
IGSTK: The Book

For release 4.2

Edited by Kevin Cleary, Patrick Cheng,
Andinet Enquobahrie, and Ziv Yaniv



©2009 Insight Software Consortium

All rights reserved. No part of this book may be reproduced, in any form or by any means, without the express written consent of the copyright holders. An electronic version of this document is available from <http://www.igstk.org> and may be used under the provisions of the IGSTK copyright found at <http://www.igstk.org/copyright.htm>

Contributors to this project include those listed on the cover page as well as:

Cover design: Dave Klemm, Educational Media, Georgetown University

Editor: Cynthia Kroger

Logo design: Julien Jomier

Printed by: Signature Book Printing, Gaithersburg, Maryland.

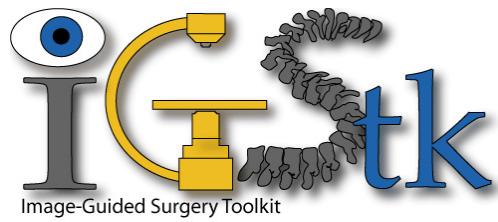
<http://www.signature-book.com>

IGSTK: The Book

Edited by
Kevin Cleary
Patrick Cheng
Andinet Enquobahrie
Ziv Yaniv

Friday 29th May, 2009

Website: <http://www.igstk.org>
Email: igstk-users@public.kitware.com



"Progress is the life-style of man."

- Victor Hugo, Les Misérables

About the Covers

The front and back covers show several images from the project.

The front cover shows the following.

- Top left. State diagram for the spatial object component.
- Top right. Lung biopsy clinical trial using IGSTK at Georgetown University Hospital. The attending physician is Filip Banovac, MD
- Bottom right. Architecture diagram showing tracker, spatial objects, spatial object representation, and viewers.

The back cover shows the four quadrant display and image reslicing from the Navigator example application.

Abstract

The Image-Guided Surgery Toolkit (IGSTK) is an open-source C++ software library that provides the basic components needed to develop image-guided surgery applications. The focus of the toolkit is on robustness using a state machine architecture.

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 CDash) 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 IGSTK and developing image-guided surgery applications with IGSTK.

Contributors

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

This book lists a few of these contributors in the following paragraphs. Not all contributors are credited here, so please check the CVS source logs for code contributions.

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

Kevin Cleary is an Associate Professor in the Department of Radiology's Imaging Science and Information Systems Center at Georgetown University Medical Center. His research focuses on image-guided surgery and medical robotics. He is the IGSTK principal investigator and manages the project. Contact him at cleary@georgetown.edu.

Patrick Cheng is a Software Engineer in the Imaging Science and Information Systems Center at Georgetown University. His research interests include medical imaging, image-guided surgery, and open source software development. He is one of the main developers of IGSTK and his major contribution is developing applications based on IGSTK.

Andinet Enquobahrie is a Research and Development Engineer at Kitware Inc. Dr. Enquobahrie has an extensive experience in development of image visualization and analysis tools for computer aided diagnosis and image-guided intervention applications. He has been actively developing various components of the IGSTK toolkit since joining Kitware in 2005. Currently, he is a project lead coordinating various aspects of the IGSTK project at Kitware.

Ziv Yaniv is an Assistant Professor in the Department of Radiology, Georgetown University, where he conducts research in image-guided interventions. He obtained his PhD in computer science from The Hebrew University of Jerusalem, Jerusalem Israel, in 2004. From 2004 to 2006 he was a postdoctoral fellow at Georgetown University. His main areas of interest are image-guided interventions, medical image analysis, and computer vision. Dr. Yaniv is a member of IEEE Engineering in Medicine and Biology and IEEE Computer societies.

Stephen Aylward is Chief Medical Scientist at Kitware, Inc. Prior to joining Kitware, Stephen was a tenured Associate Professor of Radiology and director of the Computer-Aided Diagnosis

and Display Laboratory at UNC. Dr. Aylward's research has recently focused on developing model-to-image registration strategies for image-guided surgery, vascular network segmentation for disease diagnosis, and digital library technologies such as the Insight Journal and MIDAS.

M. Brian Blake is a Professor of Computer Science at the University of Notre Dame. His research focuses on service-oriented computing, component-based software engineering, and workflow modeling. He was the main contributor for the requirements development for IGSTK.

Kevin Gary is an Assistant Professor in the Division of Computing Studies at Arizona State University's Polytechnic Campus. His interests are in teaching and applied research in Software Engineering, particularly distributed and web-based software architectures. Prior to academia, he worked in industry on open source solutions for eLearning.

