracker component had to wait until the next time its PulseGenerator fired before responding, there would be a delay of tens of milliseconds before the surgeon learned of a potentially urgent situation. Through the use of a tracking thread that is independent of the PulseGenerator, however, the Tracker component can provide feedback to the surgeon immediately.

Performance

The performance argument for using a tracking thread is, of course, not as fundamental as the safety argument. Nevertheless, preformance criteria such as frame rate and lag, which are so important to 3D video game enthusuasts, are of some consequence in image-guides surgery as well.

A tracking device uses its own internal clock to make measurements at regular intervals. It stands to reason that the Tracker component should collect these measurements at the rate at which the device takes them, and furthermore should collect each measurement as soon as possible after that measurement is taken.

A poor implementation of the Tracker component might work as follows: when the Tracker's PulseGenerator fires to indicate that a tool position measurement is to be forwarded to the other IGSTK components, the Tracker component will send a command to the tracking system to tell it to make a measurement, and will then wait patiently for the tracking system to send a send a measurement back.

In contrast, the way that the IGSTK Tracker has been implemented such that the tracking system

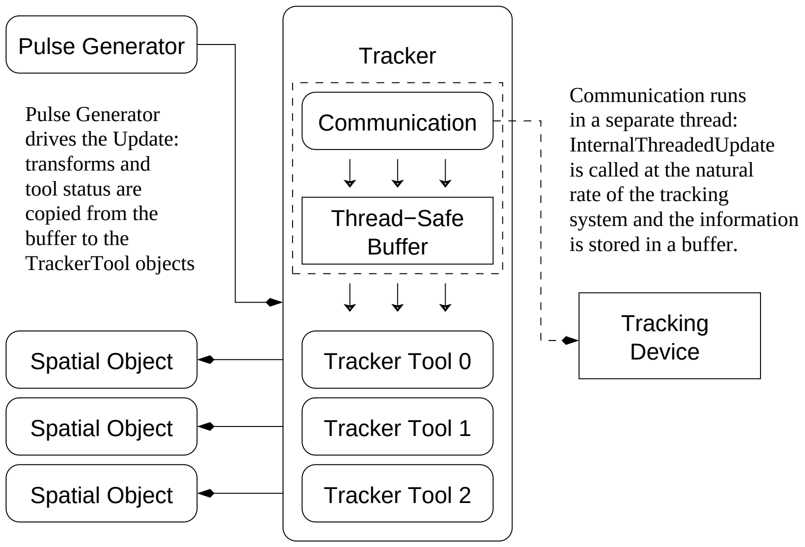

Pulse Generator
Pulse Generator drives the Update: transforms and tool status are copied from the buffer to the TrackerTool objects
Tracker
Communication
Thread−Safe Buffer
Communication runs in a separate thread: InternalThreadedUpdate is called at the natural rate of the tracking system and the information is stored in a buffer.
Tracking Device
Spatial Object
Spatial Object
Spatial Object
Tracker Tool 0
Tracker Tool 1
Tracker Tool 2

Figure 8.2: The TrackerBuffer object.

continuously streams position measurements to it, and it uses its extra thread to listen for each of these measurements and store them in a buffer as they arrive. Then, when the PulseGenerator sends a signal to forward a measurement to the other IGSTK components, the Tracker will simply take the most recently made measurement from the buffer and forward it along. The Tracker does not have to wait for the tracking system to make a new measurement.

8.2.3 Buffering

In the previous section, it was described how threading and buffering are used to improve the performance of IGSTK. The buffer has an additional purpose: it is also used to store a history of all position measurements that are made by the tracking system during the surgery.

When IGSTK displays a surgical scene with the positions of all of the surgical tools superimposed on the medical images, it is displaying where those tools were at a particular instant in time. This time is indicated by a timestamp, and it may either be the current time as given by the computer's real-time clock, or it may be some time in the past (for example, during a replay of past tool motions).

To facilitate the delivery of tool position measurements for any particular timestamp, the `igstk::TrackerBuffer` object (as displayed in Figure 8.2) stores the position history for each tool as a series of raw measurements. These measurements are ”raw” in the sense that they are uncalibrated and are not yet registered to the surgical coordinate system, and for each measurement there is a timestamp to indicate when the measurement was made. When the Tracker's

PulseGenerator signals that the TrackerTool positions are to be updated and made available to other IGSTK components, it searches the tool histories to find the measurements that are closest to the desired instant, performs any necessary calibration and coordinate transformation, and pushes the measurements to the TrackerTools.

Since the TrackerBuffer contains data that is shared between the tracking thread (within which the Tracker places the data into the buffer) and the main IGSTK execution thread (within which the Tracker extracts data from the buffer), the TrackerBuffer has a mutual-exclusion lock to ensure that the main thread does not attempt to read a data record that the tracker thread has only partially written.

8.2.4 Transforms and Timestamps

In the previous section is was mentioned that every position measurement has a timestamp associated with it. Both the position measurement and the timestamp are stored in an `igstk::Transform` object, which contains:

- a vector with three position coordinates
- a rotation versor that describes the orientation of the tool
- a timestamp that gives the time at which the measurements were made
- an expiration time after which the measurement will be considered invalid

Expiration of Transform objects is included as a safety precaution: if IGSTK is rendering a video frame that corresponds to a particular instant in time, then that video frame should only include tool position data that was measured before that instant in time, but not measured so far before as to no longer be valid.

The time interval during which a Transform is valid will depend on how fast the tool is moving, since the total distance the tool moves from its measured position during this time interval is equal to the product of the time interval and the tool velocity. If we want to be able to guarantee that the tool moves no more than 1 mm before the Transform expires, and if the tool velocity is approximately 10 mm per second, then the Transform should be set to expire after 0.1 seconds. It is crucial, however, not to set too short an expiry time, since the Transform must not expire before the next Transform is generated by the Tracker.

A final but important point to consider regarding timestamps is that there is latency of a few tens of milliseconds between the time when a position measurement is made by the tracking system, and the time when that measurement is received by the computer on which IGSTK is running. This latency will depend on the tracking system and on the conditions under which it is being used (for instance, on its internal measurement rate, the number of tools that it is tracking, and on the data transmission speed between the tracking system and the computer). The latency is not subtracted from the timestamps in the current version of IGSTK since performing this subtraction will not actually make the latency go away, the laws of physics will not permit the tool coordinates shown on the computer screen to be updated instantaneously with the motion of the tool held in the surgeon's hand. Latency becomes critical, however, when measurements are to be made simultaneously with different tracking systems (for instance with an optical and magnetic system): only after subtracting the latency of each system from the Transform timestamps for that system can you accurately compare the timestamps from the two systems.

8.2.5 Coordinate Transformations

So far, we have discussed position measurements as if they pass unmodified from the tracking system to the IGSTK components such as the spatial objects. In actuality, the following formula must be applied to the position data by the Tracker component before it is forwarded to other IGSTK components:

$$\mathrm{T} = \mathrm{M}_{\mathrm{reg}}\mathrm{T}_{\mathrm{ref}}^{-1}\mathrm{T}_{\mathrm{raw}}\mathrm{M}_{\mathrm{cal}} \tag{8.1}$$

Each M and T in this equation refers to a coordinate transformation, specifically to a rigid body transformation consisting of a rotation followed by a translation. The transformations are defined as follows:

T	tool transform that will be used by other IGSTK components
$\mathrm{T}_{\mathrm{raw}}$	raw transform data from the tracking system
$\mathrm{T}_{\mathrm{ref}}^{-1}$	raw transform data from a tracked reference (if present)
$\mathrm{M}_{\mathrm{reg}}$	patient registration transform
$\mathrm{M}_{\mathrm{cal}}$	tool calibration transform

The *reference* is stationary tool that provides a reference frame for the other tracked tools. A reference is used when the tracking system itself (the camera of an optical system or the transmitter of a magnetic system) is either some distance away from the patient or not fixed in position relative to the patient. The reference is usually affixed rigidly to the patient's anatomy, e.g. to the skull in the case of neurosurgery or to a spinal vertebra in the case of spinal surgery, to minimize any potential error.

The final two transformations, the registration and calibration transformations, are described in detail in Chapters 14 and 15. Briefly, the registration transformation is used to place the tool position measurements into the coordinate system of the image that is being used to guide the surgery, and the calibration transformation is used as a correction for the tip location and shaft orientation for a particular tool.

To see why Equation 8.1 holds, consider Figure 8.3. We know that the product of any closed loop of transformations must be identity, where "closed loop" means a series of successive transformations that eventually leads back to the initial frame of reference. Taking the directions of the arrows into account, and using the patient coordinate system as our initial frame of reference, we see that

$$\mathbf{I} = \mathrm{T}^{-1}\mathrm{M}_{\mathrm{reg}}\mathrm{T}_{\mathrm{ref}}^{-1}\mathrm{T}_{\mathrm{raw}}\mathrm{M}_{\mathrm{cal}} \tag{8.2}$$

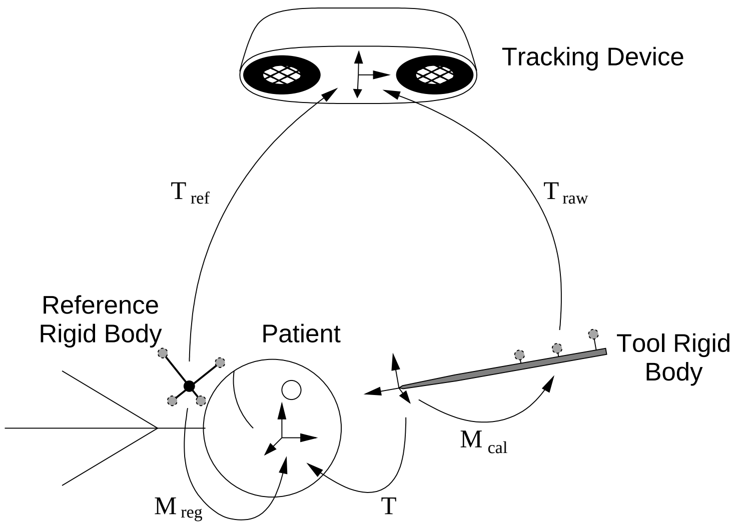


Figure 8.3: Tracker coordinate transformations.

The transformation that we want is T, which relates the position and orientation of the tool tip to the patient frame of reference, and we can get an equation for T by multiplying both sides of the above equation by T.

The best way to envision the set of transformations that are needed to obtain T is as follows:

1. start with an (x,y,z) location relative to the tool tip
2. apply $\mathrm{M}_{\mathrm{cal}}$ to find (x,y,z) relative to the tracked markers that are mounted on the tool
3. apply $\mathrm{T}_{\mathrm{raw}}$ to find (x,y,z) relative to the tracking camera
4. apply $\mathrm{T}_{\mathrm{ref}}^{-1}$ to find (x,y,z) relative to the tracked reference markers
5. apply $\mathrm{M}_{\mathrm{reg}}$ to find (x,y,z) relative to the registered patient coordinate system

Figure 8.3 is drawn for an optical tracking system, which uses a camera that tracks sets of 3 or 4 markers that define a “rigid body” which is a local frame of reference with respect to which the position of each marker is fixed at a known location. The same principles are also valid for magnetic transformations, except that the local frame of reference is defined with respect to the magnetic receiver coils.

8.3 State Machines

Figure 8.4 illustrates the State Machine Diagram of the `igstk::Tracker`.

8.3.1 States

Main states:

Idle initial state

CommunicationEstablished communication port opened to tracking system

ToolsActive tracking tools have been identified, ready to track

Tracking tracking system is sending position data

Transitional states

AttemptingToEstablishCommunication

AttemptingToCloseCommunication

AttemptingToActivateTools

AttemptingToTrack

AttemptingToStopTracking

AttemptingToUpdate

[State Machine Diagram]

Figure 8.5 illustrates the State Machine Diagram of the `igstk::TrackerTool`.

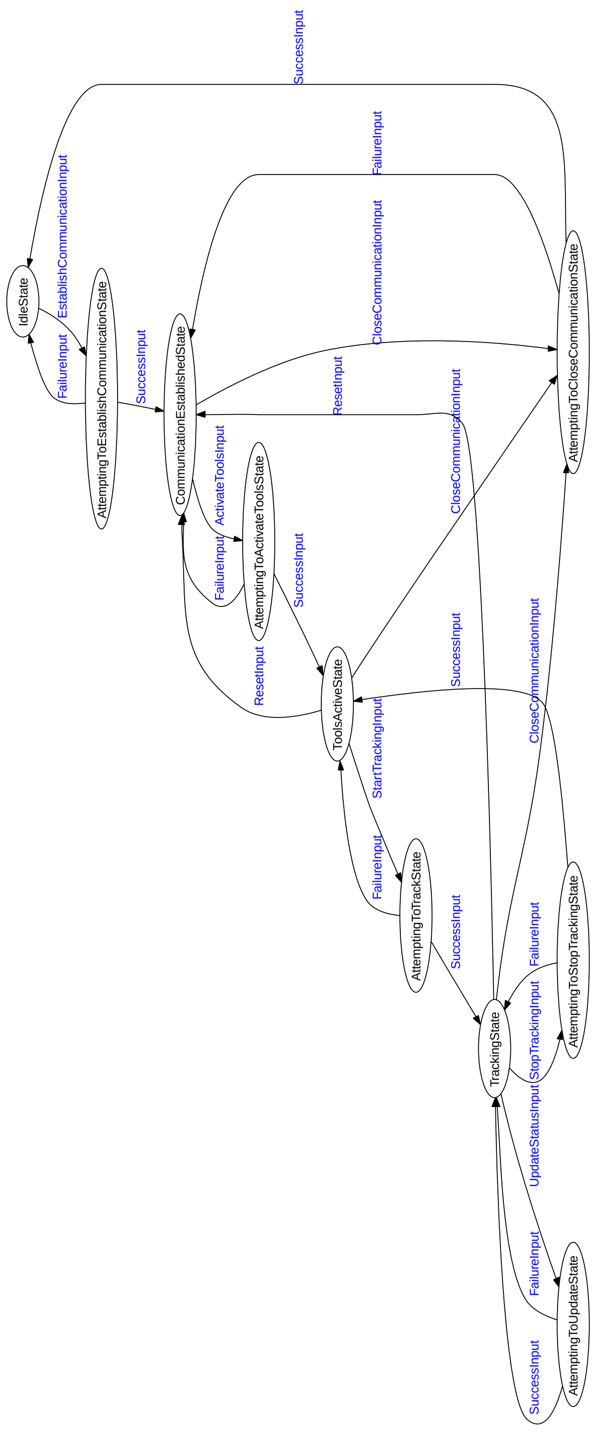


Figure 8.4: State Machine Diagram of the Tracker class.

Figure 8.5: State Machine Diagram of the TrackerTool class.

8.4 Component Interface

8.4.1 Interface Methods

8.4.2 Events

8.5 Supporting New Devices

8.5.1 Internal Interface

8.5.2 Command Interpreters

8.6 Special Topics

8.6.1 Simulation and Testing

8.7 Hazardous Conditions

Before we delve into the software, we should take some time to look at tracking systems and what can go wrong with them. This helps us to determine how the software might avoid, or at least cope with, some troublesome conditions that might arise during surgery.

8.7.1 Tracking Device Failure

The following hazards relating to device failure exist in the operating room:

1. the communication cable can become unplugged, resulting in loss of communication with the device
   (a) the user should be told that the computer cannot communicate with the tracker, and should be asked to check the cable
   (b) when the cable is reconnected, the tracker should automatically re-establish communication
2. a tool can become unplugged or damaged such that it no longer works

   (a) for some tools, such as the passive POLARIS tools, the tracker has no way of detecting that the tool is damaged
   (b) for other tools, the tracker should inform the user that the tool is unplugged or damaged
   (c) where possible the tracker should automatically re-detect the tool when it is plugged back in or replaced

3. the device can lose power when someone trips on the cord or mistakes its power cord for that of another piece of equipment
   (a) this will manifest itself similar to loss of communication, and has a similar remedy
4. the device can suffer hardware failure due to damage or age
   (a) this will manifest itself as a recurrence of one or more of the above failures
   (b) either the user or the device manufacturer will be responsible for addressing the failure

8.7.2 Loss of Accuracy

The following hazards can lead to inacurrate reporting by the tracker:

1. calibration problems, including magnetic field distortion
   (a) note that not all calibration problems can be detected
   (b) if the tracking system can detect these problems, they should be reported to the user in an unobtrusive manner
   (c) the application should refuse to make critical measurements if there are calibration problems
2. bent tools
   (a) if a tool is easily bent, then the application should request that the user test the tool before use
   (b) for example, the user could touch the tip of the tool to a known reference point so that the application can check whether it is bent
3. shifting of reference
   (a) if the reference moves relative to the patient, the application cannot detect this shift without assistance from the user
   (b) before taking critical measurements, the user should be asked to touch known anatomical landmarks to assess the accuracy
      i. if the accuracy is unsuitable, the user should be asked to repeat the reference registration if possible
      ii. in many cases, repeating the registration is not possible and image guidance may have to be abandoned if the reference shifts

CHAPTER

NINE

Spatial Objects

9.1 Introduction

SpatialObjects define a common data structure for objects in IGSTK.

The SpatialObject class hierarchy provides a consistent API for querying, manipulating, and interconnecting objects in physical space. The base SpatialObject class encapsulates an ITK spatial object; however, only functionalities that are essential for Image-Guided Surgey applications are exposed to the user.

A SpatialObject is a data structure describing the geometry of an object. Moreover, each SpatialObject contains an internal transformation that defines is location and orientation in space.

In this chapter, we first describe how SpatialObjects can be grouped in a hierarchical manner to form a tree of objects. Second, we detail the functionalities of each object. Finally we show how to read SpatialObjects from disk.

9.2 SpatialObject Hierarchy

The source code for this section can be found in the file
`Examples/SpatialObjects/SpatialObjectHierarchy.cxx`.

This example describes how to group the `igstk::SpatialObject`s to form a hierarchy of objects and also illustrates their creation and how to manipulate them.

The first part of the example makes use of the `igstk::EllipsoidObject`. The second part uses the `igstk::GroupObject`. Let's start by including the appropriate header files.

```
#include "igstkEllipsoidObject.h"
#include "igstkGroupObject.h"
```

First, we create two spheres using the `igstk::EllipsoidObject` class. They are created using smart pointers.

```
typedef igstk::EllipsoidObject SpatialObjectType;

SpatialObjectType::Pointer sphere1 = SpatialObjectType ::New();
SpatialObjectType::Pointer sphere2 = SpatialObjectType ::New();
```

We then add the second sphere to the first one by using the `RequestAddObject()` method. As a result `sphere2` becomes a child of `sphere1`.

```
sphere1->RequestAddObject(sphere2);
```

A child object can be retreive from its parent by using the `GetObject()` function.

```
sphere1->GetObject(0)->Print(std::cout);
```

Next, we assign a transformation to the object. We first create a transformation and set the translation vector to be 10*mm* in each direction, with an error value of 0.001*mm* and a validity time of 10*ms*. Second We assign the transform to the object via the `RequestSetTransform()` function.

```
igstk::Transform transform;
transform.SetTranslation(10,0.001,10);
sphere1->RequestSetTransform(transform);
```

Then, in order to retreive the transformation we setup an observer.

```
  igstkObserverMacro(Transform,
  ::igstk::TransformModifiedEvent,::igstk::Transform)
```

We then add the observer to the object using the `AddObserver()` command.

```
TransformObserver::Pointer transformObserver
                         = TransformObserver::New();
sphere1->AddObserver( ::igstk::TransformModifiedEvent(), transformObserver );
```

Then, we request the transform using the `RequestGetTransform()`.

```
sphere1->RequestGetTransform();
if( !transformObserver->GotTransform() )
  {
  std::cerr << "Sphere1 did not returned a Transform event" << std::endl;
  return EXIT_FAILURE;
  }

igstk::Transform  transform2 = transformObserver->GetTransform();
```

Next, we introduce the `igstk::GroupObject`. The GroupObject class derives from `igstk::SpatialObject` and acts as an empty container used for grouping objects together.

First, we declare a new group using standard type definition and smart pointers.

```
  typedef igstk::GroupObject GroupType;
  GroupType::Pointer group = GroupType::New();
```

Since the igstk::GroupObject derives from SpatialObject, we can use the `RequestAddObject()` function to add object into the group. For instance we group `sphere1` and the newly created `sphere3` together.

```
  group->RequestAddObject(sphere1);
  SpatialObjectType::Pointer sphere3 = SpatialObjectType::New();
  group->RequestAddObject(sphere3);
```

We can request the number of objects in the group using the `GetNumberOfObjects()` function. Since `sphere1` has a child, `sphere2`, there are actually three objects in the group.

```
  std::cout << "Number of object in my group: "
            << group->GetNumberOfObjects() << std::endl;
```

9.3 Common Objects

In this section we detail the different SpatialObjects present in IGSTK.

9.3.1 AxesObject

The source code for this section can be found in the file
`Examples/SpatialObjects/AxesObject.cxx`.

This example describes how to use the `igstk::AxesObject`. This class defines a 3-dimensional coordinate system in space. It is intended for providing a visual reference of the orientation of space in the context of the scene.

First we include the appropriate header file.

```
#include "igstkAxesObject.h"
```

We then declare the object using smart pointers.

```
  typedef igstk::AxesObject AxesObjectType;
  AxesObjectType::Pointer axes = AxesObjectType ::New();
```

The size of each axis can be set using the `SetSize()` function.

```
  double sizex = 10;
  double sizey = 20;
  double sizez = 30;
  axes->SetSize(sizex,sizey,sizez);
```

In case we need to retreive the length of the axes we can use the `GetSize()` functions.

```
  std::cout << "SizeX is: " << axes->GetSizeX() << std::endl;
  std::cout << "SizeY is: " << axes->GetSizeY() << std::endl;
  std::cout << "SizeZ is: " << axes->GetSizeZ() << std::endl;
```

9.3.2 BoxObject

The source code for this section can be found in the file
`Examples/SpatialObjects/BoxObject.cxx`.

The `igstk::BoxObject` represents an hexahedron in space.

We include the appropriate header.

```
#include "igstkBoxObject.h"
```

First we declare the box using standard smart pointers.

```
  typedef igstk::BoxObject BoxObjectType;
  BoxObjectType::Pointer box = BoxObjectType ::New();
```

Then we set the size of the box in each dimension by using the `SetSize()` function.

```
  double sizex = 10;
  double sizey = 20;
  double sizez = 30;
  box->SetSize(sizex,sizey,sizez);
```

If one needs to retreive the size of the box this can be done using the `GetSize()` function.

```
  std::cout << "SizeX is: " << box->GetSizeX() << std::endl;
  std::cout << "SizeY is: " << box->GetSizeY() << std::endl;
  std::cout << "SizeZ is: " << box->GetSizeZ() << std::endl;
```

9.3.3 ConeObject

The source code for this section can be found in the file
`Examples/SpatialObjects/ConeObject.cxx`.

As the name of class indicates, the `igstk::ConeObject` represents a cone in space.

First we include the header file.

```
#include "igstkConeObject.h"
```

We then declare the cone using standard smart pointers.

```
  typedef igstk::ConeObject ConeObjectType;
  ConeObjectType::Pointer cone = ConeObjectType ::New();
```

The `igstk::ConeObject` has two internal parameters, its radius and its height expressed in *mm*. The radius represents the radius of base of the cone. These two parameters can be set using `SetRadius()` and `SetHeight()` respectively.

```
  cone->SetRadius(10.0);
  cone->SetHeight(20.0);
```

9.3.4 CylinderObject

The source code for this section can be found in the file
`Examples/SpatialObjects/CylinderObject.cxx`.

The `igstk::CylinderObject` represents a cylinder in space.

Let's start by including the appropriate header.

```
#include "igstkCylinderObject.h"
```

First we declare the cylinder using standard smart pointers.

```
  typedef igstk::CylinderObject CylinderObjectType;
  CylinderObjectType::Pointer cylinder = CylinderObjectType ::New();
```

The `igstk::CylinderObject` has two parameters, its radius and its height expressed in *mm*. These two parameters can be set using `SetRadius()` and `SetHeight()` respectively.

```
  cylinder->SetRadius(10.0);
  cylinder->SetHeight(20.0);
```

9.3.5 EllipsoidObject

The source code for this section can be found in the file
`Examples/SpatialObjects/EllipsoidObject.cxx`.

The `igstk::EllipsoidObject` represents an ellipsoid in space.

Let's start by including the appropriate header.

```
#include "igstkEllipsoidObject.h"
```

First we declare the ellipsoid using standard smart pointers.

```
  typedef igstk::EllipsoidObject EllipsoidObjectType;
  EllipsoidObjectType::Pointer ellipsoid = EllipsoidObjectType ::New();
```

The radius of the ellipse can be adjusted in each dimension using the two `SetRadius()` functions. The easiest way is to use the `SetRadius(double,double,double)` function.

```
  ellipsoid->SetRadius(10,20,30);
```

However, in some cases, the uses of an array might be appropriate The array is defined using standard type definition, then passed to the ellipsoid as follow.

```
  typedef EllipsoidObjectType::ArrayType   ArrayType;
  ArrayType radii;
  radii[0] = 10;
  radii[1] = 20;
  radii[2] = 30;
  ellipsoid->SetRadius(radii);
```

9.3.6 ImageObjects

The source code for this section can be found in the file
`Examples/SpatialObjects/ImageObjects.cxx`.