David Gobbi is an expert in medical image analysis and visualization. He received a Ph.D. in Medical Biophysics from the University of Western Ontario, and is the original contributor of the Tracker component of IGSTK.

Özgür Güler is a computer scientist working as a research assistant at the Medical University Innsbruck, Austria. His research focuses on image-based diagnosis and therapy, visualization of medical imagery, image-guided navigation, and surgery. He is currently undertaking a PhD, researching novel paradigms for quality assurance in 3D-Navigation.

Luis Ibáñez is a Senior Research Engineer at Kitware Inc. He is one of the main developers and maintainers of the NLM's Insight Toolkit ITK. His main interests are medical image analysis and open source software as a mechanism for technology dissemination. He is also an advocate of open access publishing. He is one of the main contributors to the architectural design of IGSTK.

Julien Jomier is a Research and Development Engineer at Kitware Inc. He is a developer of the Insight Toolkit and also the main contributor of the Spatial Objects and the Spatial Object Viewer toolkit. His main areas of research include image-guided surgery and multi-modality data fusion as well as computer-aided diagnosis.

Hee-su Kim currently works for a game development company in South Korea. He was a graduate student in the Department of Computer Science at Kyungpook National University in Korea. He has interests in computer graphics, medical imaging, and related computer science fields.

Frank Lindseth is a research scientist at SINTEF Medical Technology and have been working within the national center for 3D ultrasound in surgery (Trondheim, Norway) since 1995 when the center was established. He has been working with application development based on IGSTK (CustusX), integrating real-time imaging in the toolkit (the VideoImager component) and contributing to the IGSTK design discussions (e.g. SurgicalSceenGraph and ImageReslice).

Sebastian Ordas is a biomedical engineer at IDEUNO, Argentina. His research focuses on image-guided surgery and medical image analysis. His main contributions to IGSTK are the image reslicing component and application examples.

Junichi Tokuda Junichi Tokuda is a research fellow of Radiology, Brigham and Women's Hos-

pital and Harvard Medical School. His research interests include computer assisted intervention especially hardware and software integration for image-guided therapy. He is one of the original developers of the OpenIGTLINK protocol.

Matt Turek is a R&D Engineer at Kitware Inc. Dr. Turek has extensive academic and industry experience in developing medical applications. He has been involved in the IGSTK project since joining Kitware in 2007. His key contributions are coordinate system and events architecture.

Hui Zhang is an imaging research engineer at Accuray Inc. He previously worked at the Imaging Science and Information Systems Center at Georgetown University. His research focuses on image-guided surgery, image registration, treatment planning, and visualization.

Funding Sources

- IGSTK Phase I and II (STTR) was funded by NIBIB/NIH (Georgetown-Kitware) Grant R42EB000374.
- The current IGSTK work is funded by NIBIB/NIH grant R01EB00719 under program officer Zohara Cohen, PhD.
- Additional support was provided by U.S. Army grant W81XWH-04-1-007, administered by the Telemedicine and Advanced Technology Research Center (TATRC), Fort Detrick, Maryland. The content of this manuscript does not necessarily reflect the position or policy of the U.S. Government.
- Özgür Güler's work on the VideoImager component was funded by the Austrian Science Foundation (Project 20604-B13).
- Partial funding support for OpenIGTLINK was provided by the following NIH grants 5U41RR019703, 5P01CA067165, 1R01CA111288, and 1R01CA124377, in addition to the Intelligent Surgical Instruments Project of METI(Japan).

CONTENTS

I Getting Started	1
1 Introduction	3
1.1 Rationale and Background	4
1.2 Software Organization	5
1.2.1 Obtaining the Software	5
1.2.2 Downloading the Stable Releases	6
1.2.3 Downloading the Development Release	6
1.2.4 Directory Structure	6
1.2.5 Documentation	7
1.2.6 Data	7
1.2.7 Join the Mailing List	7
2 Installation	9
2.1 Prerequisite	9
2.1.1 Third-Party Libraries	9
2.1.2 Build Tool	10
2.1.3 Caveat: Versions	10
2.2 Prepare for Installation	11
2.2.1 Download and Install CMake	11
2.2.2 Download and Install ITK, VTK, and FLTK	12
2.2.3 Caveat: Build Options	12

2.3	Configuring IGSTK	13
2.4	Hello World	15
3	Software Development Process	23
3.1	IGSTK Best Practices	23
3.2	Developer Practices	25
3.2.1	Code Conventions	25
3.2.2	Code Reviews	26
3.2.3	Managed Communication	27
3.2.4	Source Code Control	27
3.2.5	Build and Release Management Processes	30
3.2.6	Continuous Testing using CDash	31
	Software Quality Statistics	31
	IGSTK Approach	33
3.3	Agile Methods and Refactoring	36
3.4	Conclusion	37
4	Requirements	39
4.1	What is an IGSTK Requirement?	39
4.2	Lightweight Requirements Management Process	40
4.2.1	Defined Requirements Management Process	40
4.2.2	Managing Concurrent Change in Requirements and Code	42
4.3	Conceptualizing Application Requirements through Activity Modeling	44
4.4	Accessing and Contributing to IGSTK Requirements	45
II	Framework Design	47
5	Architecture	49
5.1	General Background	49
5.2	Medical Errors	51
5.3	Layered Architecture	52
5.4	The Main Components	53
5.4.1	Display	55
5.4.2	Geometric Representation	56

5.4.3	Visual Representation	58
5.4.4	Tracking	59
5.4.5	Data Sources	60
5.4.6	Calibration and Registration	61
5.4.7	Infrastructure	61
5.4.8	Services	62
5.5	Timing	63
5.5.1	Timing Collaborations	64
5.5.2	Pulse Generator Implementation	68
5.6	Conclusion	68
6	State Machine	71
6.1	General Background	71
6.2	Motivation	72
6.2.1	Deterministic Behavior	72
6.2.2	Preclude Wrong Use	73
6.2.3	Robustness to Misuse	74
6.2.4	Managing Complexity	75
6.2.5	Traceability	76
6.2.6	Suitability for Testing	76
6.2.7	Consistent Documentation	77
	Graphviz	77
	LTSA	78
	SCXML	78
6.3	Implementation	78
6.3.1	State Machine API	79
6.3.2	Safe States: Attempting Pattern	83
6.3.3	Communication Protocols	83
6.3.4	Events, Inputs, and Transduction	83
6.3.5	Integration Inside a Class	85
6.4	Conclusion	89
7	Events	91

7.1	General Background	91
7.2	Motivation	91
7.2.1	Class Decoupling	92
7.2.2	API Containment	92
7.2.3	Code Reuse	92
7.2.4	Error Handling	93
7.3	Implementation	93
7.3.1	Relationship with ITK	93
7.3.2	Events with Payload	94
7.3.3	Events and State Machines	95
7.3.4	Observers	96
7.4	Usage	97
7.4.1	Internal Usage	97
7.4.2	External Usage: The Request/Observe Pattern	98
7.5	Complete Event Hierarchy	102
7.6	Conclusion	103

III Core Components **105**

8 Coordinate Systems	107	
8.1	The Role of Coordinate Systems in IGSTK	107
8.1.1	Advantages and Disadvantages of the IGSTK Scene Graph	107
8.2	Mathematical Definition and Notation	108
8.2.1	Transform Composition	108
8.3	Coordinate System Scene Graph	109
8.3.1	Transform Computation	110
8.4	Coordinate System API	111
8.4.1	Transforms and Time Stamps	111
8.4.2	Coordinate System Classes	112
8.4.3	Coordinate System Macros	113
8.4.4	Classes that Have Coordinate Systems	114
8.4.5	Coordinate System Events	114
8.5	Coordinate Systems and Object Display	115

8.6	Scene Graph Visualization	115
8.6.1	Displaying Static Snapshots of the Scene Graph	116
8.6.2	Display Scene Graph in FLTK	116
8.6.3	Displaying Dynamic Changes and Transformations of the Scene Graph	117
8.7	Coordinate System Examples	118
8.7.1	Basic Example	118
8.7.2	View From Tracker Coordinates Example	120
8.7.3	View Follows Tracker Tool Example	123
8.7.4	"World" Coordinate System Example	127
9	Tracker	133
9.1	The Role of the Tracker Component in IGSTK	133
9.2	Structure of the Tracker Component	134
9.2.1	Communication	135
9.2.2	Threading	136
Safety	136	
Performance	136	
9.2.3	Buffering	137
9.2.4	Coordinate Transformations	138
9.3	Class Hierarchy	140
9.3.1	Tracker	141
State Machine	141	
Interface Methods	141	
Events	143	
9.3.2	Tracker Tool	144
State Machine	144	
Interface Methods	145	
Events	145	
9.4	Simulation and Testing	146
9.5	Hazardous Conditions	146
9.5.1	Tracking Device Failure	147
9.5.2	Loss of Accuracy	147
9.6	Example	148