In this example we show the main features of the ImageObject classes. IGSTK implements on class per modality, CT, MR and US.

```
#include "igstkCTImageSpatialObject.h"
#include "igstkMRImageSpatialObject.h"
#include "igstkUSImageObject.h"
```

First we declare an empty CT image using smart pointers

```
  typedef igstk::CTImageSpatialObject CTImageSpatialObject;
  CTImageSpatialObject::Pointer ctImage = CTImageSpatialObject ::New();
```

Then, for a given point in physical space, we can ask if this particular point is inside (or outside) the image using the `IsInside()` function.

```
typedef CTImageSpatialObject::PointType PointType;
PointType pt;
pt[0] = 10;
pt[1] = 10;
pt[2] = 10;
if(ctImage->IsInside(pt))
  {
  std::cout << "The point " << pt
            << " is inside the image" << std::endl;
  }
else
  {
  std::cout << "The point " << pt
            << " is outside the image" << std::endl;
  }
```

If the point is inside the image, we can convert the physical point into an index or a continuous index in the image reference frame. This is achieve using the `TransformPhysicalPointToIndex()` and `TransformPhysicalPointToContinuousIndex()` functions respectively.

```
if(ctImage->IsInside(pt))
  {
  typedef CTImageSpatialObject::IndexType IndexType;
  IndexType index;
  ctImage->TransformPhysicalPointToIndex (pt , index );

  std::cout << "Index is = " << index << std::endl;

  typedef CTImageSpatialObject::ContinuousIndexType ContinuousIndexType;
  ContinuousIndexType cindex;
  ctImage->TransformPhysicalPointToContinuousIndex (pt, cindex);
  std::cout << "Continuous index is = " << cindex << std::endl;
  }
```

We can also check if the image is empty by using the `IsEmpty()` function.

```
if(ctImage->IsEmpty())
  {
  std::cout << "The image is empty" << std::endl;
  }
else
  {
  std::cout << "The image has some non black pixel" << std::endl;
  }
```

9.3.7 MeshObject

The source code for this section can be found in the file
`Examples/SpatialObjects/MeshObject.cxx`.

This example describes how to use the `igstk::MeshObject` which implements a 3-dimensional mesh structure. The mesh class provides an API to perform operations on points and cell. Typically points and cells are created, with the cells referring to their defining points.

Let's include the header file first.

```
#include "igstkMeshObject.h"
```

Then we declare the object using smart pointers.

```
  typedef igstk::MeshObject MeshObjectType;
  MeshObjectType::Pointer mesh = MeshObjectType ::New();
```

A mesh is defined as a collection of 3-dimensional points (x,y,z) in space referenced by an identification number. In order to add points to the mesh strucutre we use the `AddPoint(unsigned int id,float x, float y,float z)` function. Let's add 4 points in our mesh.

```
mesh->AddPoint(0,0,0,0);
mesh->AddPoint(1,0,10,0);
mesh->AddPoint(2,0,10,10);
mesh->AddPoint(3,10,0,10);
```

Then we can retreive the list of points using the `GetPoints()` function.

```
  typedef MeshObjectType::PointsContainer        PointsContainer;
  typedef MeshObjectType::PointsContainerPointer PointsContainerPointer;
  PointsContainerPointer points = mesh->GetPoints();

  PointsContainer::const_iterator it = points->begin();
  while(it != points->end())
    {
    typedef MeshObjectType::PointType PointType;
    PointType pt = (*it).second;
    std::cout << "Point id = " << (*it).first << " : "
              << pt[0] << "," << pt[1] << "," << pt[2] << std::endl;
    it++;
    }
```

The next step is to define cells for the mesh. IGSTK currently supports two type of cells: tetrahedron and triangle cells. The functions to add a cell to the mesh are defined as follow:

```
  bool AddTetrahedronCell(unsigned int id,
                          unsigned int vertex1,unsigned int vertex2,
                          unsigned int vertex3,unsigned int vertex4);

  bool AddTriangleCell(unsigned int id,
                       unsigned int vertex1,
                       unsigned int vertex2,
                       unsigned int vertex3);
```

Let's add on tetrahedral cell and one triangle cell to the mesh.

```
 mesh->AddTetrahedronCell(0,0,1,2,3);

 mesh->AddTriangleCell(1,0,1,2);
```

We can then retrieve the cells using the `GetCells()`. This function returns a list of cells as described next.

```
  typedef MeshObjectType::CellsContainer        CellsContainer;
  typedef MeshObjectType::CellsContainerPointer CellsContainerPointer;
  CellsContainerPointer cells = mesh->GetCells();

  CellsContainer::const_iterator itCell = cells->begin();
  while(itCell != cells->end())
    {
    typedef MeshObjectType::CellType CellType;
    unsigned int cellId = (*itCell).first;
    std::cout << "Cell ID: " << cellId << std::endl;
    itCell++;
    }
```

9.3.8 TubeObject

The source code for this section can be found in the file
`Examples/SpatialObjects/TubeObject.cxx`.

This example describes how to use the `igstk::TubeObject` which implements a 3-dimensional tubular structure in space. The tube is defined by a set of points representing its centerline. Each point as a position and an associated radius value.

Let's start by including the appropriate header file.

```
#include "igstkTubeObject.h"
```

First we declare the object using smart pointers.

```
typedef igstk::TubeObject TubeObjectType;
TubeObjectType::Pointer tube = TubeObjectType ::New();
```

Points can be added sequentially in the tube using the `AddPoint()` function. Let's add two points: one at position (0,1,2) with a radius of 10*mm*, and second one at (1,2,3) with a radius of 20*mm*.

```
typedef TubeObjectType::PointType PointType;
PointType pt;
pt.SetPosition(0,1,2);
pt.SetRadius(10);
tube->AddPoint(pt);

pt.SetPosition(1,2,3);
pt.SetRadius(20);
tube->AddPoint(pt);
```

Then we can use the `GetNumberOfPoints()` function to get the number of points composing the tube.

```
std::cout << "Number of points in the tube = "
          << tube->GetNumberOfPoints() << std::endl;
```

There are two main functions to get points from the tube The first one is `GetPoint(unsigned int id)` which returns a pointer to the corresponding point. Note that if the index does not exist the function returns a null pointer.

```
const PointType* outPt = tube->GetPoint(0);
outPt->Print(std::cout);
```

Instead, the second `GetPoints()` function should be used because it is safer by returning the internal list of points.

```
typedef TubeObjectType::PointListType PointListType;
PointListType points = tube->GetPoints();
PointListType::const_iterator it = points.begin();
while(it != points.end())
  {
  (*it).Print(std::cout);
  std::cout << std::endl;
  it++;
  }
```

The `Clear()` function can be used to remove all the points from the tube.

```
tube->Clear();
std::cout << "Number of points in the tube after Clear() = "
          << tube->GetNumberOfPoints() << std::endl;
```

9.3.9 VascularNetwork & Vessel Objects

The source code for this section can be found in the file
`Examples/SpatialObjects/VascularNetworkObject.cxx`.

This example describes how to use the `igstk::VascularNetworkObject` to group `igstk::VesselObject`s together to represent a vascular tree.

We first include the header files.

```
#include "igstkVascularNetworkObject.h"
#include "igstkVesselObject.h"
```

Next we declare a `igstk::VascularNetworkObject`.

```
  typedef igstk::VascularNetworkObject VascularNetworkType;
  VascularNetworkType::Pointer vasculature = VascularNetworkType::New();
```

Then we create a `igstk::VesselObject`.

```
  typedef igstk::VesselObject VesselType;
  VesselType::Pointer vessel = VesselType::New();
```

Like the TubeObject, a VesselObject is defined as a collection of centerline points with associated radius.

```
  typedef VesselType::PointType PointType;
  PointType pt;
  pt.SetPosition(0,1,2);
  pt.SetRadius(10);
  vessel->AddPoint(pt);

  pt.SetPosition(1,2,3);
  pt.SetRadius(20);
  vessel->AddPoint(pt);
```

We then add the newly created vessel to the vasculature. Since the `VascularNetworkObject` derives from `GroupObject` we use the superclass `RequestAddObject()` function.

```
  vasculature->RequestAddObject(vessel);
```

In some cases, it is interesting to get a selected vessel from a `VascularNetworkObject`. To retreive the vessel, we need to setup an observer first.

```
 igstkObserverObjectMacro(Vessel,
  ::igstk::VascularNetworkObject::VesselObjectModifiedEvent,
  ::igstk::VesselObject)
```

Note that the declaration of the observer should be done outside of the class. This macro will create two functions depending on the name of the first argument:

- GotVessel() which returns true if the vessel exists.

- GetVessel() which returns the actual pointer to the vessel. Once the observer is declared we add it to the VascularNetworkProject using the AddObserver() function.

```
  typedef VesselObserver VesselObserver;
  VesselObserver::Pointer vesselObserver = VesselObserver::New();

  vasculature->AddObserver(
            VascularNetworkType::VesselObjectModifiedEvent(),
            vesselObserver);
```

We then request for a vessel given its position in the list using the RequestGetVessel(unsigned long position) function. We also check if the observer got the vessel.

```
  vasculature->RequestGetVessel(0);
  if(!vesselObserver->GotVessel())
    {
    std::cout << "No Vessel!" << std::endl;
    return EXIT_FAILURE;
    }
```

The vessel is retreived using the GetVessel() function from the observer.

```
  VesselType::Pointer outputVessel = vesselObserver->GetVessel();

  outputVessel->Print(std::cout);
  std::cout << "Number of points in the vessel = "
            << outputVessel->GetNumberOfPoints() << std::endl;
```

9.4 Reading SpatialObjects

IGSTK has the ability to read objects from files. In this section we show an example on how to read a vascular network from a file. One can notice that reading other objects is very similar.

The source code for this section can be found in the file
Examples/SpatialObjects/ReadVascularNetworkObject.cxx.

This example describes how to use the igstk::VascularNetwork to read a SpatialObject vascular tree from a file. Let's start by including the appropriate header files.

```
#include "igstkVascularNetworkReader.h"
#include "itkStdStreamLogOutput.h"
```

The SpatialObject readers return the output object via events, therefore, we declare an observer using the igstkObserverObjectMacro. This macro expect three arguments: a) the name of the observer -to be determined by the user-, b) the type of event to observe, c) the type of the object to be returned.

```
igstkObserverObjectMacro(VascularNetwork,
    ::igstk::VascularNetworkReader::VascularNetworkModifiedEvent,
    ::igstk::VascularNetworkObject)
```

First we declare the VascularNetwork using standard type definition and smart pointers.

```
  typedef igstk::VascularNetworkReader    ReaderType;
  ReaderType::Pointer  reader = ReaderType::New();
```

We then plug a logger to the reader to get information about the reading process (see the logging chapter for more information).

```
  typedef itk::Logger              LoggerType;
  typedef itk::StdStreamLogOutput  LogOutputType;

  LoggerType::Pointer   logger = LoggerType::New();
  LogOutputType::Pointer logOutput = LogOutputType::New();
  logOutput->SetStream( std::cout );
  logger->AddLogOutput( logOutput );
  logger->SetPriorityLevel( itk::Logger::DEBUG );

  reader->SetLogger( logger );
```

Then we set the filename using the `RequestSetFileName()` function.

```
  std::string filename = argv[1];
  reader->RequestSetFileName( filename );
```

Finally we ask the reader to read the object.

```
  reader->RequestReadObject();
```

In order to get the output object we first plug the observer into the reader.

```
  VascularNetworkObserver::Pointer vascularNetworkObserver
                                        = VascularNetworkObserver::New();
  reader->AddObserver(ReaderType::VascularNetworkModifiedEvent(),
                      vascularNetworkObserver);
```

Then we request the output vascular network.

```
  reader->RequestGetVascularNetwork();
```

If everything went well, the observer should receive the vascular network. We can check that this is the case using the `GotVascularNetwork()` function of the observer.

```
  if(!vascularNetworkObserver->GotVascularNetwork())
    {
    std::cout << "No VascularNetwork!" << std::endl;
    return EXIT_FAILURE;
    }
```

Finally we get the output using the `GetVascularNetwork()` function.

```
  typedef ReaderType::VascularNetworkType       VascularNetworkType;
  typedef VascularNetworkType::VesselObjectType VesselObjectType;

  VascularNetworkType::Pointer network =
                              vascularNetworkObserver->GetVascularNetwork();
```

and we display the information to make sure everything is right.

```
  network->Print(std::cout);
```

9.5 Conclusion

We have shown that IGSTK SpatialObject defines a base class for object geometry. By deriving from the base SpatialObject class, users can extend the current set of objects present in the toolkit. More complex objects can then be defined like the `igstk::UltrasoundProbeObject`.

In the next chapter we present the SpatialObjectRepresentation class which enables rendering of the SpatialObjects.

CHAPTER

TEN

SpatialObject Representation

10.1 Introduction

The SpatialObjectRepresentation classes characterize the rendering aspect of each SpatialObject. While SpatialObject defines the geometry of a given object, the SpatialObjectRepresentation describes how the object should be displayed on screen.

A SpatialObjectRepresentation can be shared between views if the rendering parameters are common between the views. However, in most of the cases, a user would want to create one SpatialObjectRepresentation per SpatialObject. This allows to tune the rendering parameters -the color of the object for instance-, while keeping the same common geometry.

This chapter is divided as follow. First, we present an example on how to display a cube in a window. Second, we show different object representations available in the toolkit.

10.2 Displaying my first object

The source code for this section can be found in the file
`Examples/SpatialObjects/ObjectRepresentation.cxx`.

This example describes how to use the `igstk::BoxObjectRepresentation` to display a `igstk::BoxObject` in a `igstk::View3D`.

This example also uses FLTK to create a window, therfore we include the appropriate header files.

```
#include "igstkBoxObjectRepresentation.h"
#include <FL/Fl_Window.h>
#include <igstkView3D.h>
```

Like any applications in IGSTK we first initialize the RealTimeClock.

```
 igstk::RealTimeClock::Initialize();
```

Then we create a cube of size 10*mm* using the `igstk::BoxObject` class.

```
typedef igstk::BoxObject                ObjectType;
typedef igstk::BoxObjectRepresentation  ObjectRepresentationType;
ObjectType::Pointer cube = ObjectType::New();
```

The appropriate object representation for the BoxObject is created using standard typedef and smart pointers.

```
ObjectRepresentationType::Pointer
                       cubeRepresentation = ObjectRepresentationType::New();
```

Every ObjectRepresentation have an RGB color and an opacity as rendering parameters. These two parameters can be tuned using the `SetColor(R,G,B)` and `SetOpacity()` functions respectively.

```
cubeRepresentation->SetColor( 0.0, 0.0, 1.0 );
cubeRepresentation->SetOpacity( 1.0 );
```

Then we tell the ObjectRepresentation to get the geometry from the BoxSpatialObject. Internally the ObjectRepresentation creates VTK actors from the actual geometry of the SpatialObject.

```
cubeRepresentation->RequestSetBoxObject(cube);
```

We then define the GUI window and the view.

```
Fl_Window * form = new Fl_Window(512,512,"Displaying my first object");

typedef igstk::View3D  View3DType;
View3DType * view3D = new View3DType(6,6,500,500,"View 3D");

form->end();
form->show();
```

We set the current representation of the object to the view using the `RequestAddObject()` function.

```
view3D->RequestAddObject( cubeRepresentation  );
```

We set the refresh rate of the view and we enable interactions.

```
view3D->RequestSetRefreshRate( 0.1 );
view3D->RequestEnableInteractions();
```

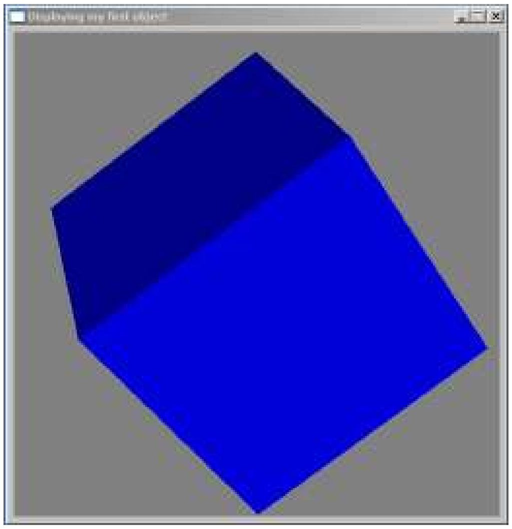


Figure 10.1: Object Representation Example.

Finally we request the view to start rendering the scene.

```
view3D->RequestStart();
```

We then refresh the display until the window is closed.

```
while(form->visible())
  {
  Fl::wait(0.05);
  igstk::PulseGenerator::CheckTimeouts();
  view3D->RequestResetCamera();
  }
```

At the end, we delete the view3D and form since they are not using smart pointers.

```
delete view3D;
delete form;
```

The output of this example is shown in figure 10.1

10.3 Standard Object Representations

In this section we present the different ObjectRepresentations available in the toolkit and we describe how they work internally.

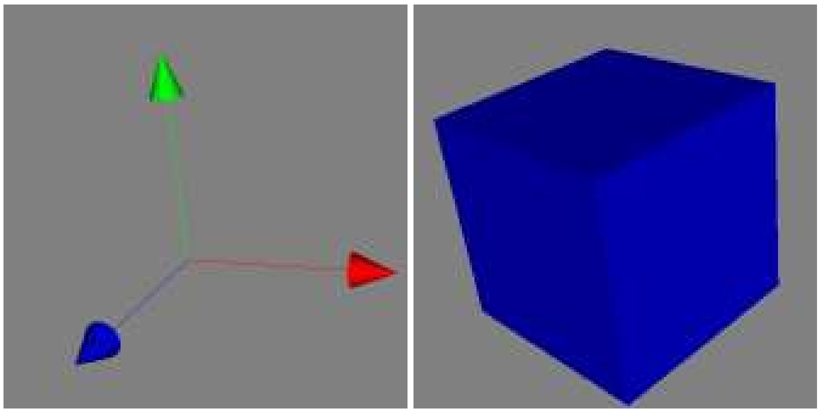


Figure 10.2: Axes and Box Object Representation.

10.3.1 Axes Object

The `igstk::AxesObjectRepresentation` uses a vtkAxesActor internally to display three orthogonal arrows representing the X, Y and Z direction. By default the X direction is represented in red, the Y direction in green and the Z direction in blue. Note that for the moment the color and the opacity of each axis cannot be changed. Also the label of the axis is turned off by default.

10.3.2 Box Object

The `igstk::BoxObjectRepresentation` uses a vtkCubeSource object internally to display a 3-dimensional hexahedron. The color and opacity of the box can be set.

10.3.3 Cone Object

The `igstk::ConeObjectRepresentation` uses a vtkConeSource object internally to display a 3-dimensional cone in space. The color and opacity of the cone can be set.

10.3.4 Cylinder Object

The `igstk::CylinderObjectRepresentation` uses a vtkCylinderSource object internally to display a 3-dimensional cylinder.

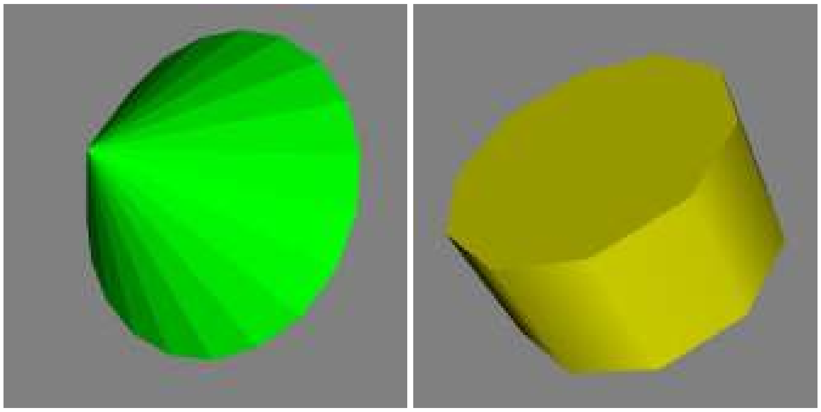


Figure 10.3: Cone and Cylinder Object Representation.

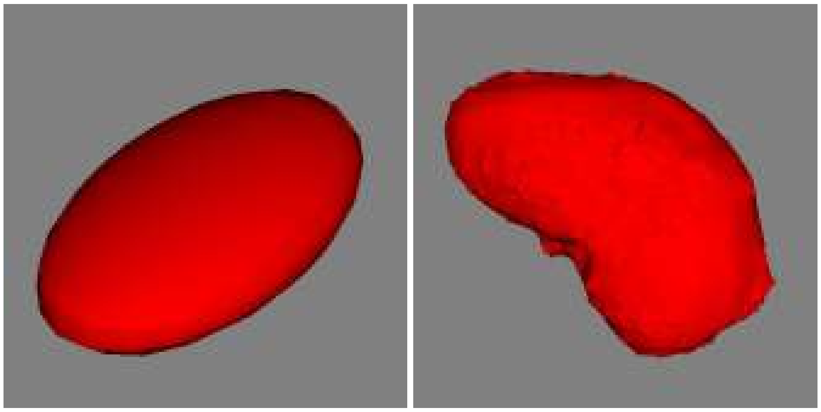


Figure 10.4: Ellipsoid and Mesh Object Representation.

10.3.5 Ellipsoid Object

The igstk::EllipsoidObjectRepresentation uses a vtkSuperquadricSource object internally to display a 3-dimensional ellipsoid.

10.3.6 Mesh Object

The igstk::MeshObjectRepresentation uses a vtkUnstructuredGrid object internally to display a 3-dimensional polydata in space. The appropriate cells are created using VTK based on their geometry.

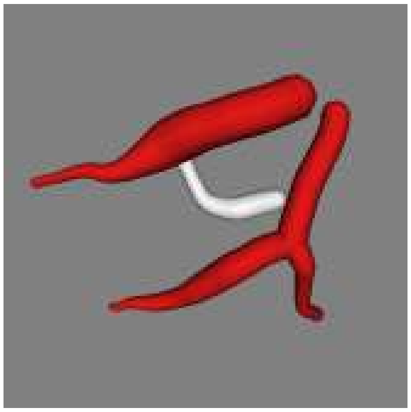


Figure 10.5: Vascular Network Object Representation.

10.3.7 Vascular Network Object

The `igstk::VascularNetworkObjectRepresentation` uses a complete VTK pipeline in order to display a 3-dimensional tube in space. First we use a vtkPolyLine to describe the centerline of the tube then we convert it to a vtkCellArray and plug it in a vtkPolyData. Here we perform a cleaning stage using the vtkCleanPolyData to make sure the tube is free from duplicate points. Finally we create the tube object using a vtkTubeFilter.

In order to look better, we also add spheres using vtkSphereSource at the extermities of each tube.

10.4 Ultrasound Probe Representation

We have seen that basic shapes can be easily represented by mapping geometrical objects into rendered objects using VTK. In this section we show that more complex shapes can be represented in IGSTK. For instance, the UltrasoundProbeObjectRepresentation class.

One can notice that the UltrasoundProbeObject class is fairly simple and only exposes the parameters the user is able to change. However its representation class involve a complex VTK pipeline. Among the VTK classes within the pipeline, the vtkCylinder, vtkPlane, vtkImplicitBoolean and the vtkMarchingContourFilter are the primary classes used.

10.5 Sharing & Duplicating Object Representations

In this section we show how to share object representations between views.

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

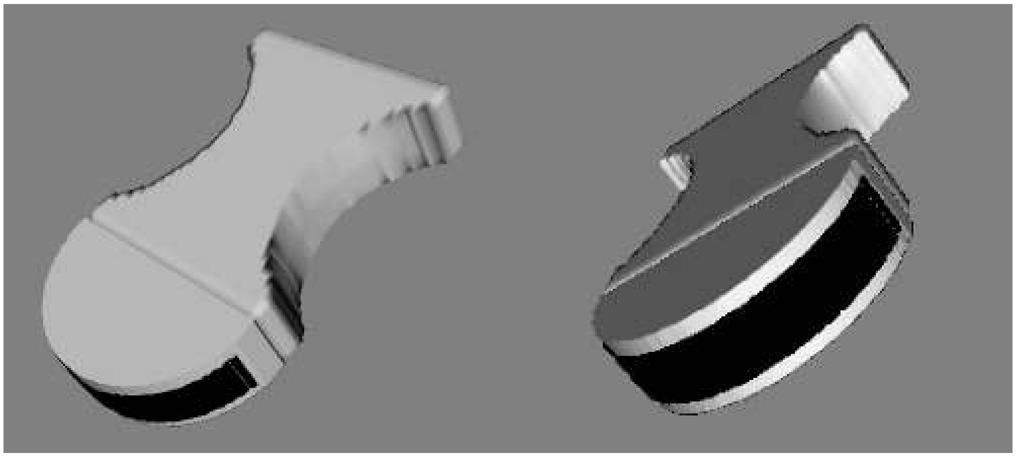


Figure 10.6: Ultrasound Probe Object Representation.

`Examples/SpatialObjects/SharedObjectRepresentation.cxx`.

This example describes how to share object representations between views. We extend the previous example and focus only on the object representation sharing. Please refer to the previous example.