9.6.1	MicronTracker Example	148
9.6.2	Polaris Tracker Example	151
9.7	Conclusion	154
10	Spatial Objects	155
10.1	Spatial Object Aggregation Hierarchy	155
10.2	State Machine	158
10.3	Component Interface	158
10.4	Common Objects	159
10.4.1	Axes Object	159
10.4.2	Box Object	161
10.4.3	Cone Object	161
10.4.4	Cylinder Object	162
10.4.5	Ellipsoid Object	162
10.4.6	Image Object	163
10.4.7	Mesh Object	165
10.4.8	Tube Object	166
10.4.9	Vascular Network and Vessel Objects	168
10.5	Reading Spatial Objects	170
10.6	Conclusion	171
11	Spatial Object Representation	173
11.1	Introduction	173
11.2	Displaying My First Object	173
11.3	State Machine	176
11.4	Component Interface	178
11.5	Common Object Representations	178
11.5.1	Axes Object	178
11.5.2	Box Object	178
11.5.3	Cone Object	178
11.5.4	Cylinder Object	181
11.5.5	Ellipsoid Object	181
11.5.6	Mesh Object	181

11.5.7	Vascular Network Object	181
11.6	Ultrasound Probe Representation	182
11.7	Sharing and Duplicating Object Representations	182
11.8	Conclusion	184
12	View	185
12.1	View Component Design	185
12.1.1	Synchronization Between Scene Generation and Rendering	185
12.1.2	GUI Interaction	185
12.2	State Machine	186
12.3	Component Interface	186
12.4	Example	187
12.5	Conclusion	189
IV	Services	191
13	Logging	193
13.1	General Background	193
13.2	Structure of the Logging Service	193
13.2.1	Logger Priorities	194
13.2.2	Logger Output Properties	194
Flushing	194	
Formatting	195	
Timestamp	196	
13.2.3	Logger Base Classes	196
LoggerBase	196	
Logger	196	
LoggerManager	196	
13.2.4	LogOutput	198
StdStreamLogOutput	198	
FLTKTextBufferLogOutput	199	
FLTKTextLogOutput	200	
MultipleLogOutput	200	

Extending LogOutput	201
13.2.5 Redirecting ITK and VTK Log Messages to Logger	201
Overriding itk::OutputWindow	201
Overriding vtkOutputWindow	202
13.2.6 Multi-Threaded Logging	202
LoggerThreadWrapper	202
ThreadLogger	203
13.3 Example	203
13.4 Conclusion	206
14 Image I/O	207
14.1 DICOM Reader	207
14.1.1 State Machine Design	208
14.1.2 Component Interface	208
14.1.3 Special Features	208
14.1.4 Example	210
14.2 Screenshot Generation	212
14.3 Conclusion	212
15 Registration	213
15.1 Landmark-Based Registration	213
15.1.1 State Machine Design	213
15.1.2 Component Interface	215
15.1.3 Example	216
15.2 Registration Error Prediction	220
15.2.1 State Machine Design	221
15.2.2 Component Interface	221
15.2.3 Example	223
15.3 Conclusion	224
16 Calibration	225
16.1 Precomputed Transformations, Reading and Writing Files	225
16.2 Pivot Calibration	227

17 Reslicing	231
17.1 General Background	231
17.2 Design	231
17.2.1 Class Hierarchy	231
17.2.2 Manual and Automatic Reslicing	233
17.2.3 Reslicing Modes	233
17.2.4 Image and Mesh Data Reslicing	233
17.2.5 State Machine Diagrams	234
igstk::ReslicerPlaneSpatialObject State Machine	234
igstk::ImageResliceObjectRepresentation State Machine	234
igstk::MeshResliceObjectRepresentation State Machine	234
17.2.6 Component Interface	234
igstk::ReslicerPlaneSpatialObject	234
igstk::ImageResliceObjectRepresentation	237
igstk::MeshResliceObjectRepresentation	237
17.3 Examples	237
17.4 Conclusion	239
18 VideoImager Component	241
18.1 The Role of the VideoImager Component	241
18.2 Structure of the VideoImager Component	242
18.2.1 Threading	243
18.2.2 Buffering	243
18.2.3 Frames and Timestamps	244
18.3 Class Hierarchy	244
18.3.1 VideoImager	245
State Machine	245
Interface Methods	245
Events	247
18.3.2 VideoImager Tool	249
State Machine	249
Interface Methods	249
Events	250