We now add a second View to our window. We have `View3D1` and `View3D2` as two `igstk::View3D`s.

```
Fl_Window * form = new Fl_Window(512,262,"Sharing Object Representations");

typedef igstk::View3D  View3DType;
View3DType * view3D1 = new View3DType(6,6,250,260,"View 3D 1");
View3DType * view3D2 = new View3DType(260,6,250,260,"View 3D 2");

form->end();
form->show();
```

We set the current representation of the object to the first view using the `RequestAddObject()` function.

```
view3D1->RequestAddObject( cubeRepresentation  );
```

For this example, we create a second object representation and we set the color to be red and the opacity to 0.5. We set the same BoxSpatialObject to the geometry.

```
ObjectRepresentationType::Pointer
                       cubeRepresentation2 = ObjectRepresentationType::New();
```

```
  cubeRepresentation2->SetColor( 1.0, 0.0, 0.0 );
  cubeRepresentation2->SetOpacity( 0.5 );

  cubeRepresentation2->RequestSetBoxObject(cube);
```

We then add the newly created representation to the second view.

```
  view3D2->RequestAddObject( cubeRepresentation2  );
```

We then remove the current object representation from the second view using the `RequestRemoveObject` function.

```
  view3D2->RequestRemoveObject( cubeRepresentation2 );
```

An important function of the ObjectRepresentation is the `Copy()` function which creates a deep copy of the current representation as shown below.

```
  view3D2->RequestAddObject( cubeRepresentation->Copy() );
```

CHAPTER

ELEVEN

View

View component present renderings of surgical scenes to the clinician. The view classes are built using VTK classes encapsulated into a restrictive API subjected to control of a state machine. Viewers aggregate spatial object representations which are graphical descriptions of spatial objects. Synchronization between scene generation and rendering frequency is achieved using `igstk::PulseGenerator`. When FLTK is used for GUI purpose, the View classs take FL events and translate them into VTK events to enable user interaction within the render window.

IGSTK provide 2D and 3D viewing capability. For this purpose, `igstk::View2D` and `igstk::View3D` are provided.

11.1 State Machine

Figure 11.1 illustrates the State Machine of the `igstk::View2D` class.

This class has the following states

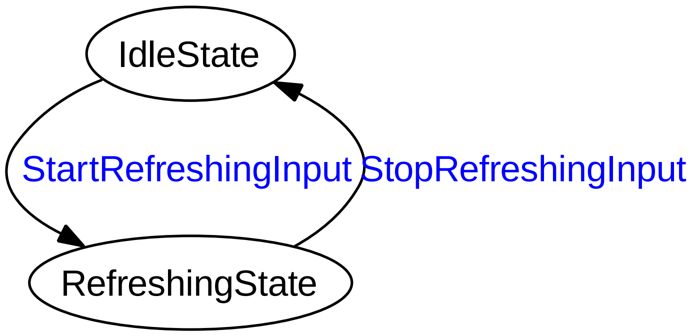


Figure 11.1: State Diagram of the View2D class.

1. *Idle* : Idle state
2. *Refreshing* : Refreshing state

11.2 Component Interface

The following methods are available in the public interface.

1. *RequestSetRefreshRate (double)* : Set the desired frequency for refreshing the view.
2. *RequestAddObject (ObjectRepresentation *)* : Add an object representation to the list.
3. *RequestAddAnnotation2D (Annotation2D *)* : Add corner annotation
4. *RequestRemoveObject (ObjectRepresentatio *)* : Remove an object representation from the list
5. *RequestSaveScreenShot (std::string)* : Save a screen shot into a file in PNG format
6. *RequestDisableInteractions()* : Disable user interactions with the render window
7. *RequestEnableInteractions()* : Enable user interactions withe render window
8. *RequestResetCamera()* : Reset the camera to a known postion and orientation
9. *RequestStart()* : Start the periodic refreshing of the view
10. *RequestStop()* : Stop the peroidic refreshing of the view

11.3 Example

The source code for this section can be found in the file
`Examples/View/View1.cxx`.

This example illustrates how to use the `igstk::View3D` class to display spatial objects.

First, a 3D view and other useful data types are defined

```
typedef igstk::View3D            View3DType;
typedef itk::Logger              LoggerType;
typedef itk::StdStreamLogOutput  LogOutputType;
```

For debugging purpose, vtk window output can be redirected to a logger, using `igstk::VTKLoggerOutput` as follows.

```
igstk::VTKLoggerOutput::Pointer vtkLoggerOutput =
                                igstk::VTKLoggerOutput::New();
vtkLoggerOutput->OverrideVTKWindow();
vtkLoggerOutput->SetLogger(logger);
```

In this example, we would like to display an ellipsoid object. To carry out this, an ellipsoid spatial object is first instantiated.

```
igstk::EllipsoidObject::Pointer ellipsoid = igstk::EllipsoidObject::New();
ellipsoid->SetRadius(0.1,0.1,0.1);
```

Next, a representation object is created using `igstk::EllipsoidObjectRepresentation` class. The representation class provides the mechanism to generate graphical description of the spatial object for visualization in a VTK scene.

```
igstk::EllipsoidObjectRepresentation::Pointer ellipsoidRepresentation =
                              igstk::EllipsoidObjectRepresentation::New();
ellipsoidRepresentation->RequestSetEllipsoidObject( ellipsoid );
ellipsoidRepresentation->SetColor(0.0,1.0,0.0);
ellipsoidRepresentation->SetOpacity(1.0);
```

Geometrical transformation can be applied to the ellipsoid spatial object as follows.

```
const double validityTimeInMilliseconds = 1e300; // 100 seconds
igstk::Transform transform;
igstk::Transform::VectorType translation;
translation[0] = 0;
translation[1] = 10;
translation[2] = 10;
igstk::Transform::VersorType rotation;
rotation.Set( 0.0, 0.0, 0.0, 1.0 );
igstk::Transform::ErrorType errorValue = 10; // 10 millimeters

transform.SetTranslationAndRotation(
    translation, rotation, errorValue, validityTimeInMilliseconds );
ellipsoid->RequestSetTransform( transform );
```

Next, FLTK window and a view object is instantiated.

```
Fl_Window * form = new Fl_Window(601,301,"View Test");
View3DType * view3D = new View3DType(310,10,280,280,"3D View");
form->end();
form->show();
```

The ellipsoid is added to the scene using `RequestAddObject` method.

```
view3D->RequestAddObject( ellipsoidRepresentation );
```

The View components are designed for refreshing their representation at regular intervals. The application developer must set the desired refresh rate in Hertz and should trigger the start of the process of internal generation of pulses that makes possible for the View class to refresh itself.

```
view3D->RequestSetRefreshRate( 30 );
view3D->RequestStart();
```

At this point it is now possible to start the event loop that will drive the user interaction of the application. Inside the loop it is of fundamental importance to invoke the call to the `igstk::PulseGenerator` method `CheckTimeouts()`. This methods ensures that the timers of pulse generators in all the autonomous IGSTK classes are checked to see if they should trigger timer events. The same for loop should have some form of wait or sleep instruction in order to prevent the loop from taking over the CPU time.

```
for(unsigned int i=0; i<10; i++)
  {
  Fl::wait( 0.01 );
  igstk::PulseGenerator::CheckTimeouts();
  Fl::check();       // trigger FLTK redraws
  }
```

Once the event loop finishes, we should stop the refresh process of the view class, by calling the method `RequestStop()`.

```
view3D->RequestStop();
```

CHAPTER

TWELVE

Logging

For most of critical software systems, logging component enables the post-analysis of the operational process and recovery from failure. In [1] ITK and IGSTK, logging component is provided to help record messages to output streams so that analysis, verification and debugging of systems and operational processes are efficiently done.

This chapter will cover the logging component in ITK 2.8.1 and the current version of IGSTK at the time of this writing.

12.1 Role of Logging in DBMS

Database management systems use logs to recover from various failures, to backup data, and to analyze the reason of failure. The integrity of databases is guaranteed using logs by logging before writing on disks (which is called Write-Ahead Logging Protocol, WAL), databases can be restored using logs even when the DBMS or computer system crashed down. Restart recovery is initiated while restarting the DBMS after system failure. The first pass of the restart recovery is to analyze logs from the last check point and initialize in-memory data structures such as a transantion table. In the second pass, DBMS redoes all previous transaction operations according to logs because the state of in-memory data structures and buffers were lost and some of data in buffers were not reflected on disks after the sudden failure. In this pass, the data that were modified only in buffers and logs from the last transaction are restored. The last pass is to undo all the loser transactions by rolling back transactions stopped before the completion. [12]

12.2 Role of Logging in IGSTK

In image-guided surgery systems, logs will be used for enhancing the robustness of the system during development, analysing the causes and points of failures, and verifying that the system

[1] As ITK needed the logging component and IGSTK is based on ITK, base logging classes are included in ITK. IGSTK extends the base classes to support FLTK, VTK, the real-time clock with the unified time unit and so on.

works in an expected way. Action-Ahead Logging or Operation-Ahead Logging protocol is to ensure that logging on a disk, not in a buffer should be done before any action or operation. These terms are derived from WAL because image-guided surgery systems do not only include writing but also human involvement and hardware operations. By sticking to OAL, logs cannot miss the last operation to be done before the system failure. A variety of hardware can be involved in the image-guided surgery procedure and it makes the automatic recovery from the sudden failure unfeasible because most of hardware requires human involvement especially at the setup and initialization stages in thesedays. To guarantee the integrity of data written during the procedure, DBMS is recommended to be used instead of writing logs and transaction processing codes in the image-guided surgery software itself. Therefore, the purposes of logging in IGSTK generally are confined to the analysis, verification, and debugging of systems and procedures. Recovery using logs might be done in certain situations but is not expected to be common in the image-guided surgery field.

12.3 Structure of the Logging Component

The level of seriousness is assigned to each log message. A logger prints only messages with the level above the logger's designated level. With this feature, developers can filter out uninteresting messages. For example, messages for debugging can be printed only on debugging phase of development. Each logger can print messages to multiple destinations (files, consoles, GUI windows). Loggers can be run in a separate thread, printing each messages one by one without mixing up multiple messages. Multiple loggers can be created and used through LoggerManager. Messages toward ITK and VTK message windows can be redirected to Loggers so that every message generated by the software using Logging components of ITK and IGSTK is concentrated in the unified logging service.

12.3.1 LogOutput

itk::LogOutput class represents the destination of the logging and serves as a base class for other LogOutput classes. Derived classes from the LogOutput class encapsulate the certain location on certain media. Currently, StdStreamLogOutput and MultipleLogOutput classes are available.

StdStreamLogOutput

itk::StdStreamLogOutput encapsulates the standard output stream, which can basically be a console output stream, an error output stream, or a file output stream.

MultipleOutput

igstk::MultipleOutput contains multiple standard output streams and can be used as standard output streams. It forwards strings to streams that it contains. This class is not a LogOutput class and cannot be used with the logging component.

MultipleLogOutput

itk::MultipleLogOutput aggregates multiple LogOutput objects in it. However, it's already used in LoggerBase class so that Logger classes can contain multiple LogOutput objects.

FLTKTextBufferLogOutput

```
igstk::FLTKTextBufferLogOutput forwards messages to a FLTK text buffer
which is in Fl_Text_Buffer type. This buffer can be used for other
FLTK widgets.
```

FLTKTextLogOutput

```
igstk::FLTKTextLogOutput displays messages in FLTK text window which
is in Fl_Text_Display type. This is meant to diplay log messages on
a GUI window.
```

Extending LogOutput

Custom LogOutputs derived from LogOutput class can encapsulate other output streams such as TCP/IP connection, OS-dependent system log records, the ring of files, database table, and so on. Developers only need to provide methods for writing, flushing a buffer, and setting up the output stream(s).

12.3.2 Logger

LoggerBase

LoggerBase class is the base implementation of other Logger classes. A Logger object can contain multiple LogOutputs and messages for that Logger object are written to every LogOutput of the Logger at the same time.

PriorityLevel

Each Logger object has a designated priority level, which can be one of following items:

MUSTFLUSH In this level, messages with any other level are filtered out.

FATAL Logger only posts messages with FATAL or MUSTFLUSH level to LogOutputs.

CRITICAL only posts CRITICAL, FATAL, MUSTFLUSH level messages.

WARNING only posts WARNING, ..., MUSTFLUSH level messages.

INFO only posts INFO, ..., MUSTFLUSH level messages.

DEBUG only posts DEBUG, ..., MUSTFLUSH level messages.

NOTSET Logger posts every message to LogOutputs.

For each message, its priority level means:

MUSTFLUSH Messages with this level must be flushed in any case.

FATAL Messages with this level contain information about fatal error or exception. Application must abort.

CRITICAL Messages with this level contain information about critical error or exception. Operation was terminated.

WARNING Messages with this level give a warning about potential danger.

INFO Messages with this level contain information without potential or expected danger.

DEBUG Messages with this level contain information or data for debugging.

NOTSET Messages with this level have no serious information.

The Logger object only posts messages if the priority level of a message is equal to or more serious than the priority level of the Logger.

Flushing

Posting data to output media generally does buffering to reduce the number of output operations. Buffering is especially effective when the minimal cost of each output operation is expensive. Hard disks only allow block I/O and each message through telecommunication requires travel time from the source location to the target location, and many other output devices need buffering. Buffered data/messages should be flushed out of the buffer to the real destination at some point. At least, flushing must be done when the buffer is full and this situation is automatically handled by the operating system or system software. Urgent messages should be flushed out even sacrificing the efficiency of the output operation and this is a common case when logging for serious reasons. Logger objects provide two options to handle the latter case.

- To use Flush() method manually
- To set a minimal LevelForFlushing and let Logger automatically flush whenever messages with an equal or more serious level than the specified LevelForFlushing are met.

Formatting

Loggers derived from LoggerBase can override BuildFormattedEntry() method to do custom formatting. The overridden method can format the message by creating and returning a string in any format. The default format is as following:

[timestamp in seconds] : [logger name] [priority] [message]

Timestamp

Timestamp for logging is provided by RealtimeClock object. Each message contains its timestamp in seconds.

Logger

The Logger class is derived from LoggerBase but Logger doesn't add any other functionality from LoggerBase. Logger has the simplest form as a derived class of LoggerBase.

LoggerManager

For managing multiple loggers, multiple logger objects can be contained in a LoggerManager object. A name is given to each logger so that loggers can be accessed by that name. Using LoggerManager's interface, logging operations to every logger in a LoggerManager is done via one method call instead of calling one method call per one logger.

12.3.3 Multi-threaded Logging

LoggerThreadWrapper

itk::LoggerThreadWrapper is a template class that wraps a logger class and enables logging in a separate thread. LoggerThreadWrapper inherits the Logger class given as a template argument. Logging method calls through the LoggerThreadWrapper object make logging operation queued and performed whenever the thread takes the available computational resource and the queue is not empty. Although LoggerThreadWrapper provides more flexibility than ThreadLogger by allowing various types of Loggers to be used but LoggerThreadWrapper has a problem when used with compilers weak at the C++ template feature.

ThreadLogger

itk::ThreadLogger provides the same functionality with LoggerThreadWrapper except that ThreadLogger is derived from the Logger class. If different types of Logger is necessary instead of the simple Logger class, Loggers classes should be further derived from ThreadLogger class by creating a new class or using LoggerThreadWrapper.

12.3.4 Redirecting ITK, VTK log messages to Logger

Overriding itk::OutputWindow

itk::LoggerOutput class can override itk::OutputWindow and redirect log messages from ITK to a Logger object. ITK applications can still use conventional ITK log codes and those messages can be sent to a Logger in any type by using LoggerOutput class.

Overriding vtkOutputWindow

igstk::VTKLoggerOutput redirects log messages from VTK to a Logger object. This class plays a similar role as itk::LoggerOutput but is included in IGSTK.

12.4 Example

The source code for this section can be found in the file
`Examples/Logging/Logging1.cxx`.

This example shows how to extend Logger for printing log messages in a custom format.

XMLLogger class is defined to construct log messages in XML format. Indentation is automatically done here. Most of XML viewers show the hierarchical structure of log messages and provide UIs to collapse and expand subelements. XMLLogger opens a new element when the first character of the log message is

```
'<'
```

and closes a element when the first character is

```
'>'
```

Otherwise, a self-closing element is created when no angular bracket is used for the first character. BuildFormattedEntry() method is redefined in XMLLogger class for overriding default formatting. It creates a string containing a timestamp, a logger name, the priority level of a message, and a message content. Some of these components can be omitted if unnecessary. The logger name was omitted here to shorten the length of messages.

```
namespace igstk
{
```

```
class XMLLogger: public itk::Logger
{
public:

  typedef XMLLogger                                  Self;
  typedef itk::SmartPointer<Self>                    Pointer;
  typedef itk::Logger                                Superclass;

  igstkNewMacro(Self);

  /** Provides a XML-formatted log entry */
  virtual std::string BuildFormattedEntry(PriorityLevelType level,
    std::string const & content)
    {
    static std::string m_LevelString[] = { "MUSTFLUSH", "FATAL",
      "ERROR", "WARNING", "INFO", "DEBUG", "NOTSET" };
    itk::OStringStream s;
    s.precision(30);
    if( content.at(0) == '<' )
      {
      for( int i = 0; i < m_Depth; ++i )
        {
        s << "  ";
        }

      s << "<Log timestamp='" << m_Clock->GetTimeStamp()
        <<  "' level='" <<  m_LevelString[level]
        << "' message='" << content.substr(1, content.size()-1) << "'>"
        << std::endl;
      ++m_Depth;
      }
    else if( content.at(0) == '>' )
      {
      --m_Depth;
      for( int i = 0; i < m_Depth; ++i )
        {
        s << "  ";
        }

      s << "</Log>" << std::endl;
      }
    else
      {
      for( int i = 0; i < m_Depth; ++i )
        {
        s << "  ";
        }
      s << "<Log timestamp='" << m_Clock->GetTimeStamp()
```

```
        <<  "' level='" <<  m_LevelString[level]
        << "' message='" << content << "'/>"
        << std::endl;
      }
    return s.str();
    }

protected:
  /** Constructor */
  XMLLogger() {m_Depth = 0;}

  /** Destructor */
  virtual ~XMLLogger() {};

private:

  int m_Depth;

};

} // namespace igstk
```

The following code fragment creates an XMLLogger instance, StdStreamLogOutput instances connected to a log file and the console, and then sets parameters for the logger.

```
  typedef igstk::XMLLogger              LoggerType;
  typedef itk::StdStreamLogOutput       LogOutputType;
  LoggerType::Pointer   logger = LoggerType::New();
  LogOutputType::Pointer logOutput = LogOutputType::New();
  LogOutputType::Pointer logOutput2 = LogOutputType::New();
  ofstream fout("log.xml");
  logOutput->SetStream( fout );
  logOutput2->SetStream( std::cout );
  logger->AddLogOutput( logOutput );
  logger->AddLogOutput( logOutput2 );
  logger->SetPriorityLevel( itk::Logger::DEBUG );
```

The XMLLogger prints log messages in XML format so that the log messages are structured hierarchically. After running this example, open the generated log.xml file using an XML viewer.

```
  logger->Debug("<main()");

  logger->Debug("Hello world1");

  logger->Critical("<nested 1");
  logger->Debug("Hello world2");
  logger->Info("<nested 2");
  logger->Debug("Hello world3");
  logger->Error("Hello world4");
  logger->Info(">nested 2");

  logger->Debug("Hello world5");
  logger->Critical(">nested 1");

  logger->Debug(">main()");
```

log.xml file is generated as the following:

```
<Log timestamp='24444134232.34433' level='DEBUG' message='main()'>
 <Log timestamp='24444134232.3451' level='DEBUG' message='Hello world1'/>
 <Log timestamp='24444134232.345699' level='ERROR' message='nested 1'>
  <Log timestamp='24444134232.346279' level='DEBUG' message='Hello world2'/>
  <Log timestamp='24444134232.346897' level='INFO' message='nested 2'>
   <Log timestamp='24444134232.347507' level='DEBUG' message='Hello world3'/>
   <Log timestamp='24444134232.348164' level='ERROR' message='Hello world4'/>
  </Log>
  <Log timestamp='24444134232.348892' level='DEBUG' message='Hello world5'/>
 </Log>
</Log>
```

CHAPTER

THIRTEEN

ImageIO

One of the essential components of an image guided surgery application is image input/output module. Preoperative/Intraoperative images are read in for surgical planning and guidance. Only DICOM image format is handled in IGSTK. This was a design decision made to make sure image formats not suitable for medical application are not processed by IGSTK algorithms. IGSTK provides classes to read DICOM data from CT, MRI and Ultrasound modalities. These classes are derived from `igstk::ImageReader` base class which is templated over `igstk::ImageSpatialObject`

13.1 DICOM Reader

DICOM image reader class serves as a base class for MRI, US and CT image reader subclasses. These derived classes are templated over image spatial object type. For example, the MRI image reader class is templated over MRI image spatial object. DICOM image reader class uses GDCM library for DICOM image reading. INSERT a block diagram depicting the class hierachy of the dicom image reader classes.

13.1.1 State Machine Design

Figure 13.1 illustrates the State Machine of the `igstk::DICOMImageReader` class.

This class has the following states

1. *Idle*: Idle state
2. *ImageDirectoryNameRead*: The name of the directory containing the DICOM data is read
3. *ImageSeriesFileNamesGenerated*: The dicom series filenames are generated
4. *AttemptingToReadImage*: The state machine is in a transition state reading the DICOM image data

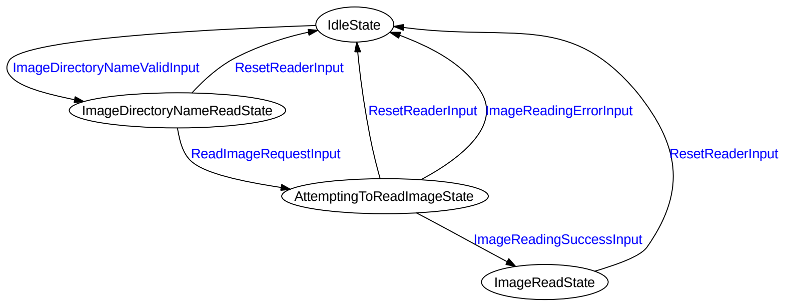


Figure 13.1: State Diagram of the DICOMImageReader class.

5. *ImageRead*:DICOM image is read

13.1.2 Component Interface

The following methods are available in the public interface.

1. *RequestSetDirectory*: Set the directory name containing the DICOM data
2. *RequestReadImage*: Request image reading
3. *RequestGetPatientNameInformation*: Request the reader to throw patient name information loaded event
4. *RequestGetModalityInformation*: Request the reader to throw a modaltiy information loaded event

13.1.3 Special features

The dicom image readers check the validity of the input dicom data to avoid incorrect reading and reconstruction of 3D volume . One of the issues is gantry tilt. The itk::OrientedImage class that is used internally by `igstk::DICOMImageReader` handles only dicom data acquired in 3D orthogonal space. In the preprocessing stage, the DICOM image reader checks on the value of the gantry tilt. If the gantry tilt is greater than a threshold value, the reader throws image invalid error event.

Similarly the modaltiy type of the input DICOM image data is checked to make sure the correct dicom image reader is used to read the input dicom data. For example, if the one attempts to read a CT image data using MRI image reader, then invalid image reading error event will be thrown.

13.1.4 Example

The source code for this section can be found in the file
`Examples/DICOMImageReader/DICOMImageReader1.cxx`.

This example illustrates how to use the DICOM image reader.

To use this class, appropriate callback subclasses need to be first defined. This procedure is important because information is passed from the reader class to the application using information loaded events. The events could be error events or events loaded with dicom information such as modality and patient ID.

For example, callback class to observe Modality information is defined as follows.

```
class DICOMImageModalityInformationCallback: public itk::Command
{
public:
  typedef DICOMImageModalityInformationCallback    Self;
  typedef itk::SmartPointer<Self>                  Pointer;
  typedef itk::Command                             Superclass;
  itkNewMacro(Self);

  typedef igstk::ImageSpatialObject<
                                short,
                                3 >                ImageSpatialObjectType;


  typedef igstk::DICOMModalityEvent DICOMModalityEventType;

  void Execute(const itk::Object *caller, const itk::EventObject & event)
    {

    }
  void Execute(itk::Object *caller, const itk::EventObject & event)
    {
    if( DICOMModalityEventType().CheckEvent( &event ) )
      {
      const DICOMModalityEventType * modalityEvent =
                   dynamic_cast< const DICOMModalityEventType *>( &event );
      std::cout << "Modality= " << modalityEvent->Get() << std::endl;
      }

    }
protected:
  DICOMImageModalityInformationCallback()   { };

private:
};
```

Simialar callback classes need to be defined to observe patient name and image reading error.

In this example, we would like to read a CT image. Therefore, first a CT image reader object is insantiated as follows.

```
typedef igstk::CTImageReader        ReaderType;
ReaderType::Pointer   reader = ReaderType::New();
```

A logger can be linked to the reader.

```
reader->SetLogger( logger );
```

First, dicom image directory is set

```
reader->RequestSetDirectory( directoryName );
```

Next, the user makes a request to read the image.

```
reader->RequestReadImage();
```

To access DICOM information about this image, callback objects need to be instantiated and added to the observer list of the reader object.

```
typedef DICOMImageModalityInformationCallback   ModalityCallbackType;

ModalityCallbackType::Pointer dimcb = ModalityCallbackType::New();
reader->AddObserver( igstk::DICOMModalityEvent(), dimcb );
reader->RequestGetModalityInformation();
```

Similar operation can be performed to access the patient name

```
/* Add observer to listen to patient name  info */
typedef DICOMImagePatientNameInformationCallback  PatientCallbackType;

PatientCallbackType::Pointer dipncb = PatientCallbackType::New();
reader->AddObserver( igstk::DICOMPatientNameEvent(), dipncb );
reader->RequestGetPatientNameInformation();
```

13.2 Screenshot generation

IGSTK provides the capability to save screen shots. For this purpose, `VTK::WindowToImageFilter` is used. This filter basically takes a screenshot of the render window and saves it as an image file. The image file format of choice is PNG.

CHAPTER

FOURTEEN

Registration

One of the critical steps in image-guided surgery is registering pre-operative images to the patient coordinate system. Various registration techniques have been developed for this purpose, and they fall into two broad categories: frame-based and frameless . In frame-based registration technique, stereotactic frames are attached to the organ of interest to provide a rigid reference. This method of registration is common in neurosurgery.

In frameless registration, fiducial marks (landmarks) in both the pre-operative image and the patient are used to compute the transform parameters that relate the pre-operative image and the patient. The fiducials can be point or surface patches. In IGSTK, a point-based 3D rigid body landmark registration class(`igstk::Landmark3DRegistration`) is implemented. Using a tracking device or pointer. the operator identifies the landmark points in the preoperative image and the patient body and transform parameters are computed based on these points. IGSTK also provides a tool to predict landmark registration error. (`igstk::igstkLandmarkRegistrationErrorPredictor`)

14.1 Landmark-based registration

IGSTK contains a landmark-based registration class which is a wrapper class around (`igstk::itk::LandmarkBasedTransformInitializer`). This registration class implements an absolute orientation solution using unit quaternions derived by Berthold K.P. Horn ([8]. He developed a closed-form solution to the least-squares problem of computing transformation parameters between two coordinate systems.

14.1.1 State Machine Design

Figure 14.1 illustrates the state diagram for `igstk::Landmark3DRegistration`.

This class has the following states.

1. *Idle* : No landmark points added

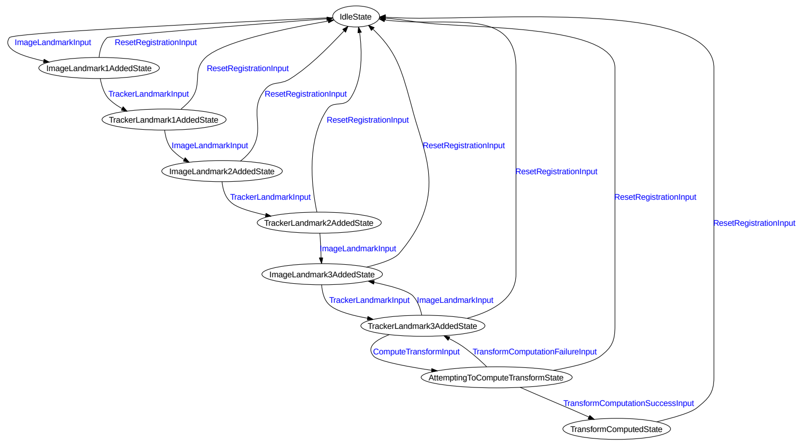


Figure 14.1: Landmark registration component state diagram.

2. *ImageLandmark1Added*: Landmark 1 image coordinate added
3. *TrackerLandmark1Added*: Landmark 1 tracker coordinate added
4. *ImageLandmark2Added*: Landmark 2 image coordinate added
5. *TrackerLandmark2Added*: Landmark 2 tracker coordinate added
6. *ImageLandmark3Added*: Landmark 3 and more image coordinate added
7. *TrackerLandmark3Added*: Landmark 3 and more tracker coordinate added
8. *AttemptingToComputeTransform* : transition state after a request is made for transform computation
7. *TransformComputed* : transform parameters computed state

More than three landmark points can be used to compute the transform parameters. In these situations, the state machine recurses between the *ImageLandmark3Added* and *TrackeLandmark3Added* states as shown in the state diagram. To avoid human error in establishing the co-ordinates, the state machine is designed so that corresponding landmark coordinates are added to the point containers successively (image coordinate followed by the tracker coordinate of the landmark).

14.1.2 Component Interface

The following methods are available in the public interface.

1. *RequestAddImageLandmarkPoint(LandmarkPointType)*: Method to add image landmark point.
2. *RequestAddTrackerLandmarkPoint(LandmarkPointType)*: Method to add tracker landmark point
3. *RequestResetRegistration( )*: Method to reset the state machine to idle state
4. *RequestComputeTransform ( )*: Method to request transform computation
5. *RequestGetTransform ( )*: Method to request transform parameters. The transform parameters are returned to the requesting application as String loaded event (`igstk::TransformModifiedEvent`) . The user needs to set up a callback to observe this transform event
6. *ComputeRMSError()* : Method to compute root mean square of landmark registration

14.1.3 Example

The source code for this section can be found in the file
`Examples/LandmarkRegistration/LandmarkRegistration1.cxx`.

This example illustrates how to use igstk's landmark registration component to determine rigid body transformation parameters between an image and the patient coordinate system.

To use the registration component, the header file for `igstk::Landmark3DRegistration` is added.

```
#include "igstkLandmark3DRegistration.h"
```

Transform parameters are returned to the application using loaded events. To handle these events, the following `igstk::Events` and `igstk::Transform` header files are needed.

```
#include "igstkEvents.h"
#include "igstkTransform.h"
```

To fully utilize the registration component, callbacks need to be set up to observer events that could be thrown by the registration component. For this purpose, the ITK command class is used to derive a callback class . The ITK command class implements a subject/observer (command design) pattern. A subject notifies an observer by running the `Execute` method of the derived callback class . For example a callback class meant to observe an error in the transform computation is defined as follows.

```
class Landmark3DRegistrationErrorCallback : public itk::Command
{
public:
  typedef Landmark3DRegistrationErrorCallback Self;
  typedef itk::SmartPointer<Self>             Pointer;
  typedef itk::Command                        Superclass;
  itkNewMacro(Self);
  void Execute(const itk::Object *caller, const itk::EventObject & event)
    {

    }
  void Execute(itk::Object *caller, const itk::EventObject & event)
    {
    std::cerr<<"Error in transform computation"<<std::endl;
    }
protected:
  Landmark3DRegistrationErrorCallback() {};

private:
};
```

Similarly, a callback class needs to be defined to observe the `igstk::TransformModified` event. This event is loaded with transform parameters that are computed by the registration component.

```
class Landmark3DRegistrationGetTransformCallback: public itk::Command
{
public:
  typedef Landmark3DRegistrationGetTransformCallback    Self;
  typedef itk::SmartPointer<Self>                       Pointer;
  typedef itk::Command                                  Superclass;
  itkNewMacro(Self);

  typedef igstk::TransformModifiedEvent TransformModifiedEventType;

  void Execute( const itk::Object *caller, const itk::EventObject & event )
    {
    }

  void Execute( itk::Object *caller, const itk::EventObject & event )
    {
    std::cout<< " TransformEvent is thrown" << std::endl;
    const TransformModifiedEventType * transformEvent =
                dynamic_cast < const TransformModifiedEventType* > ( &event );
    m_Transform = transformEvent->Get();
    m_EventReceived = true;
    }
  bool GetEventReceived()
    {
    return m_EventReceived;
    }
  igstk::Transform GetTransform()
    {
    return m_Transform;
    }
protected:

  Landmark3DRegistrationGetTransformCallback()
    {
    m_EventReceived = true;
    }

private:
  bool m_EventReceived;
  igstk::Transform m_Transform;
};
```

After the helper classes are defined, the main function implementation is started.

```
int main( int argv, char * argc[] )
{
```

All the necessary data types are defined.

```
  typedef itk::Logger                  LoggerType;
  typedef itk::StdStreamLogOutput      LogOutputType;

  typedef igstk::Landmark3DRegistration
                            Landmark3DRegistrationType;
  typedef igstk::Landmark3DRegistration::LandmarkPointContainerType
                            LandmarkPointContainerType;
  typedef igstk::Landmark3DRegistration::LandmarkImagePointType
                            LandmarkImagePointType;
  typedef igstk::Landmark3DRegistration::LandmarkTrackerPointType
                            LandmarkTrackerPointType;
  typedef Landmark3DRegistrationType::TransformType::OutputVectorType
                            OutputVectorType;
  typedef igstk::Transform  TransformType;
```

The registration component is instantiated as follows

```
  Landmark3DRegistrationType::Pointer landmarkRegister =
                                        Landmark3DRegistrationType::New();
```

The landmark containers that hold the landmark image and tracker coordinates are instantiated.

```
  LandmarkPointContainerType  imagePointContainer;
  LandmarkPointContainerType  trackerPointContainer;
```

Error event callback objects are instantiated and added to the observer list of the registration component as follows:

```
Landmark3DRegistrationInvalidRequestCallback::Pointer
                lrcb = Landmark3DRegistrationInvalidRequestCallback::New();

typedef igstk::Landmark3DRegistration::InvalidRequestErrorEvent
                                                       InvalidRequestEvent;
landmarkRegister->AddObserver( InvalidRequestEvent(), lrcb );

Landmark3DRegistrationErrorCallback::Pointer ecb =
                Landmark3DRegistrationErrorCallback::New();
typedef igstk::Landmark3DRegistration::TransformComputationFailureEvent
                                                   ComputationFailureEvent;
landmarkRegister->AddObserver( ComputationFailureEvent(), ecb );
```

A logger can be connected to the registration component for debugging purpose as follows

```
LoggerType::Pointer   logger = LoggerType::New();
LogOutputType::Pointer logOutput = LogOutputType::New();
logOutput->SetStream( std::cout );
logger->AddLogOutput( logOutput );
logger->SetPriorityLevel( itk::Logger::DEBUG );
landmarkRegister->SetLogger( logger );
```

Next, landmark points are added to the image and tracker containers. The state machine of this registration component is designed so that the image and tracker coordinates that correspond to each landmark are added consecutively. This scheme prevents the mismatch in landmark correspondence that could occur when all landmarks image coordinates are recorded first and then the tracker coordinates. This design choice is consistent with the ”safety by design” philosophy of igstk.

```
// Add 1st landmark
imagePoint[0] =  25.0;
imagePoint[1] =  1.0;
imagePoint[2] =  15.0;
imagePointContainer.push_back(imagePoint);
landmarkRegister->RequestAddImageLandmarkPoint(imagePoint);

trackerPoint[0] =  29.8;
trackerPoint[1] =  -5.3;
trackerPoint[2] =  25.0;
trackerPointContainer.push_back(trackerPoint);
landmarkRegister->RequestAddTrackerLandmarkPoint(trackerPoint);

// Add 2nd landmark
```

```
imagePoint[0] =  15.0;
imagePoint[1] =  21.0;
imagePoint[2] =  17.0;
imagePointContainer.push_back(imagePoint);
landmarkRegister->RequestAddImageLandmarkPoint(imagePoint);

trackerPoint[0] =  35.0;
trackerPoint[1] =  16.5;
trackerPoint[2] =  27.0;
trackerPointContainer.push_back(trackerPoint);
landmarkRegister->RequestAddTrackerLandmarkPoint(trackerPoint);

// Add 3d landmark
imagePoint[0] =  14.0;
imagePoint[1] =  25.0;
imagePoint[2] =  11.0;
imagePointContainer.push_back(imagePoint);
landmarkRegister->RequestAddImageLandmarkPoint(imagePoint);

trackerPoint[0] =  36.8;
trackerPoint[1] =  20.0;
trackerPoint[2] =  21.0;
trackerPointContainer.push_back(trackerPoint);
landmarkRegister->RequestAddTrackerLandmarkPoint(trackerPoint);
```

More landmarks can be added for the transform computation.

After all the landmark coordinates are added, the transform computation is requested as follows

```
landmarkRegister->RequestComputeTransform();
```

To access the tranform parameters, a GetTransform callback is instantiated to observe the transform event as follows.

```
Landmark3DRegistrationGetTransformCallback::Pointer lrtcb =
                          Landmark3DRegistrationGetTransformCallback::New();
landmarkRegister->AddObserver( igstk::TransformModifiedEvent(), lrtcb );
```

To request the registration component throw an event loaded with transform parameters, a `RequestGetTransform` function is invoked as follows.

```
landmarkRegister->RequestGetTransform();
std::cout << "Transform " << transform << std::cout;
```

14.2 Registration error prediction

Determining the accuracy of registration is critical for image guided surgery systems. The conventional way of evaluating such accuracy is to compute the root means square error between the corresponding fiducials after registration. However, it was shown that the fiducial registration error is independent of fiducial configuration and it is a poor predictor of registration accuracy [13]. Consequently, West et al [13] have derived a more robust error predictor that takes fiducial configuration into account as shown in Equation 14.1. IGSTK has implementation of this error prediction algorithm (`igstk::Landmark3DRegistrationErrorEstimator`)

$$TRE^2(t) = \frac{FRE^2}{N-2}(1+\frac{1}{3}\sum_{k=1}^{3}\frac{d_k^2}{f_k^2}) \tag{14.1}$$

Where TRE is the target point registration error, FRE is the landmark registration error, N is the number of landmarks, d_k is the distance of the target point from principal axis k, and f_k is the RMS distance of the landmarks.

14.2.1 State Machine Design

Figure 14.2 illustrates the state diagram of the `igstk::Landmark3DRegistrationErrorEstimator`

This class has the following states

1. *Idle* : idle state
2. *LandmarkContainerSet*: landmark point container added
3. *LandmarkRegistrationErrorSet* : Landmark registration error value set
4. *AttemptingToComputeErrorParameters* : Transition state to compute error parameters required to estimate the target point registration error
5. *ErrorParametersComputed* : Error parameters computed
6. *TargetPointSet* : Target point set
7. *AttemptingToEstimateTargetRegstirationError*: Transition state to estimate target point registration error
8. *TargetRegistrationErrorEstimated* : Target point registration error estimated

14.2.2 Component Interface

The following methods are available in the public interface.

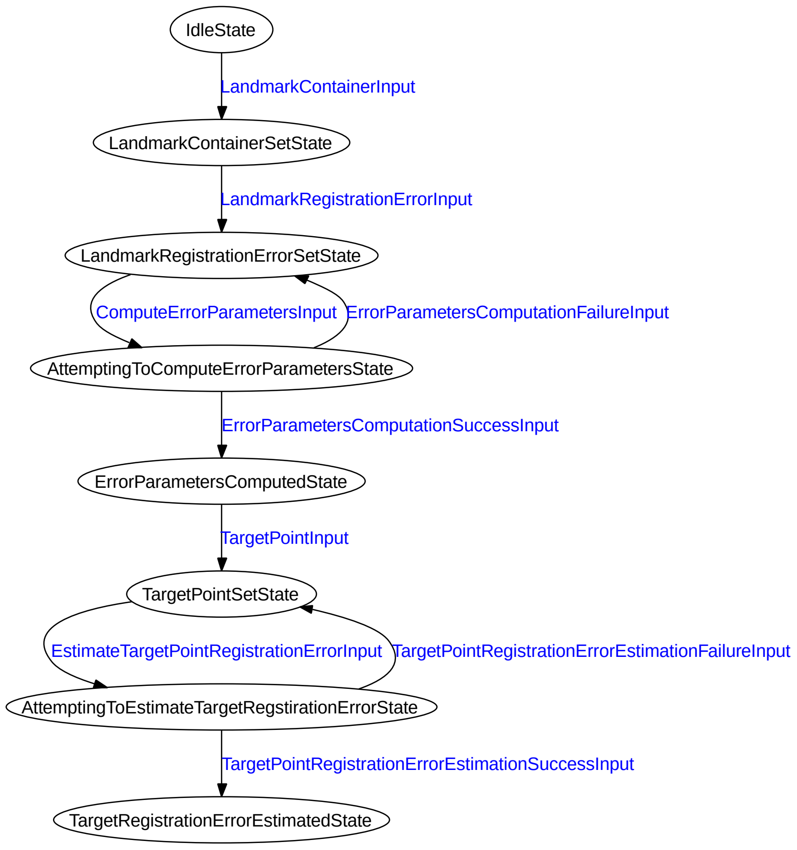

IdleState
LandmarkContainerInput
LandmarkContainerSetState
LandmarkRegistrationErrorInput
LandmarkRegistrationErrorSetState
ComputeErrorParametersInput
ErrorParametersComputationFailureInput
AttemptingToComputeErrorParametersState
ErrorParametersComputationSuccessInput
ErrorParametersComputedState
TargetPointInput
TargetPointSetState
EstimateTargetPointRegistrationErrorInput
TargetPointRegistrationErrorEstimationFailureInput
AttemptingToEstimateTargetRegstirationErrorState
TargetPointRegistrationErrorEstimationSuccessInput
TargetRegistrationErrorEstimatedState

Figure 14.2: Landmark Registration Error Estimator State Diagram.

1. *RequestSetLandmarkContainer(LandmarkContainerType)*: Method to set the landmark container.
2. *RequestSetTargetPoint(TargetPointType)*: Method to set the target point
3. *RequestSetLandmarkRegistrationError(ErrorType)*: Method to set the landmark registration error
4. *RequestComputeErrorParameters ( )*: Method to request computation of error parameters needed to compute the target point registration error
5. *RequestEstimateTargetPointRegistrationError ( )*: Method to request estimation of target point registration.
6. *RequestGetTargetPointRegistrationErrorEstimate()* : Method to get target point registration error. This method throws an event loaded with the target point registration error.

14.2.3 Example

The source code for this section can be found in the file
`Examples/LandmarkRegistrationErrorEstimation/ErrorEstimation1.cxx`.

This example illustrates how to estimate the registration error of a target point that has been registered using transformation parameters that were computed using landmark-based registration.

The error estimation is based on the closed-form equation developed by West et al. The target point registration error is dependent on the location of the target point, the registration error of the landmark points (root mean square error) and the configuration of the landmark points.

To use the IGSTK component for computing the registration error, the following `igstk::Landmark3DRegistrationErrorEstimator` header file must be added.

```
#include "igstkLandmark3DRegistrationErrorEstimator.h"
```

The registration error estimator type is defined and an object is instantiated.

```
  typedef igstk::Landmark3DRegistrationErrorEstimator   ErrorEstimatorType;

  ErrorEstimatorType::Pointer errorEstimator = ErrorEstimatorType::New();
```

The landmark point container is set as follows

```
  errorEstimator->RequestSetLandmarkContainer( fpointcontainer );
```

Next, the landmark registration error is set. The landmark registration component is used to compute this parameter. This error parameter basically the root mean square error of the landmark registration.

```
  landmarkRegistrationError = landmarkRegister->ComputeRMSError();
  errorEstimator->RequestSetLandmarkRegistrationError(
                                              landmarkRegistrationError );
```

Then, the other error parameters necessary for error estimation are computed by invoking the ComputeErrorParameter() method as shown below.

```
  errorEstimator->RequestComputeErrorParameters();
```

Next, the target point that we will be estimating the registration error is set

```
  TargetPointType targetPoint;
  targetPoint[0] =  10.0;
  targetPoint[1] =  20.0;
  targetPoint[2] =  8.0;
  errorEstimator->RequestSetTargetPoint( targetPoint );
```

Finally, the registration error for the target point is estimated.

```
  ErrorType       targetRegistrationError;
  errorEstimator->RequestEstimateTargetPointRegistrationError( );
```

To receive the error value, an observer is set up to listen to an event loaded with error value as follows

```
  ErrorEstimationGetErrorCallback::Pointer lrtcb =
                        ErrorEstimationGetErrorCallback::New();

  errorEstimator->AddObserver( igstk::LandmarkRegistrationErrorEvent(), lrtcb );
  errorEstimator->RequestGetTargetPointRegistrationErrorEstimate();

  if( !lrtcb->GetEventReceived() )
    {
    std::cerr << "LandmarkRegsistrationErrorEstimator class failed to "
              << "throw a landmark registration error event" << std::endl;
    return EXIT_FAILURE;
    }

  targetRegistrationError = lrtcb->GetError();
```

14.3 Conclusion

Registration is an essential component in image guided surgery applications. IGSTK provides 3D point-based registration tool for this purpose. This tool is based on a closed-form solution to the least-squares problem of computing transformation parameters between two coordinate systems. The accuracy of the computed transformation parameters needs to be verified before using them in actual application. For this purpose, IGSTK provides a robust error prediction tool that takes also into account landmark configuration.

CHAPTER

FIFTEEN

Calibration

In image guided surgery, a surgeon uses tracked tools for navigation or intervention. A general transform procedure called calibration is required to align the output of the tracking device to the specific operation points on the tool, such as the tip of a surgical needle or the end of its handle. For surgeons, these specific operation points are more important and intuitive than the positions of the trackers which are often embedded in the middle or at the hind of the tool.

DG: The paragraph above should be reorganized so that it does a better job of getting the main idea across. Starting from the second sentence:

These tools have elements embedded in them that the tracking system can measure the positions of: for magnetic tracking, the elements are small coils of wire, and for optical tracking small reflective markers or light-emitting diodes are used. The positions of these elements are usually not important to the surgeon, who would instead rather know the position of the tip of the tool or of the end of the tool's handle. The purpose of tool calibration is to measure the position of these important points on the tool, relative to the positions of the tracked elements (coils or markers) on the tools.

Generally speaking, calibration refers to the process of setting the magnitude of the output (or response) of a measuring instrument to the magnitude of the input property or attribute within a specified range of accuracy and precision [1]. When calibrating the tracker, those inputs are the positions and the orientation reported from the tracking device, and the outputs are the positions of the specific operation points. When calibrating both the tracker and an imaging device, such as an ultrasound probe, the inputs are the positions of the probe tracker and the image coordinate in the ultrasound imaging, and the outputs are the 3D coordinates of those image pixels in the global world coordinate system. Hence, in most cases, the calibration procedure produces the transform between the trackers and specific positions.

DG: Don't give the general definition of calibration, just describe what it means in the context of image-guided surgery.

In the IGSTK calibration package, several calibration classes are currently provided in the main repository and sandbox, including igstkPivotCalibration, igstkPrincipalAxisCalibration

[1] http://en.wikipedia.org/wiki/Calibration

and igstkLandmarkUltrasoundCalibration. Those calibration classes work closely with the tracker component to provide a safe and accurate environment for the image-guided surgical procedures. Some specialized features, such as the essential distortion calibration for the electro-magnetically tracking device, will also be provided following the requirement process of IGSTK framework.

DG: You should make a section called "Calibration in IGSTK" and move this paragraph to that section.

This chapter will firstly describe the rationale and design patterns for the existing calibration components. Next it will introduce the calibration data format used and its I/O classes in the sample application. Finally, it will discuss the future extension of the calibration package.

DG: You should have a section here called "Calibration in IGSTK" where you describe how calibration fits into the architecture of IGSTK. For example, you should give the names of the calibration classes, a short description of what each class is for, and you should describe how the calibration transform is used by the tracker.

15.1 Pivot Calibration

15.1.1 Introduction

The reported positions from both optical and electro-magnetic trackers are generally at fixed points on their sensors. For example, in the AURORA tracking system, the reported position for a single sensor is at the center of the sensor coil that is embedded in the surgical tool. When multiple sensors are used, the reported position is determined by the configuration SROM file that is stored in the tools. Generally, this position is a specific point that is rigidly related to the tracked tool.

For a tracked needle, its tip and the end of its handle are important for the surgeon. Those points provide an intuitive means to visualize and locate the tool's body. In an image-guided surgery application, a point at the tip of the tool is always used to locate the spatial position of landmark points, such as skin fiducials or the internal bifurcation positions of the vessels. Additionally, for the navigation and validation, the tip of the tracked needle or the guide-wire indicates the surgeon's point of focus. Thus, a tracked tool can also serve as a 'locator' or a 'pointer'. The transform between the internal sensor or marker's positions to those specific operation points is accomplished by a procedure called pivot calibration.

DG: When you use quotation marks in Latex, the first quotation mark must be a back-quote. For example: 'Quotation'.

Most tracking device manufacturers provide specific software to handle this pivot calibration; for example, Northern Digital Inc. provides '6D Architecture Aurora' and 'Toolviewer' for this purpose. This kind of software establishes communication, tracks the tools, visualizes the positions, and calculates the pivot calibration transform. The pivot calibration result is calculated before the experiment or the procedure, then used by the application for each specific

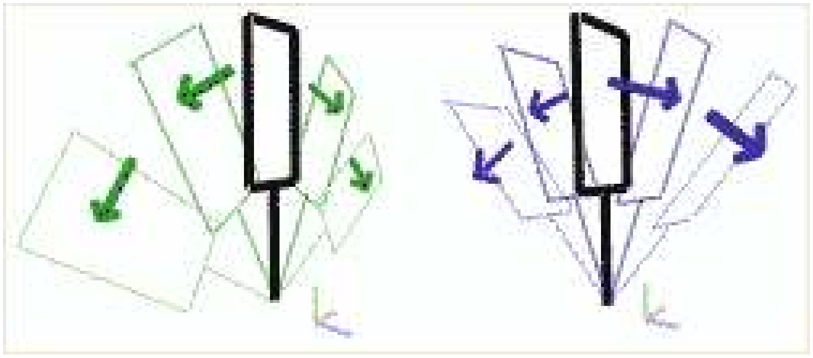


Figure 15.1: Pivot Calibration Routine.

tool. However, for the on-site pivot calibration, a general purpose class gives the developer more flexibility.

In a typical pivot procedure like the one shown in Figure 15.1 [2], the tip of the instrument is placed in a divot (a series of small holes) and the instrument is rotated back and forth (it pivots) while tracking data is collected with enough sample input, the transformation from the tracked sensor's point to the pivot point is calculated, along with the calibration error represented as a root mean square error.

15.1.2 Principle

DG: Add a short explanation to help the reader understand why this method works. For example, the following description is from Calvin R. Maurer, Jr., J. Michael Fitzpatrick, Matthew Y. Wang, Robert L. Galloway, Jr., Robert J. Maciunas, and George S. Allen, "Registration of Head Volume Images Using Implantable Fiducial Markers," IEEE Transactions on Medical Imaging 16(4):447-462, 1997.

"Each probe is calibrated by placing the probe tip in a fixed location and pivoting the probe about this fixed point. The position of he probe tip relative to the coordinate system of the probe attachment is determined by finding the most invariant point (in a least-squares sense) in these pivot motions."