18.4 Example	251
18.4.1 VideoImager Component Example	251
18.5 Conclusion	255
19 OpenIGTLink	257
19.1 Applications and Use Cases	257
19.2 The OpenIGTLink Protocol	258
19.2.1 General Header Section	259
19.2.2 Data Body Section	260
19.3 The OpenIGTLink Library	261
19.4 Example: Exporting Tracking Data using the OpenIGTLink	262
20 State Machine Validation	267
20.1 Motivation	267
20.2 Background	268
20.3 Approach	269
20.3.1 State Machine Export	270
20.3.2 Global State Validation	271
20.3.3 State Machine Simulation	272
20.3.4 Visualization and Animation	272
20.3.5 Continuous Testing Integration	273
20.4 Installing and Running the Validation Toolset	273
20.4.1 Getting and Building the Validation Software	273
20.4.2 Running the Validation Software	274
20.4.3 Using the Tools	275
Exporting SCXML	276
Structural Analysis	277
Coverage Tools	277
Parsing Logfiles	279
Running Simulations	280
Rules as Test Conditions	280
Visualizing, Animating, Interacting with SMVIZ	281
20.5 Conclusion	282

20.6 Acknowledgements	282
V Example Applications	283
21 Tracking Working Volume Viewer	285
21.1 Introduction	285
21.2 Tracker-independent Application Development	286
21.3 Example	288
22 Needle Biopsy 2.0	291
22.1 Running the Application	291
22.2 Building the Scene Graph	293
23 Navigator	299
23.1 Introduction	299
23.2 Load Image	299
23.3 Load Mesh	301
23.4 Modify Fiducials	301
23.5 Configure Tracker	301
23.6 Register	303
23.7 Navigate	303
VI Legacy Applications	305
24 Needle Biopsy	307
24.1 Running the Application	308
24.2 Implementation	309
24.2.1 State Machine in Application	309
24.2.2 Mapping Clinical Work Flow to a State Machine	309
24.2.3 Coding the State Machine	311
24.2.4 Should I Use the State Machine in My Application?	313
24.3 Results	313
25 Ultrasound-Guided Radiofrequency Ablation	315

25.1	Introduction	315
25.2	Running the Application	315
25.3	Implementation	317
25.3.1	Tracker	317
25.3.2	Registration	318
25.3.3	Read and Display	318
25.4	Conclusion	319
26	Robot-Assisted Needle Placement	321
26.1	Running the Application	322
26.2	Implementation	323
26.2.1	Pass IGSTK Image Objects to ITK Filters	323
26.2.2	Write Your Own Representation Class	326
26.2.3	Using the Socket Communication Class	328
26.3	Result	329
VII	Appendices	333
A	IGSTK Style Guide	335
A.1	Purpose	335
A.2	Implementation Framework	335
A.2.1	Implementation Language	335
A.2.2	Generic Programming	335
A.2.3	Portability	336
A.2.4	CMake Configuration Environment	336
A.2.5	Doxygen Documentation System	336
A.2.6	vnl Math Library	336
A.2.7	Reference Counting and SmartPointers	336
A.2.8	CVS Environment	336
A.2.9	CDash Dashboard Testing Environment	337
A.3	Copyright	337
A.4	File Organization	338
A.5	Namespaces	339

A.6	Naming Conventions	339
A.6.1	Naming Classes	339
A.6.2	Naming Files	340
A.6.3	Naming Methods and Functions	340
A.6.4	Naming Class Data Members	340
A.6.5	Naming Local Variables	340
A.6.6	Naming Template Parameters	341
A.6.7	Naming Typedefs	341
A.6.8	Using Underscores	341
A.6.9	Preprocessor Directives	341
A.7	Const Correctness	342
A.8	Code Layout and Indentation	342
A.8.1	General Layout	342
A.8.2	Class Layout	342
A.8.3	Method Definition	343
A.8.4	Use of Braces	344
A.8.5	Use of Whitespace	344
A.9	Doxygen Documentation System	345
A.9.1	Documenting a Class	345
A.9.2	Documenting a Method	345
A.10	Using Standard Macros	345
A.11	Exception Handling	346
A.12	Documentation Style	346
A.13	Programming Practices	347
A.13.1	Choice of double or float	347
A.13.2	Choice of signed or unsigned	347
B	Glossary	349
Index		355

LIST OF FIGURES

2.1	CMake User Interface	14
2.2	“Hello World” Screen Shot	22
3.1	Dashboard Nightly Builds	34
3.2	Dashboard Continuous Builds	35
4.1	C-PLAD Requirements Process	40
4.2	IGSTK Requirements Management Process	43
4.3	Guidewire Tracking Activity Diagram	45
4.4	Requirements Management using Mantis BugTracker	46
5.1	Layered Architecture	52
5.2	Component Categories	54
5.3	Timing Architecture	64
5.4	Timing Architecture	66
5.5	PulseGenerator State Machine Diagram	69
6.1	State Machine Public/Private Separation	73
6.2	State Machine If Statement Equivalent	75
6.3	State Machine Attempting Pattern	84
6.4	State Machine Structure	85
7.1	Events Class Hierarchy	93