The information from the tracker consists of the position and the orientation. Some systems also provide the measurement error at that position. Related with the original point, the position is located by a translation vector and a quaternion that represents the rotation from the default principal axis (mostly along the z-axis from the manufacturer settings). Depending on the sensor or sensors, the reported tracker information may have three, five, or six degrees of freedom (DOF). The first three degrees are represented by the translation, and the others are determined by the quaternion. The transformation from the original point to the tool tip in the tracking coordinate system is represented by:

[2] NDI 6D Architecture Aurora

$$\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{15.1}$$

In this equation, R is the rotation matrix, T is the translation vector and (x_0, y_0, z_0) is the pivot position. In most pivoting cases, the pivot position is the tip of the tool. Typically, we record several hundred samples while pivoting the tools. Equation 15.1 can be re-written as follows where the constraints of offset and (x_0, y_0, z_0) are arranged as:

$$\begin{aligned} r_{00} \cdot offset_x + r_{01} \cdot offset_y + r_{02} \cdot offset_z - 1 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 &= -t_x \\ r_{10} \cdot offset_x + r_{11} \cdot offset_y + r_{12} \cdot offset_z + 0 \cdot x_0 - 1 \cdot y_0 + 0 \cdot z_0 &= -t_y \\ r_{20} \cdot offset_x + r_{21} \cdot offset_y + r_{22} \cdot offset_z + 0 \cdot x_0 + 0 \cdot y_0 - 1 \cdot z_0 &= -t_z \end{aligned} \tag{15.2}$$

With several input samples, those equations can then be accumulated as:

$$M \cdot \begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ x_0 \\ y_0 \\ z_0 \end{bmatrix} = N \tag{15.3}$$

Since M is not a square matrix, the unknowns are solved using the singular value decomposition (SVD) or Moore-Penrose inverse:

$$\begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ x_0 \\ y_0 \\ z_0 \end{bmatrix} = (M^T \cdot M)^{-1} \cdot M^T \cdot N \tag{15.4}$$

Additionally, the root mean square error is computed as:

$$RMS = \sqrt{|M \cdot [offset_x \;\; offset_y \;\; offset_z \;\; x_0 \;\; y_0 \;\; z_0]^T - N|^2 / num} \tag{15.5}$$

where *num* is the number of samples. Note that in the pivot calibration procedure, the final transform only contains only a translation factor and no rotation factor.

When calculating the calibration only along the Z-axis only for a cylinder-like track tools, sometimes users may only want to get the calibration offset along the Z-axis. In this condition, the $offset_x$, $offset_y$ variables are restricted to 0 and the computation formulas are slightly changed.

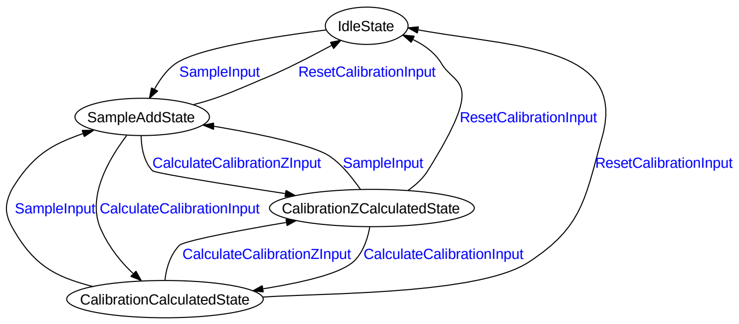

IdleState
SampleInput
ResetCalibrationInput
SampleAddState
ResetCalibrationInput
ResetCalibrationInput
CalculateCalibrationZInput
SampleInput
SampleInput
CalculateCalibrationInput
CalibrationZCalculatedState
CalculateCalibrationZInput
CalculateCalibrationInput
CalibrationCalculatedState

Figure 15.2: State Diagram of the igstkPivotCalibration class.

15.1.3 State Machine Diagram

Figure 15.2 is a state machine diagram of the `igstk::PivotCalibration` class. There are four states inside this class: initial 'Idle' state, 'SampleAdd' state, 'CalibrationCalculated' state and 'CalibrationZCalculated' state. The 'Idle' state is the initial state for all IGSTK classes. When the position samples from pivoting the tracker are input into the class, the internal state will be invoked to the 'SampleAdd' state. As described in the previous section, there are two ways to calculate the calibration matrix, either in the full translation mode or in the Z-axis only mode. The final stages are the 'CalibrationCalculated' and CalibrationZCalculated' states respectively. Only in these two states can the PivotCalibration class return a valid calibration transform.

15.1.4 Component Interface

The core functions of the igstkPivotCalibration class include:

1. Input the samples (translation and quaternion) from the pivoting trackers;

   - igstkPivotCalibration::RequestAddSample();

2. Calculate the calibration transform;

   - igstkPivotCalibration::RequestCalculateCalibration();

   - igstkPivotCalibration::RequestCalculateCalibrationZ();

3. Return the final calibration transform and pivot position;

   - igstkPivotCalibration::GetValidCalibration();

   - igstkPivotCalibration::GetCalibrationTransform();

   - igstkPivotCalibration::GetPivotPosition();

4. Calculate the root mean square error to evaluate whether a feasible calibration transform has been computed;

- igstkPivotCalibration::GetRootMeanSquareError();

5. Provide the convenient function to retrieve the input sample;

   - igstkPivotCalibration::GetNumberOfSamples();

   - igstkPivotCalibration::RequestGetInputSample();

6. Provide the convenient function to simulate the pivot position for any input translation and quaternion from the calculated calibration transform;

   - igstkPivotCalibration::RequestSimulatePivotPosition();

15.1.5 Example

The source code for this section can be found in the file
`Examples/PivotCalibration/PivotCalibration1.cxx`.

This example illustrates how to use IGSTK's pivot calibration class to determine a calibration matrix for the tracker tools.

To use the pivot calibration component, the header file for `igstk::PivotCalibration` should be added.

```
#include "igstkPivotCalibration.h"
```

After defining the headers, the main function implementation is started.

```
int main( int argc, char * argv[] )
{
```

All the necessary data types in the pivot calibration are defined. VersorType and VectorType are used to represent the quaternion and translation inputs from the tracker; PointType is used to represent the position coordinate of the specific point; and ErrorType is used to represent the root mean square error.

```
  typedef igstk::PivotCalibration           PivotCalibrationType;
  typedef PivotCalibrationType::VersorType  VersorType;
  typedef PivotCalibrationType::VectorType  VectorType;
  typedef PivotCalibrationType::PointType   PointType;
  typedef PivotCalibrationType::ErrorType   ErrorType;
  typedef itk::Logger                       LoggerType;
  typedef itk::StdStreamLogOutput           LogOutputType;
```

At the beginning, a pivot calibration class is initialized as follows:

```
  PivotCalibrationType::Pointer pivot = PivotCalibrationType::New();
```

A logger is then created for logging the process of calibration computation, and then attached to the pivot calibration class

```
LoggerType::Pointer                          logger = LoggerType::New();
LogOutputType::Pointer                       logOutput = LogOutputType::New();

logOutput->SetStream( std::cout );
logger->AddLogOutput( logOutput );
logger->SetPriorityLevel( itk::Logger::DEBUG );

pivot->SetLogger( logger );
```

To use the pivot calibration class, some input samples from the tracker should be provide. Those samples come directly from tracker tools. In our example, those samples are read from the record data file in the IGSTK data directory.

```
input.open( argv[1] );

if (input.is_open() == 1)
  {
  std::cout << "PivotCalibration data open sucessully!" << std::endl;
  }
else
  {
  std::cout << "PivotCalibration data open error!" << std::endl;

  return EXIT_FAILURE;
  }
```

Before the computation, it is better to reset the calibration class to to remove all necessary information which may come from the previous codes.

```
pivot->RequestReset();
```

Then, the sample frame is read from the data file and input to calibration class

```
while ( !input.eof())
  {
  double vx;
  double vy;
  double vz;
  double vw;

  input >> frame >> temp >> time;
  input >> pos[0] >> pos[1] >> pos[2];
  input >> vw >> vx >> vy >> vz;
```

```
    versor.Set( vx, vy, vz, vw );
    pivot->RequestAddSample( versor, pos );
    }
```

After this, a simple request will invole the class to compute the calibration transform.

```
  pivot->RequestCalculateCalibration();
```

Before the final calibration transform is retrieved, the user should check the tag to see whether a valid calibration has been computed. The final calibration result is stored in the translation factor in the transform matrix. The pivot position is also retrievable. The sample code is as follows:

```
  if ( !pivot->GetValidPivotCalibration())
    {
    std::cout << "No valid calibration!" << std::endl;

    return EXIT_FAILURE;
    }
  else
    {

    // Get the calibration transformation
    VectorType translation = pivot->GetCalibrationTransform().GetTranslation();

    // Get the pivot focus position
    PointType position = pivot->GetPivotPosition();

    // Get the calibration RMS error
    ErrorType error = pivot->GetRootMeanSquareError();

    // Dump the calibration class information
    std::cout << "PivotCalibration: " << std::endl;
    std::cout << "NumberOfSamples: " << pivot->GetNumberOfSamples()
                                     << std::endl;
    std::cout << "Translation: " << translation << std::endl;
    std::cout << "Pivot Position: " << position << std::endl;
    std::cout << "Calibration RMS: " << error << std::endl;

    }
```

For only computing the calibration along Z-axis, another function is used instead:

```
  pivot->RequestCalculateCalibrationZ();
```

15.2 Principal Axis Calibration

15.2.1 Introduction

The pivot calibration provides only the translation information of the tracked tool and thus is based on the assumption that the geometry coordinate and the default tracking coordinate are the same. In most cases, the principal axis is along the Z-axis. If the principal axis of the tracked tool is not well aligned with the geometry representation, a rotation transform is required to work together with the pivot calibration routine to construct the full transform matrix. Since IGSTK works closely with ITK, and the transform is based on the itk::Vector and itk::Versor classes, the itk::Versor can be used to represent the rotation directly. For example, if the tracked tool's principal axis is along the Z-axis, but the spatial geometry object is along the Y-axis, an itk::Versor::SetRotationAroundX() function solves the problem. This approach works for these specific rotations, but for an arbitrary rotation between two unspecific directions it is hard to use a combination of rotations around the X, Y or Z-axes to produce the exact result. In this case, a general purpose igstkPrincipalAxisCalibration class provides an intuitive means to set the initial and desired orientations (tracker tool's principal axis and spatial object's principal axis) and to return the rotation between them.

For cylindrical tracker tools, the spatial principal axis runs mostly along the cylinder geometry axis. In some different configurations, this alignment may change. For example, the ITK spatial object's default axis is along the Y-axis, but the IGSTK spatial object's default axis is along the Z-axis. For the 5DOF and 6DOF trackers, the default principal axis in the tracking coordinate system, as defined by most hardware manufacturers, is along the Z-axis. Note that this default principal axis in the tracking coordinate system often is not along the tracker tool's geometry principal axis. A combination tracker tool is shown in Figures 15.3 and 15.4 [3]. For the tracker handle, the optical markers are arranged along the handle and the tracking coordinate system's principal axis is well aligned with the spatial geometry shape. But for the tracker probe tip, the specially designed curves make them different. In an image-guided surgery application, these probe tips are of greater concern to the surgeon, for they are real operational parts and will touch the patient. When the tracker probe tip and the handle are attached together, the principal axis in the tracking coordinate system is different than the operational part's principal axis. For those tracker tools, an igstkPrincipalAxisCalibration class provides an intuitive means to calculate the rotation matrix.

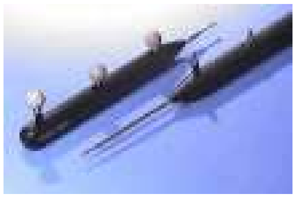


Figure 15.3: Tracker Handle

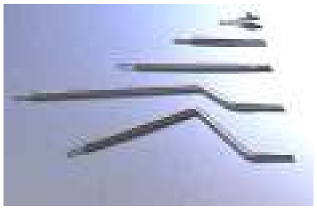


Figure 15.4: Tracker Probe Tip

[3] Traxtal Inc.

15.2.2 Principle

The purpose of computing the principal axis calibration is to find the rotation between two defined orientations. Our method is based on the rotation matrix multiplication. In IGSTK, the rotation factor is stored in the quaternion format, and the convert between the quaternion and the rotation matrix is unified. When a point is rotated, in the mathematically representation, the vector is multiplied by a rotation matrix. So, the difference between two orientations can be calculated by the divide operation of the two matrices, as follows:

$$M = M_{ori1} \cdot M_{ori2}^{-1} \tag{15.6}$$

M_{ori1} is the rotation matrix from the desired orientation, and M_{ori2} is the rotation matrix from the initial orientation.

The igstkPrincipalAxisCalibration class provides an intuitive means to set the orientation, which is defined by the principal axis and the normal axis of the tool. The principal axis is generally the major axis along the cylindrical tools that are very popular in clinical procedures. The normal vector defines the view-up direction of the system coordinate. Those two parameters are easy to measure by the geometry shape design profile of the tools. The rotation matrix can be built from the principal axis and normal vector using the following equation:

$$M_{ori} = \begin{bmatrix} p_x & p_y & p_z \\ n_x & n_y & n_z \\ l_x & l_y & l_z \end{bmatrix} \tag{15.7}$$

where (p_x, p_y, p_z) is the normalized principal axis vector, (n_x, n_y, n_z) is the normalized orthogonal view-up vector, and (l_x, l_y, l_z) is the vector along the third direction which is made by the cross production of the first two vectors.

15.2.3 State Machine Diagram

Figure 15.5 illustrates the State Machine of the `igstk::PrincipalAxisCalibration` class. There are five states inside this class: ’Idle,’ ’InitialOrientationSet,’ ’DesiredOrientationSet,’ ’OrientationAllSet,’ and ’RotationCalculated.’ ’Idle’ is the initial state for all IGSTK classes. Subsequently, a class’s state will be ’InitialOrientationSet,’ ’DesiredOrientationSet,’ or ’OrientationAllSet,’ depending on whether the initial or desired orientations of the tracker are put into the class. Only when a class has reached the ’OrientationAllSet’ state will a request to calculate the calibration function bring the class into the final ’RotationCalculated’ state.

15.2.4 Component Interface

From the description, the core functions of the igstkPrincipalAxisCalibration class include:

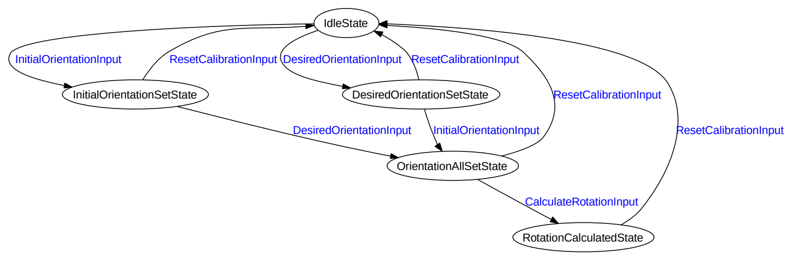


Figure 15.5: State Diagram of the PrincipalAxisCalibration class.

1. Set the initial principal axis and view-up normal;

   - igstkPrincipalAxisCalibration::RequestSetInitialOrientation();

2. Set the desired principal axis and view-up normal;

   - igstkPrincipalAxisCalibration::RequestSetDesiredOrientation();

3. Automatically adjust the plane normal to make it perpendicular with the principal axis;

4. Calculate the rotation matrix from those two orientations;

   - igstkPrincipalAxisCalibration::RequestCalculateRotation();

15.2.5 Example

The source code for this section can be found in the file
`Examples/PrincipalAxisCalibration/PrincipalAxisCalibration1.cxx`.

This example illustrates how to use IGSTK's princiapl axis calibration class to determine the rotation matrix for the tracker tools.

To use the principal axis calibration component, the header file for `igstk::PrincipalAxisCalibration` should be added.

```
#include "igstkPrincipalAxisCalibration.h"
```

After defining the headers, the main function implementation is started.

```
int main( int argc, char * argv[] )
{
```

All the necessary data types in the principal axis calibration are defined. VectorType and CovariantVectorType are used to represent the vectors along the principal axis and the plane normal.

```
typedef igstk::PrincipalAxisCalibration   PrincipalAxisCalibrationType;

typedef PrincipalAxisCalibrationType::VectorType          VectorType;
typedef PrincipalAxisCalibrationType::CovariantVectorType CovariantVectorType;
typedef itk::Logger                                       LoggerType;
typedef itk::StdStreamLogOutput                           LogOutputType;
```

At the beginning, a principal axis calibration class is initialized as follows:

```
PrincipalAxisCalibrationType::Pointer principal
                                      = PrincipalAxisCalibrationType::New();
```

A logger is then created for logging the process of calibration computation, and then attached to the principal axis calibration class

```
LoggerType::Pointer                         logger = LoggerType::New();
LogOutputType::Pointer                   logOutput = LogOutputType::New();

logOutput->SetStream( std::cout );
logger->AddLogOutput( logOutput );
logger->SetPriorityLevel( itk::Logger::DEBUG );

principal->SetLogger( logger );
```

Before the computation, it is better to reset the calibration class to to remove all necessary information which may come from the previous codes.

```
principal->RequestReset();
```

Some parameters, such as axis and normal, are defined to store the input information to determine the initial and desired orientations

```
VectorType          axis;
CovariantVectorType normal;
```

An initial orientation is given as the default one for the tracker tools

```
axis[0] = 0.0;
axis[1] = 1.0;
axis[2] = 0.0;
normal[0] = 0.0;
normal[1] = 0.0;
normal[2] = 1.0;
principal->RequestSetInitialOrientation( axis, normal );
```

A desired orientation of the tracker tools are also specified

```
axis[0] = 0.0;
axis[1] = 0.0;
axis[2] = 1.0;
normal[0] = 0.0;
normal[1] = 1.0;
normal[2] = 0.3;
principal->RequestSetDesiredOrientation( axis, normal );
```

Then a RequestCalculateRotation function is invoked to compute the final results

```
principal->RequestCalculateRotation();
```

Before the final calibration transform is retrieved, the user should check the tag to see whether a valid calibration has been computed. The sample code is as follows:

```
if ( !principal->GetValidRotation())
  {
  std::cout << "No valid calibration!" << std::endl;

  return EXIT_FAILURE;
  }
else
  {
  std::cout << "Initial Principal Axis:"
            << principal->GetInitialPrincipalAxis() << std::endl;
  std::cout << "Initial Plane Normal:"
            << principal->GetInitialPlaneNormal() << std::endl;
  std::cout << "Desired Principal Axis:"
            << principal->GetDesiredPrincipalAxis() << std::endl;
  std::cout << "Desired Plane Normal:"
            << principal->GetDesiredPlaneNormal() << std::endl;
  std::cout << "Calibration Transform:"
            << principal->GetCalibrationTransform() << std::endl;

  principal->Print( std::cout);

  }
```

15.3 Calibration Data I/O

15.3.1 Data Format

IGSTK needs a common file format for storing tool calibration transformations. Because the tools must be calibrated before the surgery, it is necessary to verify that the correct calibration file is applied to the correct tool.

The tool calibration file must contain the following information:

1. The date and time that the calibration was performed (in DICOM date/time format: YYYYMMDD HHMMSS.SSSS);
2. Information about the method and the equipment used to calibrate the tool;
3. Identification information for the tool, including the manufacturer, part number, and serial number for the tool;
4. The transform type (which will be limited to rigid quaternion transforms for now);
5. The transform parameters;
6. A description of the error associated with the calibrated transform.

A sample pre-computed calibration file is like:

DG: The description of the file format might be better as an appendix, at the end of the book.

```
<?xml version=”1.0”?>
<IGSTKFile type=”ToolCalibration” version=”0.1”>
  <Creation date=”20050824” time=”070907.0705” method=”PivotCalibration”
/>
 <Tool type=”Pointer” manufacturer=”Traxtal” partNumber=”023-X” serialNum-
ber=”200501268” />
 <Transform type=”Rigid3D”>
   <ParameterNames>
    translation_x translation_y translation_z
    quaternion_x quaternion_y quaternion_z quaternion_w
   </ParameterNames>
   <ParameterValues>
    5.0 2.0 3.0
            9.7467943448089631 -0.20519567041703082 0.9233805168766388
0.30779350562554625
   </ParameterValues>
   <ErrorParameterNames>
    rms
   </ErrorParameterNames>
   <ErrorParameterValues>
```

```
      0.187876234
     </ErrorParameterValues>
   </Transform>
 <IGSTKFileCRC32>
   4f6a3b2d
 </IGSTKFileCRC32>
 </IGSTKFile>
```

The *CRC*32 is a 32-bit *CRC* that can be checked to validate the integrity of the data. The *CRC* is calculated from the start of the <IGSTKFile> tag to the end of the </IGSTKFile> tag, not for the whole file. A proper *DTD* is specified for the above XML file. The following error parameters could be defined:

1. *rms*: the root-mean-square error for the translation (Fiducical Registration Error for landmark registration)
2. *centroid_x*, *_y*, *_z*: the landmark centroid (or the center of rotation that was used for image registration)
3. Additional parameters to express rotational error

As the calibration data format is to handle all calibration information and is still improving, it is better to check the online update of the current calibration file format [4].

15.3.2 Data Reader

For the convenience of reading/writing the calibration data, IGSTK also provides some utility classes to handle the input and output of calibration data. Figure 15.6 illustrates the State Machine of the `igstk::PivotCalibrationReader` class. This class provides the function to input the calibration from the pivoting routine.

15.3.3 Example

The source code for this section can be found in the file
`Examples/PivotCalibrationReader/PivotCalibrationReader1.cxx`.

This example illustrates how to use IGSTK's pivot calibration reader class to read the calibration matrix from an offline calibration file.

To use the pivot calibration reader component, the header file for `igstk::PivotCalibrationReader` should be added.

[4] http://public.kitware.com/IGSTKWIKI/index.php/Calibration_Data

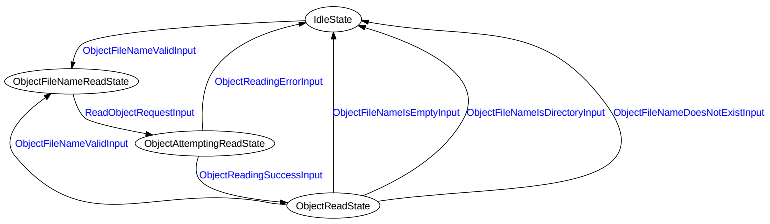


Figure 15.6: State Diagram of the PivotCalibrationReader class.

```
#include "igstkPivotCalibrationReader.h"
```

At the very beginning of the program, two kinds of event and observers are defined to track the information from the reader. The first event is igstk::CalibrationModifiedEvent, which is to retrieve the calibration class from the reader. The second one is igstk::StringEvent, which is to retrieve some string-like information from the general calibration class.

```
namespace ToolCalibrationTest
{
igstkObserverMacro(Calibration,::igstk::CalibrationModifiedEvent,
                                           ::igstk::PivotCalibration::Pointer)
igstkObserverMacro(String,::igstk::StringEvent,std::string)
}
```

After defining the headers, the main function implementation is started.

```
int main( int argc, char * argv[] )
{
```

A pivot calibration reader is created and then a logger is attached.

```
  LoggerType::Pointer                       logger = LoggerType::New();
  LogOutputType::Pointer                    logOutput = LogOutputType::New();

  logOutput->SetStream( std::cout );
  logger->AddLogOutput( logOutput );
  logger->SetPriorityLevel( itk::Logger::DEBUG );

  // Create the pivot calibration reader and attach the logger
  igstk::PivotCalibrationReader::Pointer reader =
                              igstk::PivotCalibrationReader::New();

  reader->SetLogger( logger );
```

The pivot calibration file's name is passed through the argument. After the filename is designated, a RequestReadObject function is invoked to parse the data file. The information in the reader can be easily dumped by default Print function.

```
reader->RequestSetFileName(argv[1]);
reader->RequestReadObject();

reader->Print(std::cout);
```

To retrieve the whole calibration data information, the previous defined observer is attached to the reader class. After the RequestGetCalibration function is called, the calibration info is passed by observer's GetCalibration function. The sample code is as follows.

```
typedef ToolCalibrationTest::CalibrationObserver CalibrationObserverType;
CalibrationObserverType::Pointer calibrationObserver
                                            = CalibrationObserverType::New();

reader->AddObserver(::igstk::CalibrationModifiedEvent(),calibrationObserver);
reader->RequestGetCalibration();

igstk::PivotCalibration::Pointer calibration = NULL;

std::cout << "Testing Calibration: ";
if( calibrationObserver->GotCalibration() )
  {
  calibration = calibrationObserver->GetCalibration();
  }
else
  {
  std::cout << "No calibration!" << std::endl;
  return EXIT_FAILURE;
  }

std::cout << "[PASSED]" << std::endl;
```

To retrieve some specific information, like serial number and manufacturer, from the trackers, another string even and observer is attached to the calibration class we just get. For each request, the information content will be passed by observer's GetString function. Some sample codes are shown as below:

```
typedef ToolCalibrationTest::StringObserver StringObserverType;
StringObserverType::Pointer stringObserver = StringObserverType::New();
calibration->AddObserver( ::igstk::StringEvent(), stringObserver );

std::cout << "Testing Date: ";
calibration->RequestGetDate();
if(stringObserver->GotString())
```

```
    {
    std::cout << stringObserver->GetString().c_str() << std::endl;
    std::cout << "[PASSED]" << std::endl;
    }
  else
    {
    std::cout << "No date!" << std::endl;
    return EXIT_FAILURE;
    }

  std::cout << "Testing Manufacturer: ";
  calibration->RequestGetToolManufacturer();
  if(stringObserver->GotString())
    {
    std::cout << stringObserver->GetString().c_str() << std::endl;
    std::cout << "[PASSED]" << std::endl;
    }
  else
    {
    std::cout << "No tool manufacturer!" << std::endl;
    return EXIT_FAILURE;
    }

  std::cout << "Testing Serial Number: ";
  calibration->RequestGetToolSerialNumber();
  if(stringObserver->GotString())
    {
    std::cout << stringObserver->GetString().c_str() << std::endl;
    std::cout << "[PASSED]" << std::endl;
    }
  else
    {
    std::cout << "No tool serial number!" << std::endl;
    return EXIT_FAILURE;
    }
```

15.4 Future Extension

Calibration, as well as registration, play an important role in the IGSTK toolkit. It serves like a broker between the tracker device and the physical and image space, and its role is to precisely guide the surgical tools and display them in the correct positions.

Currently igstkPivotCalibration, igstkPrincipalAxisCalibration and igstkLandmarkUltrasound-Calibration are implemented in IGSTK's main and sandbox repositories. These classes works with surgical tracker tools and a tracked ultrasound probe. The calibration data I/O classes, which provide the off-line processing of those calibration transforms, are another important part for calibration components. Definitely it covers only a small part of the calibration field. Following IGSTK's requirement-driven implementation style, some other features, such as distortion calibration for the specific electro-magnetically tracking, can be provided when it is required by the user.

Part III

User Guide

CHAPTER

SIXTEEN

HelloWorld

This example illustrates the minimal applications that can be written using IGSTK. The application uses three main components. They are the View, the SpatialObjects and the Tracker. The View is the visualization window that is presented to the user in the graphical user interface (GUI) of the application. The SpatialObjects are used for representing geometrical shapes in the scene of the surgical room. In this simplified example, a cylinder and a sphere SpatialObjects are used. The Tracker is the device that provides position and orientation information about some of the objects in the scene. A Tracker can track multiple objects, and each one of them is referred as a TrackerTool. In this minimal example we use a MouseTracker that is a class intended mainly for demonstration and debugging purposes. This tracker get the values of positions from the position of the mouse on the screen. The position values are then passed to the sphere object in the scene. The MouseTracker is not intended to be used in a real image guided surgery application.

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

To add a graphical user interface to the application, we use FLTK. FLTK is a a light weight cross-platform GUI toolkit. FLTK stores a description of an interface in files with extension .fl. The FLTK tool *fluid* takes this file and uses it for generating C++ code in two files. One header file with extension .h, and an implementation file with extension .cxx. In order to use that GUI from the main program of our application we must include the header file generated by fluid. This is done in the following line.

```
#include "HelloWorldGUI.h"
```

The geometrical description of the Cylinder and the Sphere in the scene are managed by SpatialObjects. For this purpose we need the two classes `igstk::EllipsoidObject` and `igstk::CylinderObject`. Their two header files are included below.

```
#include "igstkEllipsoidObject.h"
#include "igstkCylinderObject.h"
```

The visual representation of SpatialObjects in the visualization window is created using SpatialObject Representation classes. Every SpatialObject has one or several representation objects associated with it. We include now the header files of the igstk::EllipsoidObjectRepresentation and igstk::CylinderObjectRepresentation.

```
#include "igstkEllipsoidObjectRepresentation.h"
#include "igstkCylinderObjectRepresentation.h"
```

As stated above, the tracker in this minimal application is represented by a igstk::MouseTracker. This class provides the same interface of a real tracking device but with the convenience of running based on the movement of the mouse in the screen. The header file of this class is included below.

```
#include "igstkMouseTracker.h"
```

Since image guided surgery applications are used in a critical environment, it is quite important to be able to trace the behavior of the application during the intervention. For this purpose IGSTK uses a igstk::Logger class and some helpers. The logger is a class that receives messages from IGSTK classes and forward those messages to LoggerOutput classes. Typical logger output classes are the standard output, a file and a popup window. The Logger classes and their helpers are taken from the Insight Toolkit (ITK).

```
#include "itkLogger.h"
#include "itkStdStreamLogOutput.h"
```

We are now ready for writing the code of the actual application. Of couse we start with the classical main() function.

```
int main(int , char** )
{
```

The first IGSTK command to be invoked in an application is the one that initialize the parameters of the clock. Timing is critical for all the operations performed in an IGS application. Timing signals make possible to synchronize the operation of different components and to ensure that the scene that is rendered on the screen actually displays a consistent state of the environment on the operating room.

```
  igstk::RealTimeClock::Initialize();
```

First, we instantiate the GUI application.

```
  HelloWorldGUI * m_GUI = new HelloWorldGUI();
```

Next, we instantiate the ellipsoidal spatial object that we will be attaching to the tracker.

```
igstk::EllipsoidObject::Pointer ellipsoid = igstk::EllipsoidObject::New();
```

The ellipsoid radius can be set to one in all dimensions (X,Y and Z) using the SetRadius member function as follows.

```
ellipsoid->SetRadius(1,1,1);
```

To visualize the ellipsoid spatial object, an object representation class is created and the ellipsoid spatial object is added to it.

```
igstk::EllipsoidObjectRepresentation::Pointer
      ellipsoidRepresentation = igstk::EllipsoidObjectRepresentation::New();
ellipsoidRepresentation->RequestSetEllipsoidObject( ellipsoid );
ellipsoidRepresentation->SetColor(0.0,1.0,0.0);
ellipsoidRepresentation->SetOpacity(1.0);
```

Similarly, a cylinder spatial object and cylinder spatial object representation object are instantiated as follows.

```
igstk::CylinderObject::Pointer cylinder = igstk::CylinderObject::New();
cylinder->SetRadius(0.1);
cylinder->SetHeight(3);

igstk::CylinderObjectRepresentation::Pointer
        cylinderRepresentation = igstk::CylinderObjectRepresentation::New();
cylinderRepresentation->RequestSetCylinderObject( cylinder );
cylinderRepresentation->SetColor(1.0,0.0,0.0);
cylinderRepresentation->SetOpacity(1.0);
```

Next, the spatial objects are added to the view, and the camera position of is reset to observe all objects in the scene.

```
m_GUI->Display->RequestAddObject( ellipsoidRepresentation );
m_GUI->Display->RequestAddObject( cylinderRepresentation );
m_GUI->Display->RequestResetCamera();
m_GUI->Display->Update();
```

Function `RequestEnableInteractions()` allows the user to interactively manipulate (rotate, pan, zoomm etc.) the camera. For `igstk::View2D` class, `vtkInteractorStyleImage` is used; For `igstk::View3D` class, `vtkInteractorStyleTrackballCamera` is used. In IGSTK, the keyboard events are disabled, so it doesn't support the original VTK key-mouse-combined interactions. In summary the mouse events are as follows: Left button click triggers pick event; Left button hold rotates the camera, in `igstk::View2D`, camera direction is always perpendicular to image plane, so there is no rotational movement available for `igstk::View2D`; Middle mouse button pans the camera; Right mouse button dollys the camera.

```
m_GUI->Display->RequestEnableInteractions();
```

The following code instantiate a new mouse tracker and initialize it. The scale factor is just a number to scale down the movement of the tracked object in the scene.

```
igstk::MouseTracker::Pointer tracker = igstk::MouseTracker::New();
tracker->Open();
tracker->Initialize();
tracker->SetScaleFactor( 100.0 );
```

Now we attach previously created spatial object to the tracker and set the tracker to monitor the mouse events from the user interface. The tool port and tool number is naming convention from NDI trackers. *Reference to tracker chapter* object in the scene.

```
const unsigned int toolPort = 0;
const unsigned int toolNumber = 0;
tracker->AttachObjectToTrackerTool( toolPort, toolNumber, ellipsoid );
m_GUI->SetTracker( tracker );
```

Now we setup a logger. We will direct the log output to both the standard output (std::cout) and a file (log.txt). *need reference to logger chapter about priority level*

```
itk::Logger::Pointer logger = itk::Logger::New();
itk::StdStreamLogOutput::Pointer logOutput = itk::StdStreamLogOutput::New();
itk::StdStreamLogOutput::Pointer fileOutput = itk::StdStreamLogOutput::New();

logOutput->SetStream( std::cout );
logger->AddLogOutput( logOutput );
logger->SetPriorityLevel( itk::Logger::DEBUG );

std::ofstream ofs( "log.txt" );
fileOutput->SetStream( ofs );
logger->AddLogOutput( fileOutput );
```

By connecting the logger to the Display and the Tracker, messages from the these components will be redirected to the logger.

```
m_GUI->Display->SetLogger( logger );
tracker->SetLogger( logger );
```

Next, we set the refresh frequency of the display window. After we call the `RequestStart()` function, the pulse generator inside the display window will start ticking, and call the display to update itself 60 times per second.

```
m_GUI->Display->RequestSetRefreshRate( 60 );
m_GUI->Display->RequestStart();
```

All application should includes the following code. This is the main event loop of the application. First it checks if the application is aborted by user, if not, it calls for the `igstk::PulseGenerator` to check its time out.

```
while( !m_GUI->HasQuitted() )
  {
  Fl::wait(0.001);
  igstk::PulseGenerator::CheckTimeouts();
  }
```

Finally, before exiting the application, the tracker is properly closed and other clean up procedures are executed.

```
tracker->StopTracking();
tracker->Close();
delete m_GUI;
ofs.close();
return EXIT_SUCCESS;
```

When one build IGSTK with CMake option "IGSTK_BUILD_EXAMPLES" on, the "HelloWorld" application will be built automatically with the toolkit. You will see a user interface like Figure 16.1 when you run the application. If you press the "Tracking" button on the lower left, the sphere will start following the mouse cursor.

A logging file "log.txt" will also be created. The file contains logging output statments similar to the following :

```
...
24445310681.509983 : (DEBUG) draw() called ...
24445310681.511478 : (DEBUG) UpdateSize() called ...
24445310681.526333 : (DEBUG) Tracker::StartTracking called ...
24445310681.528706 : (DEBUG) State transition is being made : (...omited...)
24445310681.534176 : (DEBUG) Tracker::AttemptToStartTrackingProcessing called ...
24445310681.537331 : (DEBUG) State transition is being made : (...omited...)
24445310681.54253  : (DEBUG) Tracker::StartTrackingSuccessProcessing called ...
24445310681.545425 : (DEBUG) Tracker::EnterTrackingStateProcessing called ...
...
```

These are the log messages from different IGSTK classes. It gives you information on time stamp, priority level, function calls, state machine transitions. This log file can be used for debugging purposes and clinical procedure reviews.

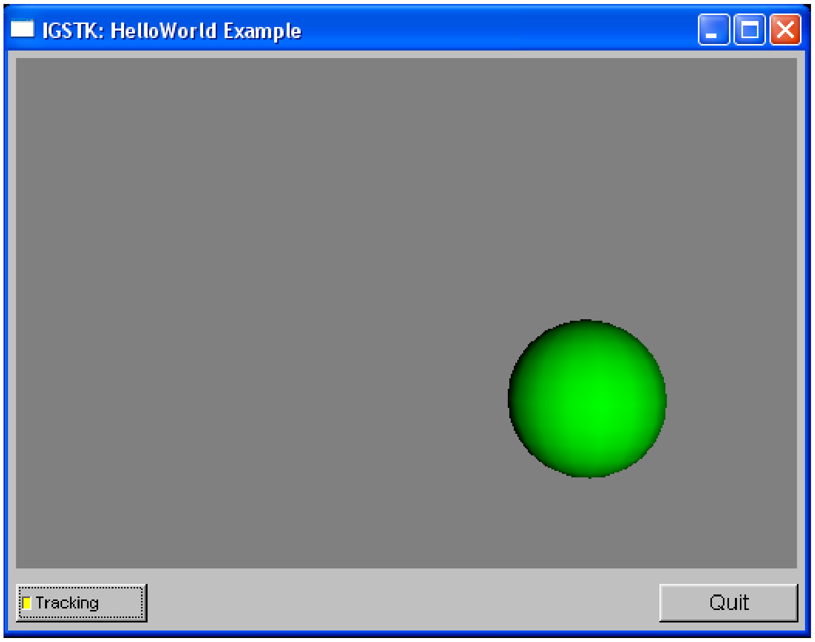


Figure 16.1: Screen shot of ”Hello World” example.

CHAPTER

SEVENTEEN

TwoViews

CHAPTER

EIGHTEEN

FourViews

CHAPTER

NINETEEN

FourViewsAndTracking

CHAPTER

TWENTY

FourViewsTrackingWithCT

Part IV

Example Applications

CHAPTER

TWENTYONE

Needle Biopsy

In this part of the book, we will show three example applications developed using IGSTK. These applications come from our observation of clinical procedures, such as needle biopsy, ultra-sound guided radio frequency ablation, and robot assisted needle placement. You will be able to learn the concepts and work flows of these common image-guided procedures as well as how to implement these applications under the IGSTK framework.

The first application is image guided needle biopsy. The definition for needle biopsy procedure on Society of Interventional Radiology website is:

> Needle biopsy is a medical test performed by interventional radiologists to identify the cause of a lump or mass, or other abnormal condition in the body. During the procedure, the doctor inserts a small needle, guided by X-ray or other imaging techniques, into the abnormal area. A sample of tissue is removed and given to a pathologist who looks at it under a microscope to determine what the abnormality is – for example, cancer, a noncancerous tumor, infection, or scar.(`http://www.sirweb.org/patPub/needleBiopsy.shtml`)

Needle biopsy is a widely used procedure for lung, breast, liver, and prostate cancer diagnosis. A typical image-guided needle biopsy procedure involves first acquiring a pre-operative CT image and then registering the CT image to the patient coordinate system. For this purpose, fiducial-based rigid body registration techniques are commonly used. During the biopsy phase, the needle is tracked by an optical tracking device with real-time visualization of its location overlaid on top of the CT image. This overlay image provides guidance to the surgeon for better targeting of the needle to its desired location.

Figure 21.1 shows the setup of the application. You will need the NDI Vicra tracker to run the program. You can also modify the program to use other supported trackers.

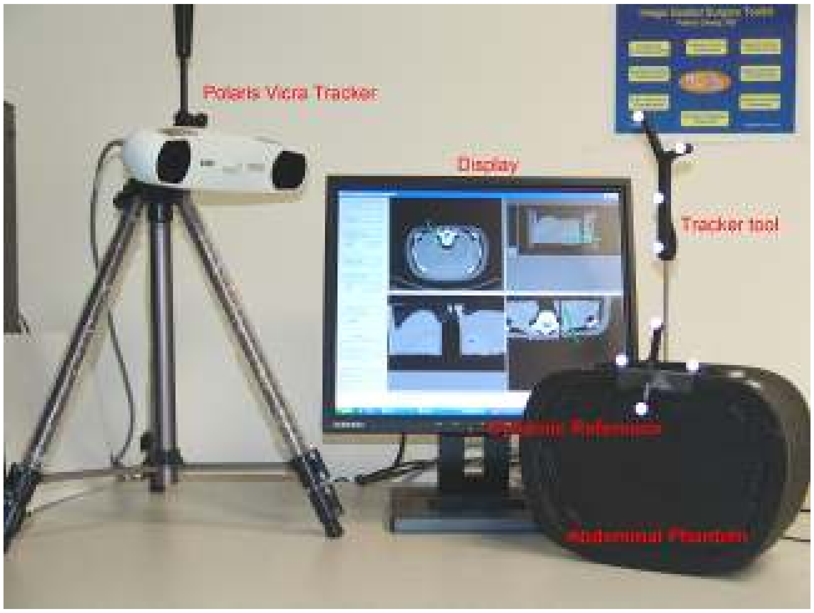


Figure 21.1: System setup for needle biopsy application.

21.1 Running the Application

This application can be found in the Examples/NeedleBiopsy. In order to build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you need to make sure that IGSTK_USE_FLTK is turned ON when running CMake.

The following steps outline the work flow or operating sequences of this application.

1. Obtain the patient demographic information (name, etc.).
2. Load in the pre-operative CT image using the DICOM file format. Fiducials (small markers) are usually placed on the anatomy prior to the scan for landmark based registration in Steps 4-7.
3. Verify the patient information against the information in the image. Prompt the surgeon if there is a discrepancy. This step is typical of the error checking that should be done and one should assume that if anything can go wrong it will go wrong and safeguards should be provided.
4. Identify the image landmarks by going through the CT image slices and selecting the fiducials using the mouse. For paired-point based registration, at least three points are required although at least four are preferred.
5. Initialize the tracking device.

6. Add patient landmarks by touching the physical fiducials attached to patient using the tracked pointer device.
7. Perform the image to patient landmark registration.
8. Path planning. The surgeon will select a target point and an entry point to plan the path for the needle puncture.
9. Provide a real-time display of the overlay of the needle probe and pre-operative images in the quadrant viewer window during the biopsy procedure.

21.2 Implementation

State machine is very important and distinct feature in IGSTK. Although it seems to be very complicated to many people, we are trying to provide some guidance here to help users to familiarize this concept and design pattern.

21.2.1 State Machine in Application

Given that IGSTK is intended for developing applications that will be used to treat patients, robustness and quality are the highest priorities in the design of the toolkit. To minimize the risk of harm to the patient resulting from misuse of the classes, IGSTK incorporates state machine design pattern into its components. All IGSTK components are governed by a state machine. A state machine is contained within the class to control the access to the class. Components are always in a valid state to ensure they will perform in a predictable manner. The use of a state machine also helps enforce high quality standards for code coverage and run-time validation.

In this example, state machine is being implemented at the application level(Figure 21.2 shows the partial state machine diagram of this application). While this is not mandatory, it is strongly recommended when using IGSTK. A state machine architecture gives the application developer an easier way to prototype the application and to control the work flow of the surgical procedure, and also adds an extra layer of security to the application to make it more robust. The following sections demonstrate how to write an application using the IGSTK framework.

21.2.2 Mapping clinical work flow to state machine

The first step to develop an application is to analyze the surgical procedure and develop a minimal specification. By analyzing a typical needle biopsy procedure, we identify a serial of tasks or work flow. Then it becomes relatively easy to translate clinical work flow into state machine logic. If we think the application as a state machine, the completion of each task will cause the application to enter a new state, and there will be a set of states to indicate the status of the application. The user interaction with the GUI can be translated into inputs to the state machine.

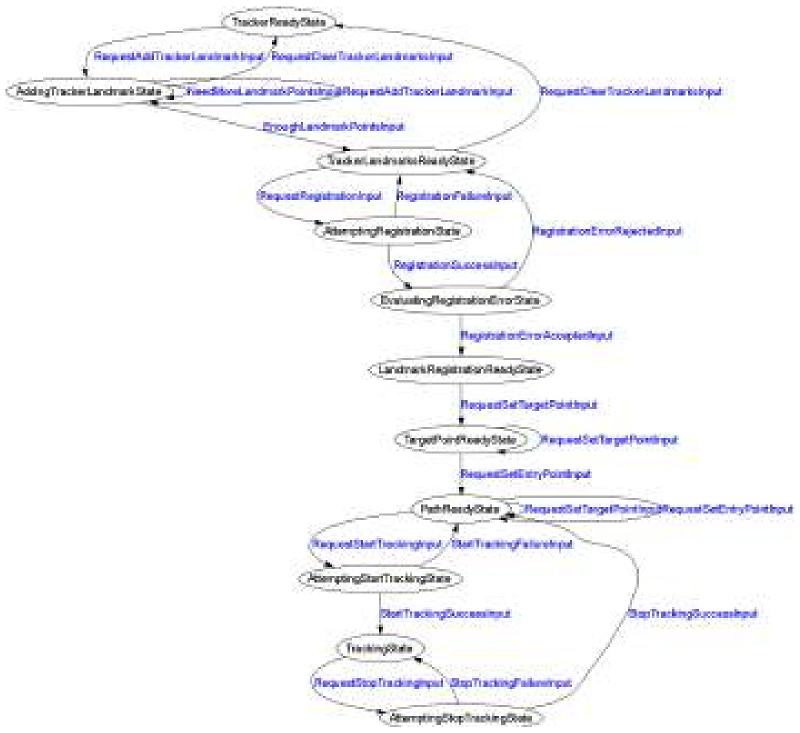

TrackerReadyState
RequestAddTrackerLandmarkInput
RequestClearTrackerLandmarksInput
AddingTrackerLandmarkState
NeedMoreLandmarkPointsInput
RequestAddTrackerLandmarkInput
RequestClearTrackerLandmarksInput
EnoughLandmarkPointsInput
TrackerLandmarksReadyState
RequestRegistrationInput
RegistrationFailureInput
AttemptingRegistrationState
RegistrationErrorRejectedInput
RegistrationSuccessInput
EvaluatingRegistrationErrorState
RegistrationErrorAcceptedInput
LandmarkRegistrationReadyState
RequestSetTargetPointInput
TargetPointReadyState
RequestSetTargetPointInput
RequestSetEntryPointInput
PathReadyState
RequestSetTargetPointInput
RequestSetEntryPointInput
RequestStartTrackingInput
StartTrackingFailureInput
AttemptingStartTrackingState
StartTrackingSuccessInput
StopTrackingSuccessInput
TrackingState
RequestStopTrackingInput
StopTrackingFailureInput
AttemptingStopTrackingState

Figure 21.2: State machine diagram (partial) for the needle biopsy application(Circle for state and arrow for transition and corresponding input).

For instance, when we click on the register patient button, this will generate a "RequestSetPatientNameInput" to the state machine. The state machine will take this input and change its current state from "InitialState" to "WaitingForPatientNameState", and the action is to pop up a window asking for input of the patient name. If the user inputs a valid name, then there will be a "PatientNameInput" which brings the state machine into "PatientNameReadyState", otherwise there will be a "PatientNameEmptyInput", which will return the state machine to the "InitialState". Thus, we can map the application into series of states and inputs and this higher level abstraction will help the developer design a clear work flow for the application. Figure 21.2 shows the state machine diagram for the needle biopsy application which was generated automatically when the state machine is constructed using the 'dot' tool from Graphviz.

21.2.3 Coding the state machine

This section shows how to code the state machine into the needle biopsy application. IGSTK has a number of convenient macros to facilitate the programming of the state machine. The details of these macros can be found in `Source/igstkMacro.h`. More information on the state machine design pattern and guidelines can be found on the IGSTK wiki page under "Development" section on "Design discussions" page.`http://public.kitware.com/IGSTKWIKI`

Once we have the higher level abstraction of the application and prototyped it in the state machine model, we need to take the following steps to program the state machine into the applications.

1. The first step is to use the state machine declaration macro in your class's header file. This macro defines types for state and input, creates a private member variable `m_StateMachine`, and two private member functions for exporting the state machine description into dot format for the state machine diagram visualization and LTSA (Labeled Transition Systems Analyzer) format for state machine animation and validation.

   ```
   igstkStateMachineMacro();
   ```

2. Then take the states and inputs mapped out during the prototyping stage and define them in the header file using the following macros. To enforce the naming conventions of the state machine, the declaration macros will append "State" or "Input" automatically after the variable name. For instance, the following two lines will define `m_InitialState` and `m_RequestLoadImageInput`.

   ```
   igstkDeclareStateMacro( Initial );
   igstkDeclareInputMacro( RequestLoadImage );
   ```

3. The next step is to construct the state machine in the constructor of the source file. First, we need to add all the states and inputs declared in the header into the state machine.

   ```
   igstkAddStateMacro( Initial );
   igstkAddInputMacro( RequestLoadImage );
   ```

4. The next step is a crucial step which creates the state machine transition table to control the logic and workflow of the application. This is done using the macro `igstkAddTransitionMacro( From_State, Received_Input, To_State, Action)`. This means when the state machine is in the `From_State` and receives the `Received_Input`, it will enter into the `To_State` and evoke the `ActionProcessing()` as an action for this transition. This macro requires the "ActionProcessing" method to be pre-defined in the class for the state machine to call. For example:

   ```
   igstkAddTransitionMacro( Initial, RequestSetPatientName,
                            WaitingForPatientName, SetPatientName );
   ```

   In this case, we need to have a SetPatientNameProcessing() method defined in the class for this code to compile.

5. After we have setup the transition table, the next step is to select an initial state, and flag the state machine to be ready to run. After the state machine is ready to run, we cannot change the state machine transition table in the code. This is designed this way to enhance the safety of the state machine and prevent accidentally changes to the state machine behavior in the code.

   ```
   igstkSetInitialStateMacro( Initial );
   m_StateMachine.SetReadyToRun();
   ```

6. Now the state machine is setup and ready to run. We can then export the state machine description in dot format and generate the graphical visualization as shown in Figure 21.2. This graph will help us to examine the workflow of the application and the state transition table.

   ```
   std::ofstream ofile;
   ofile.open("DemoApplicationStateMachineDiagram.dot");
   const bool skipLoops = false;
   this->ExportStateMachineDescription( ofile, skipLoops );
   ofile.close();
   ```

   This will output the state machine into a dot file when we execute the application. If you have the dot tool installed in your system, then you can run the following command, which will take the dot file and generate a png format picture named "SMDiagram.png" for the state machine.

   ```
   >dot -T png -o SMDiagram.png DemoApplicationStateMachineDiagram.dot
   ```

7. All the requests to a state machine should be translated into inputs and the state machine will response to those inputs depending on its current state. These actual actions should be protected methods and only called by the state machine directly. In the code, a click on the load image button will be translated to a `RequestLoadImageInput`, and then we call `ProcessInputs()` to let the state machine handle this request.

```
igstkPushInputMacro( RequestLoadImage );
m_StateMachine.ProcessInputs();
```

If the state machine is in the right state to load the image, a protected method associated with this transition (eg. `LoadImageProccessing()`) will be evoked by the state machine as defined in the transition table constructed in the constructor.