7.2 Event/Input Transduction	96
7.3 Events Usage	98
7.4 Request/Observe Pattern	101
7.5 Events Class Hierarchy	102
8.1 Transformation Example	109
8.2 Coordinate Systems Tree	110
8.3 Rendering the Scene Graph Using GraphViz	117
8.4 Scene Graph FLTK GUI	117
8.5 Transform Computation Path Display	118
9.1 Tracker Component	134
9.2 Tracker Buffer	137
9.3 Tracker Coordinates	139
9.4 Tracker Class Hierarchy	140
9.5 TrackerTool Class Hierarchy	140
9.6 Tracker State Machine	142
9.7 TrackerTool State Machine	144
10.1 Spatial Object State Machine	158
10.2 Spatial Object Types in IGSTK	160
11.1 Object Representation Example	176
11.2 Spatial Object Representation State Machine	177
11.3 Spatial Object Representation Types in IGSTK	179
11.4 Axes and Box Object	180
11.5 Cone and Cylinder Object	180
11.6 Ellipsoid and Mesh Object	181
11.7 Vascular Network Object	182
11.8 Ultrasound Probe Object	183
12.1 View2D State Machine	186
13.1 Logging Class Hierarchy	197

14.1 Image Reader Hierarchy	207
14.2 DICOMReader State Machine	209
15.1 Landmark Registration State Machine	214
15.2 Landmark Registration Error Estimator State Machine	222
16.1 Pivot Calibration Coordinate Systems and Transformations	228
16.2 FLTK Pivot Calibration Example Application	229
17.1 ReslicerPlaneSpatialObject Class Hierarchy	232
17.2 ImageResliceObjectRepresentation Class Hierarchy	232
17.3 MeshResliceObjectRepresentation Class Hierarchy	232
17.4 ReslicerPlaneSpatialObject State Machine	235
17.5 ImageResliceObjectReperesentation State Machine	236
17.6 MeshResliceObjectReperesentation State Machine	236
18.1 VideoImager Component	242
18.2 VideoImager Class Hierarchy	244
18.3 VideoImagerTool Class Hierarchy	245
18.4 VideoImager State Machine	246
18.5 VideoImagerTool State Machine	248
18.6 VideoImager Example Screen Shot	251
18.7 VideoImager Example Screen Shot 2	252
19.1 OpenIGTLLink Message Structure	259
19.2 The OpenIGTLLink Library Architecture	262
20.1 IGSTK Component Request-Event Exchange	271
20.2 Validation Tools Flow	275
20.3 State Machine Visualization Tool	281
21.1 Tracking Working Volume Viewer GUI	286
22.1 User Interface for the New Needle Biopsy Program	292
22.2 Building Scene Graph – Step 1	293
22.3 Building Scene Graph – Step 2	294

22.4 Building Scene Graph – Step 3	295
22.5 Building Scene Graph – Step 4	296
22.6 Building Scene Graph – Step 5	297
22.7 Building Scene Graph – Step 6	298
23.1 Navigator Interface	300
23.2 Navigator Workflow	300
24.1 System Setup for Needle Biopsy Application	308
24.2 State Machine Diagram (Partial) for the Needle Biopsy Application	310
24.3 User Interface for Needle Biopsy Program	314
25.1 Typical RFA Ablation Surgery Workflow	316
26.1 Drawing for Needle Placement Robot	321
26.2 Robot Assisted Needle Placement Phantom Study Setup	322
26.3 Clinical Workflow for Robot Assisted Needle Placement	324
26.4 User Interface for the Robot Application	330
26.5 Robot Needle Holder	331

LIST OF TABLES

4.1	Guideware Placement Workflow	44
6.1	igstkStateMachineMacroBase types	86
18.1	VideoImager: Supported Operating Systems	256

Part I

Getting Started

Introduction

Welcome to *IGSTK: The Image-Guided Surgery Toolkit* (pronounced "Eye-Gee-Stick").

IGSTK is an open source¹ software toolkit designed to enable biomedical researchers to rapidly prototype and create new applications for image-guided surgery [10]. 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 several open source software packages:

- The Insight Segmentation and Registration Toolkit (ITK)
- The Visualization Toolkit (VTK)
- GUI toolkits such as FLTK and Qt (other GUI toolkits can also 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 the scrutiny of the core developers. Since IGSTK is intended to be used in the operating room, its developers emphasize robustness motivated by the directive of protecting patient safety. In this context, IGSTK is a

¹<http://www.igstk.org/copyright.htm>

system that is open for discussion and willing to receive contributions from the community, but that will adapt and thoroughly test 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 BSD² licensing, you have access to all the source code and are free to incorporate the code in your program whether it 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 researchers 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, the research group has been attempting to standardize their software process based on the open

²BSD 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.