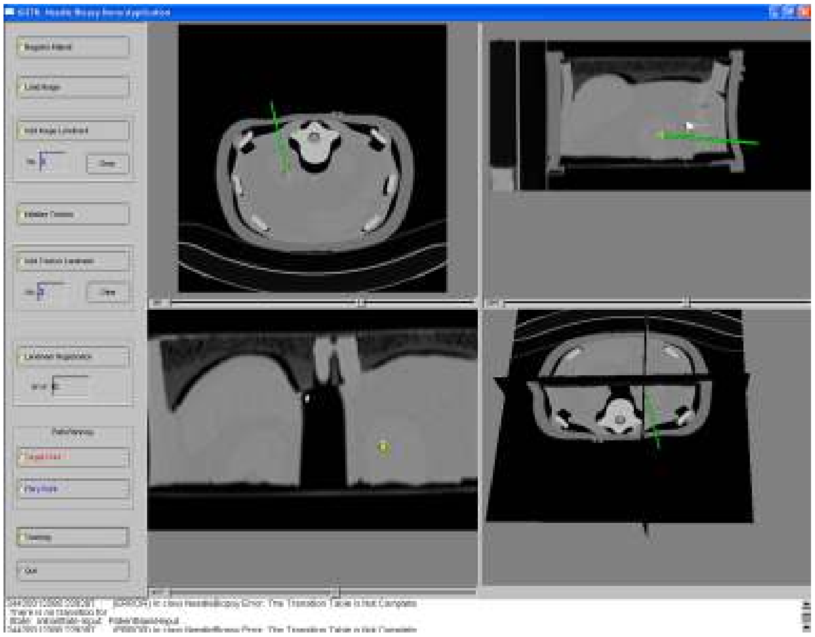
21.2.4 Should I use the state machine in my application?

From the computational theory point of view, all computers are state machines, and all computer programs are state machines regardless of whether the developers used the state machine programming pattern or not. Traditional programming approaches represent the states ambiguously by using a large number of variables and flags, which result in many conditional tests in the code (if-then-else or switchcase statements in C/C++). Programmers could neglect to consider all possible paths in the code while struggling with if-else conditional tests and flag checks. These practices may result in unpredictable behavior and limit safety in the design of the underlying applications. Since predictability is critical for mission critical applications running in the surgery room, this approach is not suitable for our purpose.

In comparison to traditional approaches, state machines will reduce the number of paths in the code, save the developers from convoluted conditional tests, and encourage them to focus on higher level design. From the above example, we can see that the state machine is easy to program and manage under the IGSTK framework. We encourage developers to design and code the state machine of their application first, and then generate the state machine diagram as shown in Figure 21.2. They can go through the diagram, examine and verify their design of the work flow. If they want to add or change a path of the application, it is just a matter of adding or deleting a transition table entry. This eliminates the level of difficulty required for going through the code and struggling with if-then-else logic. This will largely facilitate the application prototyping, and the implementation code can be plugged into the skeleton program later. These techniques should result in clearer designs and safer applications.

21.3 Result

Figure 21.3 shows the user interface of the needle biopsy application written in FLTK. The left side is the control panel, consists of a set of buttons corresponding to the series of tasks performed during the procedure. These buttons' callbacks should call the public request methods of the application, which will be translated into state machine inputs. The state machine will then take proper action according to its own state. For example, when the patient information is not set, the 'Load Image' button won't respond to the user click. There is no need for conditional checks or disabling of buttons here as these actions are already in the state machine transition table. On the right hand side, there are four standardized views, axial, sagittal, coronal, and 3D view. Here we loaded an abdominal phantom CT images. The green cylinder represents



```
[illegible]    [ERROR] In class NeedleBiopsy Error: The Transition Table is Not Complete
 There is no transition for
 State: [illegible]  Input: [illegible]
[illegible]    [ERROR] In class NeedleBiopsy Error: The Transition Table is Not Complete
```

Figure 21.3: User interface for needle biopsy program.

the needle being tracked by the tracker. The viewer will automatically reslice the images as the needle tip is moving in the anatomy.

CHAPTER

TWENTYTWO

Ultrasound Guided Radio-Frequency Ablation

22.1 Introduction

Liver lesions suitable to be treated using radio-frequency ablation (RFA) are often clearly visible under CT/MR but not using Ultrasound (US) imaging. Therefore most of the RFA surgeries of the liver are performed under CT. Other alternatives consist of waiting until the lesions enlarge and show up under US or perform an open surgery. An ideal visualization system would show tumor contours under US to the surgeons. A typical workflow of an RFA surgery is described in figure 22.1.

This application registers a pre-operative model of the tumors with a 2D US slice in pseudo real-time. The system can be divided into three parts: a) tracking devices, b) registration algorithm and c) display. First, the 2D ultrasound probe is tracked using an optical tracker (Polaris from NDI). Second, an image-to-image registration algorithm registers each 2D slice with a the pre-operative CT. One can notice that this registration step should be performed as quickly as possible. Third and last, a display presents the actual 2D US slice with tumor outlines to the surgeon.

Next the different components of the application, the tracking systems and the registration algorithm are presented.

22.2 Running the Application

This application can be found in the Examples/UltrasoundGuidedRFA. In order to build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you need to make sure that IGSTK_USE_FLTK is turned ON when running CMake.

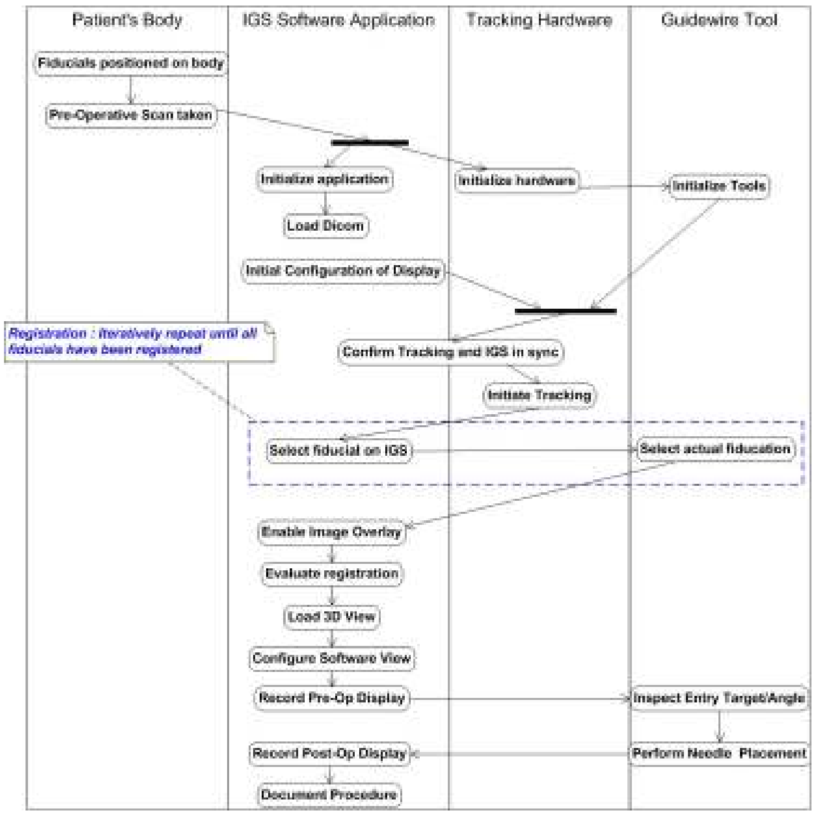


Figure 22.1: Typical RFA Ablation Surgery Workflow.

22.3 Implementation

22.3.1 Tracker

Using a tracker in IGSTK is quite easy. First we create a tracker object using smart pointers.

```
typedef igstk::PolarisTracker    TrackerType;
TrackerType::Pointer m_Tracker = TrackerType::New();
```

Then, we set a ToolCalibrationTransform which defines the relationship between the tracking device and the origin of the tool. In our case, the optical sensor is attached to arm the probe, therefore the calibration transform is defined as a rigid transform from the sensor position to the tip of the probe. This transform can be computed from a calibration experiment or using some heuristics.

```
m_Tracker->SetToolCalibrationTransform( TRACKER_TOOL_PORT, 0,
                                                                                                    ToolC
```

Next, we need to define the relationship between the tracking system origin and the actual patient position in the OR. Most of the surgical applications define the OR as the world coordinate origin. This PatientTransform is often assessed via calibration.

```
m_Tracker->SetPatientTransform(PatientTransform);
```

The IGSTK spatial object to be tracked is attached to the tracker using AttachObjectToTrackerTool. Therefore when the position and orientation of the tracking device is modified, the updated position of the spatial object is automatically computed. One can notice that this step involves the concatenation with the ToolCalibrationTransform and the PatientTrasnform.

```
m_Tracker->AttachObjectToTrackerTool( TRACKER_TOOL_PORT,
                                                                                            TRACKER_T
                                                                                        m_UltrasoundP
```

Our tracker is ready to be used, we start the tracking by first opening the serial communication port using `Open()`, then we initialize the tracker and start the tracking.

```
m_Tracker->Open();
m_Tracker->Initialize();
m_Tracker->StartTracking();
```

To stop the tracking device we just use the `StopTracking()` function.

```
m_Tracker->StopTracking();
```

One can notice that switching from one tracker to another can be done by modifying a single line of the code above, e.g. the tracker type definition.

22.3.2 Registration

Using the tracking information of the ultrasound probe the location of the US slice is roughly defined in the OR. To define a proper alignment of the US slice and the pre-operative CT volume registration is needed. The registration algorithm is an image-to-image technique based on the cross-correlation.

IGSTK makes use of registration algorithms already implemented in the Insight Segmentation and Registration Toolkit [**?**]. However, IGSTK propose algorithm already tuned from specific modalities and organs. Using class hierarchies, programmers can still makes use of higher level registration technique. For instance, the igstkMR3DImageToUS3DImageRegistration class performs registration of any 3D MR to 3D US. The parameters of the registration are already tuned to support most of the MR-to-US registrations but some other tuning might be required for different organs.

22.3.3 Display

IGSTK propose several visualization techniques based on the Visualization Toolkit [**?**]. Basically for a given IGSTK spatial object, several representation objects can be created and added the display as shown in figure **??**.

Image volumes such as CT or MR datasets can be renderered as textured oblique slices or can be volume renderered. Mesh objects such as segmented tumors can be renderered in 3D as triangle surfaces and in 2D as contours. The update of the display is done in real-time when the number of objects in the scene is not excessive and does not require extensive computation, i.e volume rendering of large datasets.

22.3.4 Implementation

Here we show an example on how to read and display a vasculature extracted from CT.

First we create a vascular network reader using smart pointers.

```
typedef igstk::VascularNetworkReader  VascularNetworkReaderType;
VascularNetworkReaderType::Pointer m_VascularNetworkReader;
m_VascularNetworkReader = VascularNetworkReaderType::New();
```

Then we set the vasculature filename and we ask the reader to read the file using the RequestReadObject() function.

```
m_VascularNetworkReader->RequestSetFileName(VasculatureFilename);
m_VascularNetworkReader->RequestReadObject();
```

In order to get a spatial object from a reader, IGSTK uses the *event/observer* mechanism. We declare a specific observer to get the vasculature from the reader and we ask the reader to return the object.

```
VascularNetworkObserver::Pointer vascularNetworkObserver
                                        = VascularNetworkObserver::New();
m_VascularNetworkReader->AddObserver(
                        VascularNetworkReader::VascularNetworkModifiedEvent(),
                        vascularNetworkObserver);
m_VascularNetworkReader->RequestGetVascularNetwork();
```

Next, we instantiate an object representation for the VascularNetwork object.

```
 typedef igstk::VascularNetworkObjectRepresentation
                                       VascularNetworkRepresentationType;
 VascularNetworkRepresentationType::Pointer m_VascularNetworkRepresentation =
                                                                    VascularN
```

Then we set the spatial object to the object representation. Internally the object representation creates a suitable visualization of the object from its internal geometry.

```
m_VascularNetworkRepresentation->RequestSetVascularNetworkObject(
                              vascularNetworkObserver->GetVascularNetwork() );
```

Finally, we add the object to the display.

```
this->Display3D->RequestAddObject( m_VascularNetworkRepresentation );
```

22.4 Conclusion

CHAPTER

TWENTYTHREE

Robot Assisted Needle Placement

Even with the image guidance introduced in the previous needle biopsy application (Chapter 21), the physician might be limited by the view of the exact position of any surgical instruments in the interventional field, and they might need to spend a fair amount of time to align the instruments with the planned path. In this chapter we present an image-guided platform for precision placement of surgical instruments based upon a small four degree-of-freedom robot shown in Figure 23.1(B-RobII; ARC Seibersdorf Research GmbH, Vienna, Austria). This platform includes a custom needle guide with an integrated spiral fiducial pattern as the robot's end-effector and uses pre-operative computed tomography (CT) to register the robot to the patient directly before the intervention. The robot can then automatically align the instrument guide to a physician-selected path for percutaneous access. The path is chosen by the physician before the intervention using an established graphical user interface built using open-source toolkits such as the Image-Guided Surgery Toolkit (IGSTK). Potential abdominal targets include the liver, kidney, prostate, and spine. This system is aimed to increase the accuracy and speed of the biopsy procedure by incorporating robot. Figure 23.2 shows the setup of the whole system.

23.1 Running the Application

This application can be found in the `Examples/DeckOfCardRobot`. In order to build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you need to make sure that IGSTK_USE_FLTK is turned ON when running CMake.

As shown in Figure 23.3, the workflow of this application is (Suppose we are having a lung biopsy procedure):

1. Place the phantom on the CT table and mount the robot to the CT gantry.
2. Position the robot needle holder close to the region of interest.
3. Scan the phantom together with the robot.

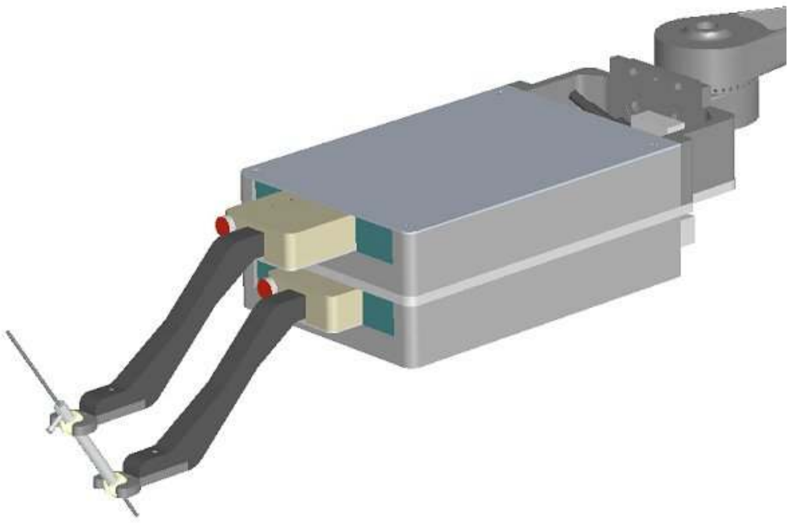


Figure 23.1: B-RobII four degree-of-freedom precision placement modules for needle positioning and orientation.

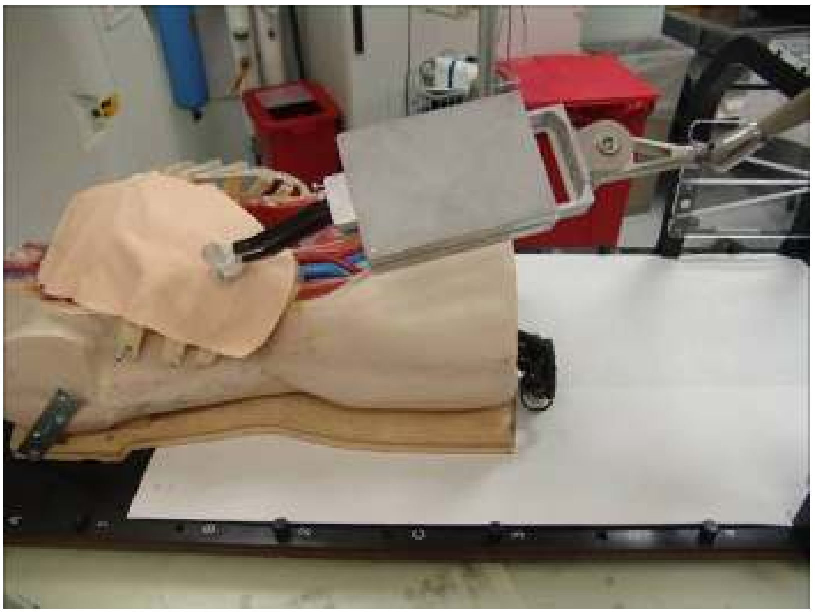


Figure 23.2: Robot assisted needle placement phantom study setup.

4. Load CT images into the robot control software.
5. Using the control software, segment out the fiducials in the CT image and perform the paired-point registration.
6. In the display window, plan the needle insertion path.
7. If the planned path is within the robot's working range, then command the robot to align the needle to the planned path. Otherwise, go back to step 2 and reposition the robot closer to the biopsy entry point.
8. Advance the needle by hand. The depth of insertion will also be calculated by the system and this depth can be judged by observing depth graduations on the needle itself.

23.2 Implementation

1. INTRODUCTION 1.1. Image guided needle placement Needle and needle tools are widely used in the clinical environment, especially in minimum invasive procedures, for both diagnostic and treatment purposes. One of the most commonly practiced procedures is needle biopsy, in which clinicians deploy needles into patient body to sample a small amount of tissue for laboratory analysis. This procedure is mostly being used for lung, breast, liver, and prostate for tumor diagnosis and cancer staging. In other procedures such as radio frequency ablation, surgeon insert ablation needle into the center of tumor and use the radio frequency energy to 'cook' the tumor. Both procedures are minimum invasive, but they also require a great amount of experience to accurately targeting the tumor to achieve the best result. By overlaying real-time location of tracked surgical tool on top of pre or intra-operative images, image guided technology can provide insight into the patient anatomy, thus increase the accuracy of minimum invasive procedures. 1.2. Deck of card robot The deck of cards robot is designed and manufactured by ARC Seibersdorf Research GmbH, Austria. (Figure 1 left). The robot has two joints (upper box and lower box, which can move parallel to each other) and 4 degree of freedoms (+19mm in translation and + 30o in rotation). Its unique shape gave it the name deck of card robot. Figure 1 right shows the system setup. The robot is mounted on the CT table after patient is in place. Robot arm is adjusted to position the needle holder close to the biopsy area. A CT scan is then acquired and loaded into robot assisted needle biopsy application. Surgeon can go through the image slices, identify tumors, and plan an optimal biopsy path by setting proper target and entry points to avoid important and vulnerable organs and tissues. The robot will then move the needle holder and align it with the planed path. Surgeon can advance the needle manually to hit the target. The deck of card robot can be operated remotely by multiple clients through TCP/IP communication. Client application should first connect to the server application as an active client before it can command the robot.

Figure 1. Deck of card robot (left) and robot assisted needle biopsy system setup (right) 1.3. Image-Guided Surgery Toolkit (IGSTK) IGSTK is an open source toolkit designed for development of image guided surgical applications [1, 2]. IGSTK is developed on top of three other open source toolkits i.e. ITK (segmentation and registration), VTK (visualization) and FLTK

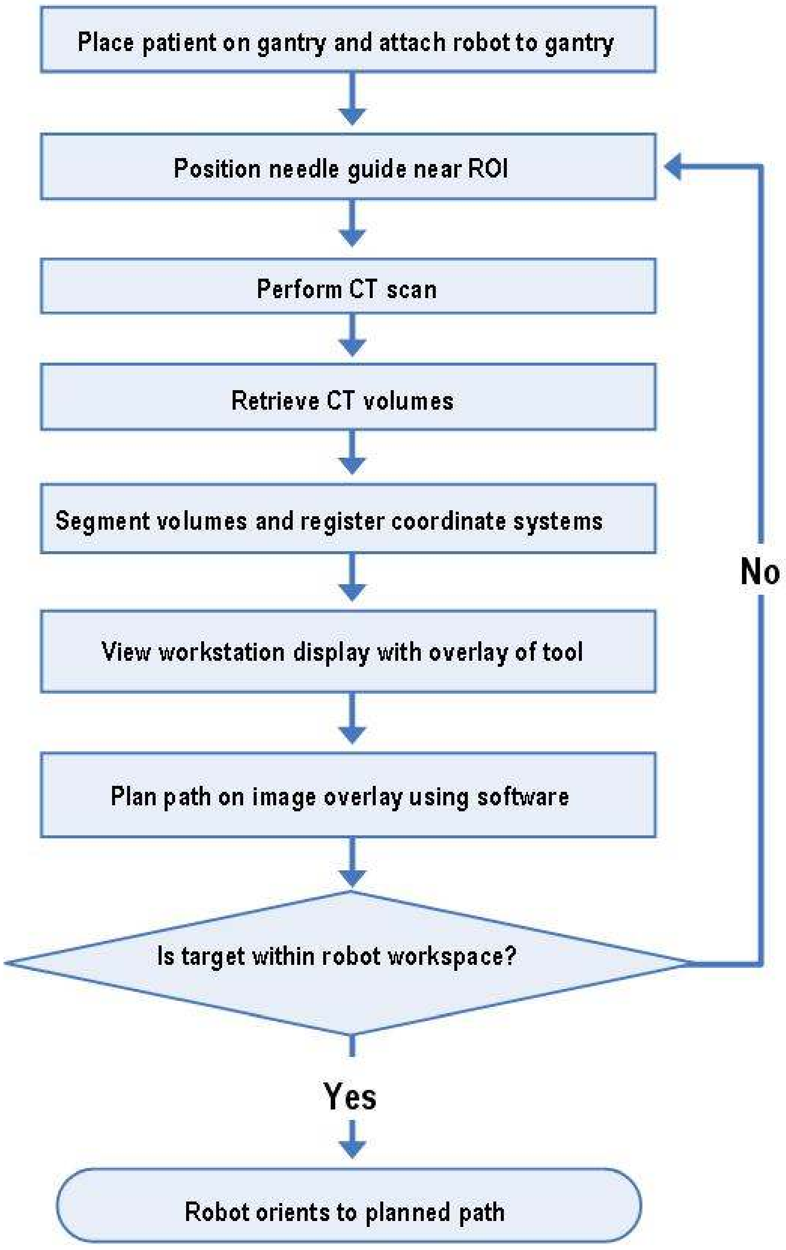

Place patient on gantry and attach robot to gantry
Position needle guide near ROI
Perform CT scan
Retrieve CT volumes
Segment volumes and register coordinate systems
View workstation display with overlay of tool
Plan path on image overlay using software
Is target within robot workspace?
No
Yes
Robot orients to planned path

Figure 23.3: Clinical Workflow for Robot Assisted Needle Placement.

for graphical user interface. IGSTK contains basic components needed in image guided surgery applications such as view classes for displaying and presenting results to the clinician, spatial object and spatial object representation classes for modeling and displaying physical object including images and anatomical structures, and tracker classes to handle and communicate with tracked surgical tools. 2. METHOD The critical step in robot guided needle placement application is to determine the transformation parameters between the robot and patient/image coordinate system. This registration procedure is commonly performed using fiducial based (point-based) registration technique. In this procedure, surgeons examine the image slice by slice and identify fiducial markers and establish the pairing to the physical fiducial markers. This manual tagging and matching procedure is time consuming and is prone to human errors. Hence, we developed automatic fiducial marker detecting and matching algorithm. We embedded 1mm diameter metal fiducial markers (19 in total) onto the surface of the cylindrical needle holder (Figure 2 left). Figure 2 right shows 3D reconstructed image of the fiducial markers from a CT scan. The markers follow a spiral pattern. The positions of each fiducial marker with respect to the robot space origin are known and are referred as the model points later in this paper.

Figure 2. Needle holder with embedded fiducial markers (left), and reconstructed image showing the fiducial markers (right). 2.1. Fiducial point detection Metal fiducial markers have very high absorption rate in CT images. Thus we can threshold the image, extract high intensity objects, and then calculate the centroid of each object as fiducial point position. In the case of the existence of other metal objects in the image, we used maximum and minimum size criteria to filter out non fiducial points such as needle, metal part of the robot, and metal implantation in patient. Even if we use the size restriction, there are still some small objects in the segmentation results can not be separate from the true fiducial points. The goal for fiducial clustering is to filter out those false positive segmentation results. Given the high density distribution of fiducial markers in the end effecter (19 fiducials in total), and no other metal object except fiducial points are detected within or around plastic needle holder (needle can be filter out by size criteria), we can conclude that the remaining false positives are all outliers. We can then calculate the distance map of the segmented point set, and iteratively delete point with the largest average distance to the rest of the points, until the maximum distance between point pair is less or equal than the maximum scale of the model. This is similar to the max-cut clustering algorithm. More sophisticated algorithm can be used to cluster the similarity map between segmented points and model points. Using the former two steps won't guarantee to segment out exact 19 fiducial points which is required by the next step landmark based registration. If a high threshold value is used, then we might miss detect some of the fiducial markers, if a low threshold value is used, then we might end up detecting too much false positive points, which does not the meet the assumption of the clustering algorithm that no points other than fiducial markers are detected within or around the needle holder area. Here we used two stage segmentation algorithms (Figure 3). First we use a high threshold value (3000) to segment the whole volume and followed by the clustering algorithm to extract the high confidence fiducial markers, this stage guarantee all the result points are true positive. We then use these points to extract the volume of interest which only contains the needle holder, and perform the segmentation again with a lower threshold (2000) followed by the clustering. This will guarantee to segment all the fiducial points. Because it's applied to a sub volume contains only the needle holder, it's much

faster, and won't generate false positive segmentation results.

Figure 3. Flow chart for the two stage segmentation algorithm 2.2. Fiducial point matching and registration After segmenting all the fiducial points in the image, we can then sort the points by projecting them on to its center axis, the axis of cylinder shaped needle holder. This axis can be approximated by extracting the principle component axis of the 3D points set. One design flaw of this particular end effecter is that it is symmetric with respect to the center plane perpendicular to end effecter's axis, which makes it hard for computer program to identify what's the right order of the point sequence. In this program we use the assumption that the point with largest Y coordinate (the lowest elevation, because the robot can not flip upside down, so the end effecter is always pointing downwards) corresponding to the smallest Z axis point in the model. A better solution to this problem is to manually take out one of the fiducial point on either upper or lower side of end effecter and make it asymmetric, and try the landmark based registration with two different orders of the points, and take the registration result with the smaller RMS error. 3. IMPLEMENTATION The fiducial segmentation and matching algorithm was implemented using ITK classes. The algorithm takes an ITK image, a threshold, and maximum and minimum sizes as input and generates a segmented points list. The major classes used are [3]: itk::BinaryThresholdImageFilter itk::ConnectedComponentImageFilter itk::RelabelComponentImageFilter Fiducial clustering filter takes a list of sample points and list of the model points as input and returns the clustered points list. The communication between the application and robot server is through TCP/IP protocol. For this purpose, socket communication component of IGSTK was used. Other modules of this application, user interface, visualization, registration and path planning are implemented using view, registration components of IGSTK. 4. RESULTS 4.1. Application GUI Figure 4 left shows the user interface of the application with control panel on the left, 3 standard 2D slice views and a 3D volume rendering on the right. The yellow cylinder is the needle holder, the purple square indicates robot's working region, a path is being planed to target the tumor while avoiding the ribs, and it is showing the robot being aligned with the planned path. Figure 4 right shows the phantom study setup. After registering the robot, the application can command the robot to align with the planned path, and the robot is able to hit the pre-attached skin fiducial markers on the planned path.

Figure 4. Application user interface showing the registered robot (left) and phantom study (right) 4.2. Validation study To validate the whole system especially the robot registration and communication part, we acquired three groups of data, each group contains one home position and three other know robot positions. That's 12 scans in total. The image is reconstructed with 0.637x0.637x1.00mm resolution. It takes about 30 seconds to run the algorithm on a moderate PC with an image size of 512x512x105. The automated algorithm can successfully register all 12 data sets with an accuracy of 0.298+0.018mm RMS errors. 5. DISCUSSION and CONCLUSION The results showed that the registration method developed here is robust. We are currently investigating to use more relaxed registration methods such as iterative closest points and model-to-image registration which do not require the exact same number of segmented fiducial points as in the model and the points pairing, and compare their success rate and accuracy to the this landmark based method. Furthermore, the algorithm doesn't take respiratory motion effect into consideration. In lung and liver biopsy, respiratory motion can introduce as much as 5 cm errors into the system. Motion compensation techniques must be used to reduce

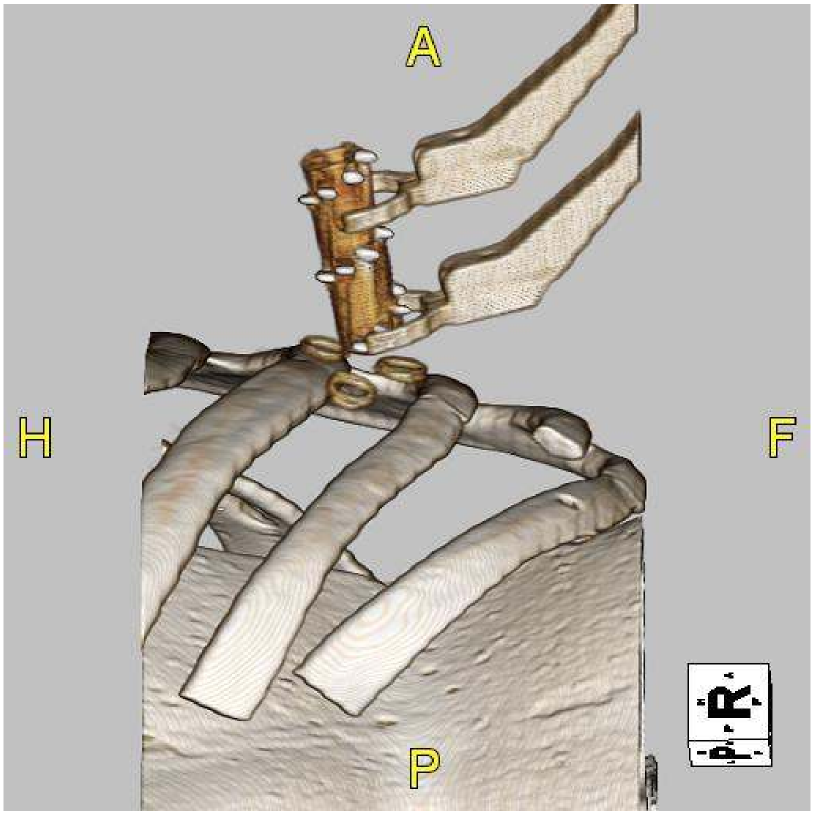
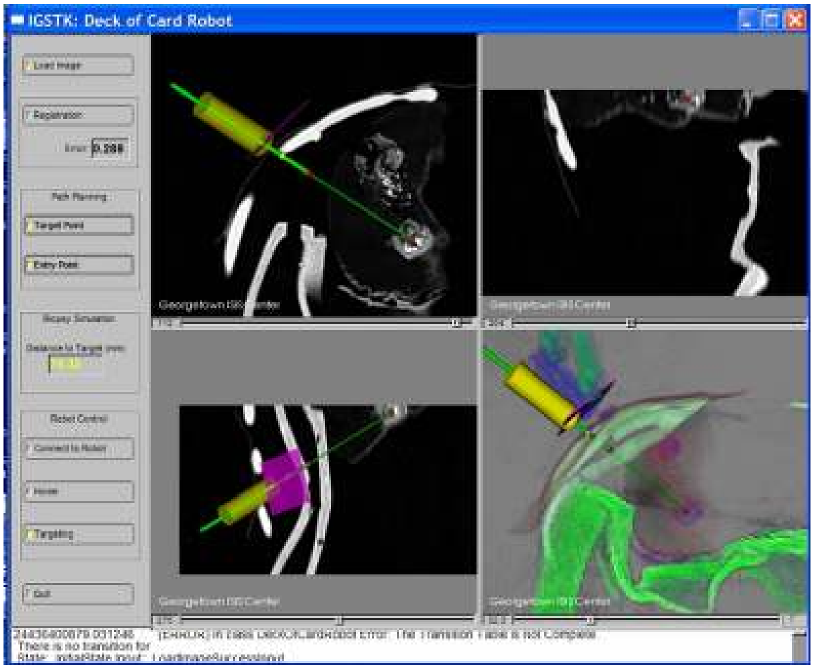


Figure 23.4: 3D rendering of CT scan showing spiral fiducial pattern for registration of robot to CT images.

the motion error when using the robot assisted needle biopsy system. 6. DISCLAIMER and ACKNOWLEDGEMENT This application is being developed as an example application with the release of IGSTK open source software toolkit, and it's not suitable for clinical use. The author would like to thank for help from other IGSTK developers while developing the system, there are Luis Ibanez, Julien Jomier, Stephen Aylward, Rick Avila, David Gobbi, Brain Blake, and Kevin Gary. This research is supported by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) at the National Institutes of Health (NIH) under grant R42EB000374 and by U.S. Army grant W81XWH-04-1-0078

23.3 Result



```
24436400079.031246    [ERROR] in class DeckOfCardRobot Error: The Transition Table is not Complete
 There is no transition for
 State: InitialState input: LoadImageSuccessInput
```

Figure 23.5: User interface for the robot application.

Part V

Appendices

APPENDIX

A

IGSTK Style Guide

A.1 Purpose

The following document is a description of the accepted coding style for the Image Guided Software Toolkit (IGSTK). We have chosen to follow the Style Guide for NLM Insight Registration and Segmentation Toolkit (ITK) with minor modifications where needed.

A.2 Document Overview

This document is organized into the following sections.

- XXX

This style guide is an evolving document. Any changes to style guide (addition, modification, or deletion of rules) you wish to propose has to be discussed and agreed upon at the IGstk Developers meeting or mailing list.

A.3 Implementation Framework

A.3.1 Implementation Language

The core implementation language is C++. C++ was chosen for its flexibility, performance, and familiarity to development members. IGstk uses the full spectrum of C++ features including const and volatile correctness, and namespaces. Operator overloading is done in moderation, and only for very basic object types.

A.3.2 Generic Programming

The use of templates in IGSTK is discouraged, and it is only accepted for well justified cases.

A.3.3 Generic Programming

Use of the STL is encouraged. STL is typically used by a class, rather than serving as a base class for derivation of IGstk classes. We encourage the use of STL in the internal code of IGSTK classes, but avoid it at the API level. For example, if an IGSTK class needs a list of points as argument to a method, we should not define the method's signature as `SetPoints(std::vector<points>)`, but rather define and IGSTK ListOfPoints class, and define the method `SetPoints(const ListOfPoints &list)`. This allows enforcing correctness on the types passed to methods and reduces the risks of unwanted or inadverted castings that may result in run-time errors.

A.3.4 Portability

IGstk is designed to compile on a set of target operating system/compiler combinations. These combinations include:

- XXX

Since some of these compilers do not support all C++ features, some important C++ features (such as partial specialization) may not be used because of limitations in compilers (e.g., MSVC 6.0).

Whatever is the list of compilers we select to support, we must make sure that we setup Nightly builds for all of them. The de-facto definition of a supported compiler is *compiler that is submitting a green nightly build to the Dashboard*.

A.3.5 CMake Configuration Environment

The IGstk configuration environment is CMake. CMake is an open-source, advanced cross-platform build system that enables developers to write simple ed native build tools for a particular operating system/compiler combinations. (www.cmake.org).

A.3.6 Doxygen Documentation System

The Doxygen open-source system is used to generate on-line documentation. Doxygen requires the embedding of simple comments in the code which is in turn extracted and formatted into documentation. (http://www.stack.nl/ dimitri/doxygen/). (It is important for developers to get familiar with Doxygen tokens. The Doxygen manual is quite detailed and offers a large number of possibilities. For example, relating one class to another, including equations, including images, including links to URLs.

A.3.7 vnl Math Library

IGstk would use vnl Math Library for math related functions. These library are included in the ITK source code. (http://www.robots.ox.ac.uk/ vxl/). We should try to avoid using VNL inside IGSTK classes, and rather try to use it through ITK. Definitely never expose vnl at the IGSTK API level (in the same way that we should avoid exposing ITK or VTK).

A.3.8 Reference Counting and SmartPointers

IGSTK has adopted reference counting via *smart pointers* to manage object references. Smart pointers automatically increment and decrement an instance's reference count, deleting the object when the count goes to zero. The use of SmartPointers is limited to classes that have a significant memory footprint, such as high level IGSTK components.

A.3.9 CVS Environment

CVS would be used for the Version Control, software updates and downloads.

A.3.10 Dart Dashboard Testing Environment

IGstk intends to have testing for 100which uses every line of the IGstk code. Test code should be written preferably before the main code is written, or along with the main code. The Dart/Dashboard would be used as the testing environment. The notion of 100coverage. It goes beyond 100been executed as part of a test. In the case of IGSTK we must test all possible combinations of function calls in order to ensure the robustness of the toolkit against inappropriate usage. The use of State Machines is fundamental in order to achieve this goal. The stringent testing must be enforced if we expect IGSTK ever to be used in a surgery room.

A.4 Copyright

IGSTK has adopted a standard copyright. This copyright should be placed at the head of every source code file. The current header added to every IGSTK file reads as follows:

```
/*=================================================================

  Program:   Image Guided Surgery Software Toolkit
  Module:    $RCSfile: IGSTKStyleGuide.tex,v $
  Language:  C++
  Date:      $Date: 2006/10/26 19:19:11 $
  Version:   $Revision: 1.2 $

  Copyright (c) ISIS Georgetown University. All rights reserved.
  See IGSTKCopyright.txt or http://www.igstk.org/HTML/Copyright.htm
  for details.

  This software is distributed WITHOUT ANY WARRANTY; without even
  the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
  PURPOSE.  See the above copyright notices for more information.

=================================================================*/
```

Note the use of embedded CVS commands at the top of the header, such as $RCSfile : IGSTKStyleGuide.tex,v$, $Date : 2006/10/2619 : 19 : 11$, $Revision : 1.2$. These should be used in each source file under CVS control.

The copyright read as follows:

```
/*===========================================================================

Copyright (c) 1999-2003 Imaging Science and Information Systems Center
                        Georgetown University.
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 Imaging Science and Information Systems Center, nor the names
   of any of the developers, 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.5 File Organization

Classes are created and (usually) organized into a single class per file set. A file set consists of .h header file, .cxx implementation file, and/or a .txx templated implementation file. Helper classes may also be defined in the file set, typically these are not visible to the system at large, or placed into a special namespace. Source files must be placed in the correct directory for logical consistency with the rest of the system, and to avoid cyclic dependencies. Currently the IGstk source directory looks as follows:

```
IGSTK
   Source
     {all *.h, *.txx and *.cxx code of the basic code}
   Examples
     {all *.h, *.txx and *.cxx code of the example (application)  code}
   Testing
     {all *.h, *.txx and *.cxx code of the test  code}
   Data
     Baseline
       {all sample result data for comparing}
     Input
       { all input data for the test code }
```

A.6 Namespaces

All classes should be placed in the *igstk* namespace. Sub-namespaces may be used for helper IGstk classes. Additional sub-namespaces may be designed to support special functionality.

A.7 Naming Conventions

In general, names are constructed by using case change to indicate separate words, as in TimeStamp (versus Time Stamp). Underscores are not used. Variable names are chosen carefully with the intention to convey the meaning behind the code. Names are generally spelled out; use of abbreviations is discouraged. (Abbreviation are allowable when in common use, and should be in uppercase as in RGB). While this does result in long names, it self-documents the code. Depending on whether the name is a class, file, variable, or other name, variations on this theme result as explained in the following subsections.

A.7.1 Naming Classes

Classes are named beginning with a capital letter. Classes are placed in the appropriate namespace, typically igstk. Classes are named according to the following general rule:

```
class name = <process><input><concept>
```

In this formula, the name of the process (possibly with an associated adjective or adverb) comes first, followed by an input type (e.g. modality of the images), and completed by a concept name. A concept is an informal classification describing what a class does. Here are some of the concepts defined in IGSTK:

- Reader - A component that read medical images from files.
- Tracker - A proxy for a hardware device.
- Viewer - An abstraction of a visualization window.

The naming of classes is an art form; please review existing names to catch the spirit of the naming convention. Example names include: DicomReader, TrackerInterface, DefaultImageTraits.

A.7.2 Naming Files

Files should have the same name as the class, with an igstk" prepended. Header files are named .h, while implementation files are named either .cxx or .txx, depending on whether they are implementations of templated classes. For example, the class igstk::DicomReader is declared and defined in the files igstkDicomReader.h and files igstkDicomReader.txx (because DicomReader is templated).

A.7.3 Naming Methods and Functions

Global functions and class methods, either static or class members, are named beginning with a capital letter. The biggest challenge when naming methods and functions is to be consistent with existing names.

When referring to class methods in code, an explicit *this* pointer should be used. The use of the explicit *this* pointer helps clarify exactly which method, and where it originates, is being invoked. Similarly the global namespace (::) should be used when referring to a global function.

A.7.4 Naming Class Data Members

Member variables are named beginning with a capital letter. All member variables are declared as private, with Get/Set methods. When refereeing to a member variable, an explicit *this* pointer should be used. Derived classes must us the combination of *this* and Get/Set methods.

A.7.5 Naming Local Variables

Local variables begin in lowercase. There is more flexibility in the naming of local variables. Please remember that others will study, maintain, fix, and extend your code. Explanatory variable names and comments will go a long way towards helping other developers. Variable names are considered to be part of the documentation of the code. Avoid short names that do not describe the role of the variable.

A.7.6 Naming Template Parameters

Template parameters follow the usual rules with naming except that they should start with either the capital letter T or V. Type parameters begin with the letter T while value template parameters begin with the letter V.

A.7.7 Naming Typedefs

The use of Typedefs is encouraged. They significantly improve the readability of code, and facilitate the declaration of complex syntactic combinations. Unfortunately, creation of typedefs is tantamount to creating another programming language. Hence typedefs must be used in a consistent fashion.

The general rule for typedef names is that they end in the word Type. For example

```
typedef TPixel PixelType;
```

However, there are certain exceptions to this rule in order to highlight important concepts used in IGSTK. Following is the list of these exceptional cases:

- Self as in: `typedef Image Self;` *All classes should define this typedef.*
- Superclass as in: `typedef ImageBase<VImageDimension> Superclass;` *All classes should define the Superclass typedef.*

- Pointer as in a smart pointer to an object as in: `typedef SmartPointer<Self> Pointer;` and ConstPointer as in: `typedef SmartPointer< const Self > ConstPointer;`

For classes following concepts Container, Iterator and Identifier, the concept name is used in preference to Type at the end of a typedef as appropriate. For example

```
 typedef typename ImageTraits::PixelContainer PixelContainer;
```

Here Container is a concept used in place of Type.

A.7.8 Using Underscores

Don't use them. The only exception is when defining class data member variables, preprocessor variables and macros (which are discouraged). In this case, underscores are allowed to separate words.

A.7.9 Preprocessor Directives

Please avoid using preprocessor directives, except to support minor differences in compilers or operating systems. If a class makes extensive use of preprocessor directives, it is a candidate for separation into its own class.

A.8 Const Correctness

Const correctness is important. Please use it as appropriate to your class or method. Const correctness is fundamental for maintaining the integrity of IGSTK classes. A safe approach is to start considering everything as *const* and making classes and methods *non-const* only when a justification exists. Const verification is done by the compiler and prevents inappropriate and unsafe use of the classes and methods. Note that VTK does not enforce const correctness and ITK still have some flexible spots. IGSTK must cover these eventual const-correctness failures and enforce complete const-correctness verification.

A.9 Code Layout and Indentation

We chose to follow the accepted ITK code layout rules and indentation style, and we are reproducing them below. After reading this section, you may wish to visit many of the source files found in ITK. This will help crystalize the rules described here.

A.9.1 General Layout

Each line of code should take no more than 79 characters. Break the code across multiple lines as necessary. Use lots of whitespace to separate logical blocks of code, intermixed with comments. To a large extent the structure of code directly expresses its implementation.

The appropriate indentation level is two spaces for each level of indentation. DO NOT USE TABS. Set up your editor to insert spaces. Using tabs may look good in your editor but will wreak havoc in others.

The declaration of variables within classes, methods, and functions should be one declaration per line.

```
 int i;
int j;
char* stringname;
```

A.9.2 Class Layout

Classes are defined using the following guidelines.

- Begin with #ifndef guards and finish with #endif guard.
- Follow with the necessary includes. Include only what is necessary to avoid dependency problems.
- Place the class in the correct namespace.
- Public methods come first.
- Protected methods follow.
- Private members come last.
- Public data members are forbidden.
- Templated classes require a special preprocessor directive to control the manual instantiation of templates. (See the example below and look for ITK MANUAL INSTANTIATION.)

The class layout looks something like this:

```
#include ‘‘igstkDicomReaderBase.h’’
#include ‘‘itkPixelTraits.h’’
#include ‘‘itkDefaultImageTraits.h’’
#include ‘‘itkDefaultDataAccessor.h’’
```

```
namespace igstk
{
template <class TPixel, unsigned int VImageDimension=2,
                 class TImageTraits=DefaultImageTraits< TPixel, VImageDimension > >
class IGSTK_EXPORT DicomReader : public DicomReaderBase<VImageDimension>
{
public:
....stuff...
protected:
....stuff...
private:
....stuff...
};
}//end of namespace
#ifndef ITK_MANUAL_INSTANTIATION
#include ‘‘igstkDicomReader.txx’’
#endif
#endif //end include guard
```

A.9.3 Method Definition

Methods are defined across multiple lines. This is to accommodate the extremely long definitions possible when using templates. The starting and ending brace should be in column one. For example:

```
 template<class TPixel, unsigned int VImageDimension, class TImageTraits>
const double *
Image<TPixel, VImageDimension, TImageTraits>
::GetSpacing() const
{...}
```

The first line is the template declaration. The second line is the method return type. The third line is the class qualifier. And the fourth line in the example above is the name of the method.

A.9.4 Use of Braces

Braces must be used to delimit the scope of an *if*, *for* *while*, *switch*, or other control structure. Braces are placed on a line by themselves:

```
for (i=0; i<3; i++)
   {
    ...
   }
```

or when using an if:

```
if (condition)
    {
...
    }
else if ( other condition )
    {
...
    }
else
    {
....
    }
```

A.9.5 Use of Whitespace

Use spaces around arguments of functions and around operators. For example, instead of

```
 function(sum,operator,output);
```

write

```
 function(sum, operator, output);
```

A.10 Doxygen Documentation System

Doxygen is an open-source, powerful system for automatically generating documentation from source code. To use Doxygen effectively, the developer must insert comments, delimited in a special way, which Doxygen extracts to produce the documentation. We chose that every comment starts with /**, each subsequent line has an aligned *, and the comment block terminates with a */.

A.10.1 Documenting a Class

Classes should be documented using the class and brief Doxygen commands, followed by the detailed class description:

```
 /** Object
* Base class for most itk classes.
*
* Object is the second-highest level base class for most igstk objects.
* It extends the base object functionality of LightObject by
* implementing debug flags/methods and modification time tracking.
```

```
*/
```

A.10.2 Documenting a Method

Methods should be documented using the following comment block style as shown in the following example. Make sure you use correct English and complete, grammatically correct sentences.

```
 /** Access a pixel at a particular index location.
* This version can be an lvalue.  */
TPixel &operator[](const IndexType &index)
{return this->GetPixel(index); }
```

The documentation is only written in the header files (.h). Additional comments may be added to the .cxx files, but they will not be collected by Doxygen. Note that documentation must be maintained along with the code. Whenever a method is modified, its documentation must be checked in order to ensure that it is still applicable to the new modified method.

A.11 Using Standard Macros

There are several macros defined for IGSTK in the file igstkMacros.h. These macros help perform several important operations, that if not done correctly can cause serious, hard to debug problems in the system. These operations are:

- Management of Object modified time.
- Printing Debug information.
- Handling Reference counting.

Some of the more important object macros are the following.

igstkNewMacro(T) Creates the static class method New(void) that instantiates objects without using factories. The method returns a SmartPointer¡T¿ properly reference counted.

igstkSetMacro(name,type) Creates a method SetName() that takes argument type type.

igstkGetMacro(name,type) Creates a method GetName() that returns a non-const value of type type.

A.12 Exception Handling

C++ Exceptions will not be used since they make difficult to guarantee that the state of the classes is valid after recovering from an exceptions.

Error conditions will be modeled as error states in the state machines of every IGSTK component. Events and Observers will be used for notifying other classes whenever an error condition is encountered.

A.13 Documentation Style

The guidelines for producing supplemental documentation (other than the documentation produced by Doxygen) are as follows:

- The common denominator for documentation is either PDF or HTML. All documents in the system should be available in these formats, even if ther are mastered by another system.
- Presentations are acceptable in Microsoft PowerPoint format.
- Administrative and planning documents are acceptable in Microsoft Word format (either .doc or .rtf).
- Larger documents, such as the user's or developer's guides, are written in Microsoft Word.

A.14 Programming Practices

A.14.1 Choice of double or float

Use *double* instead of *float*; Most of the computation is done internally with double even if the variables are declared float. Use *float* when you are allocating a large number of them, such as a pixel type of an image.

A.14.2 Choice of signed or unsigned

Be careful for signed/unsigned mismatch warnings. In general, for for loops, you want to use *unsigned int*.

APPENDIX

B

State Machine Validation

```
Need 2-3 pages on state machine validation
```

BIBLIOGRAPHY

[1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Professional Computing Series. Addison-Wesley, 2001. 3.7.4

[2] A. Cheng. *RealTime Systems: Scheduling, Analysis, and Verification*. Wiley Interscience, 2002. 3.1, 3.7.4, 6.2

[3] B. P. Douglas. *RealTime Design Patterns: Robust Scalable Architecture for RealTime Systems*. Addison-Wesley Professional, 2002. 3.1, 3.7.4, 6.1, 6.2.7, 7.1

[4] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976. 5.2.2

[5] K. Fogel. *Open Source Development with CVS*. Corolis, 1999. 1.4.2, 2.1.4

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 3.7.4, 7.1

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languges, and Computation*. Addison Wesley, 2001. 6.2

[8] B. K. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4:629–642, April 1987. 14.1

[9] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, 2003. 5.2.6

[10] L. Kohn, J. Corrigan, and M.Donaldson, editors. *To Err is Human: Building a safer health system*. National Academy Press, 2001. 3.2

[11] D. C. Kozen. *Automata and Computability*. Springer, 1997. 6.1

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17:94–162, 1992. 12.1

[13] J. B. West, J. M. Fitzpatrick, S. A. Toms, and C.R. Maurer R.J Maciunas. Fiducial Point Placement and Accuracy of Point-based Rigid body registration. *Neurosurgery*, 48:810–817, 2001. 14.2

[14] K.E. Wiegers. *Peer Reviews in Software*. Addison-Wesley, 2002. 5.2.2

[15] L. Wingerd and C. Seiwald. High-level best practices in software configuration management. Technical report, Perforce, Inc., 1998. http://www.perforce.com/perforce/bestpractices.html. 5.2.4

INDEX