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 the University of North Carolina, Atamai Inc., and the Arizona State University later joined the project. This book and the associated software is the result of that effort.

1.2 Software Organization

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

1.2.1 Obtaining the Software

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

1. periodic stable releases are available on the IGSTK web site <http://www.igstk.org>;
2. development version can be obtained from the CVS source code repository (instructions found at IGSTK wiki page <http://public.kitware.com/IGSTKWIKI>).³

Official releases are available a few times a year and are announced on the IGSTK web pages and mailing lists. However, they may not provide the latest and greatest features of the toolkit. The development version 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 IGSTK version 4.2 release. If you are a new user, we highly recommend that you use the stable release version of the library. It is stable, consistent, and better tested than the code available from the repository. Later, as you gain experience with IGSTK, you may wish to work with the latest development version from the repository. However, if you do so, please be aware of the IGSTK quality testing dashboard. IGSTK is heavily tested using the open source CDash regression testing system (<http://public.kitware.com/dashboard.php?name=igstk>). Before updating your local copy of the source code, make sure that the dashboard is *green* indicating stable code. If it is not green it is likely that the current source code is unstable. (Learn more about the IGSTK quality dashboard in Section 3.2.6 on page 31.)

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

<http://www.igstk.org/download.htm>

³Editor's note: at this writing, we are considering switching to Subversion for source code control.

You can get a zipped file of a stable release or you can get the development version from the repository. The release version is stable and dependable but may lack the latest features of the toolkit. The development version will have the latest additions but might be unstable and contain work-in-progress components. The following sections describe the details of each one of these two alternatives.

1.2.2 Downloading the Stable Releases

Choose the zipped file 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 `IGSTK` will be created in your disk and you will be ready for starting the configuration process described in Section 2.3 on page 13.

1.2.3 Downloading the Development Release

Development release refers to the latest version in the source code repository. The repository is a tool for software version control and is mostly design for and used by developers. For more information about source code control, please read Section 3.2.4 on page 27. (Note: if you decide to use the development release, make sure to update your source code only when the IGSTK quality dashboard is stable. Learn more about the quality dashboard at Section 3.2.6 on page 31.)

Detailed instructions on how to access the IGSTK source code repository can be found on the this page:

http://public.kitware.com/IGSTKWIKI/index.php/Download_IGSTK

Once you obtain the software, you are ready to configure and compile it (see Section 2.3 on page 13). First, however, we recommend that you join the mailing list (Section 1.2.7) and read the following sections describing the organization of the software.

1.2.4 Directory Structure

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

IGSTK is organized into several different modules. If you are using an official release, you will see three important modules: the `Source`, `Testing`, and `Examples` modules. The source code can be found in the `Source` module; testing code, input data, and baseline images can be found in `Testing`; and example applications using IGSTK are available in the `Examples` directory. There is also an `Utilities` directory containing some auxiliary scripts. The `Source` directory

contains all the components of IGSTK. When you are working with this library, you can refer to the code in Testing and Examples to learn how the classes are being used.

1.2.5 Documentation

The user guide book (this book) is the most comprehensive documentation for the IGSTK project. The electronic version of this book (pdf) can be obtained from these two sites:

<http://www.igstk.org/documentation.htm> or
<http://public.kitware.com/IGSTKWIKI>

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 IGSTK. 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-line at <http://public.kitware.com/IGSTK/NightlyDoc/>. The IGSTK home page also has a link to this Doxygen manual page.

Header Files. Each IGSTK 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 produce its output.)

1.2.6 Data

There are some small testing data sets available in the IGSTK/Testing/Data directory. Larger data sets will be provided at the Open Data Repository (<http://www.insight-journal.org/dspace/handle/1926/378>) hosted on the MIDAS system at Kitware.

1.2.7 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. If you want to be part of the IGSTK development team, you may be able to join the developers' mailing list. You can subscribe to these two lists online at

<http://igstk.org/IGSTK/project/getinvolved.html>

The IGSTK-Users mailing list is 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. IGSTK developers are committed to creating a self-sustaining open source IGSTK community. Feedback from users is fundamental to achieving this goal.

Installation

This section describes the process for installing IGSTK on your system. Keep in mind that IGSTK is a toolkit, and as such, once it is installed in your computer there will be no application to run. Rather, you will use IGSTK to build your own applications. What IGSTK does provide - besides the toolkit proper - is a large set of test files and examples that will introduce you to IGSTK concepts and will show you how to use IGSTK in your own projects.

IGSTK is designed to compile on a set of target operating system/compiler combinations. These combinations include:

- Microsoft Windows, Visual C++ 7.1
- Microsoft Windows, Visual C++ 8.0
- Microsoft Windows, Visual C++ 9.0
- Linux GCC 4.0 and above
- MacOSX GCC 4.0

The version of Visual C++ 6.0 was intentionally left out due to its poor support for some important C++ features, such as partial specialization of templates.

2.1 Prerequisite

2.1.1 Third-Party Libraries

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

- The Insight Segmentation and Registration Toolkit (ITK)
- The Visualization Toolkit (VTK)

- GUI toolkits such as FLTK and Qt (other GUI toolkits can also be accommodated)

For an initial installation of IGSTK you must have ITK and VTK installed first to just build the toolkit itself. Some of the examples distributed with IGSTK require third-party libraries, such as FLTK. You may also need to install those libraries to compile the example applications.

2.1.2 Build Tool

IGSTK supports cross-platform builds by using CMake, a cross-platform and open-source build system. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native “Makefiles” and “Solutions” that can be used in the compiler environment of your choice. CMake is quite sophisticated - it supports complex environments requiring system configuration, compiler feature testing, and code generation.

CMake generates “Makefiles” under UNIX and Cygwin systems and generates Visual Studio “Solution” under Windows (and appropriate build files for other compilers like Borland). The information used by CMake is provided by `CMakeLists.txt` files that are present in every directory of the IGSTK source tree. These files contain information that the user provides to CMake at configuration time. Typical information includes paths to utilities in the system and the selection of software options specified by the user.

CMake manages the build process for ITK, VTK, FLTK, and IGSTK.

2.1.3 Caveat: Versions

IGSTK Release **4.2** is built and tested with the following toolkits and their specific versions.

- ITK. Release **3.10.1**
- VTK. Release **5.2**
- FLTK. Take the zip file with the snapshot of FLTK **1.1** on the IGSTK Wiki – “How to build IGSTK” page.
http://public.kitware.com/IGSTKWIKI/index.php/How_to_build_IGSTK
- Qt. Release **4.3**
- CMake. Release **2.6**

FDA guidelines for software development¹ require that medical application/product developers must specify the operation system, compiler, any Off-The-Shelf (OTS) software that the application is linking to, and the OTS software’s version (In this context, ITK, VTK, FLTK,

¹“General Principles of Software Validation; Final Guidance for Industry and FDA Staff;”
<http://www.fda.gov/cdrh/comp/guidance/938.html>.

and IGSTK are to be considered as OTS products to support your application). FDA guidelines for software validation² state that the validation process should be performed against a specific version of any OTS software package. If the software developer changes the version of the OTS software library the application is linking to, he has to go through the validation process again and to demonstrate that the application is tested with the new version of the OTS software library.

It is very important to make sure that you use the appropriate version of these required toolkits to build your IGSTK application. This is to ensure that the third-party toolkits behave as expected in the context of your application.

For instance, an IGSTK user reported a problem with the `igstk::Landmark3DRegistration` class. Further investigation revealed that it was a bug in one of the ITK classes. This was reported to the ITK development team, and they fixed it by patching ITK Release 3.0 to make an ITK Release 3.0.1. IGSTK developers updated the version specification accordingly and announced it in the mailing list and on the web site. If an application developer ignores this and develops applications using an older version of the ITK library, the registration result will be incorrect. If a physician uses this application in the surgery room, it may put the patient in danger. Therefore, version control is extremely important when developing an image-guided surgery application.

When configuring IGSTK with CMake, CMake will check if you are using the right version of ITK and VTK. It will prompt error messages when the wrong version of libraries are used.

2.2 Prepare for Installation

2.2.1 Download and Install CMake

CMake can be downloaded at no cost from

<http://www.cmake.org>

IGSTK requires at least CMake version 2.4. You can download binary versions for most of the popular platforms including Windows, Mac, and Linux. Alternatively, you can download the source code and build CMake on your system. Follow the instructions in the CMake Web page for downloading and installing the software.

Running CMake initially requires that you provide two pieces of information: where the source code directory is located (`IGSTK_SOURCE_DIR`), and where the object code is to be produced (`IGSTK_BINARY_DIR`). These are referred to as the *source directory* and the *binary directory*. We recommend setting the binary directory to be different than the source directory (an *out-of-source* build), but IGSTK will still build if they are set to the same directory (an *in-source* build). On Unix, the binary directory is created by the user and CMake is invoked with the path to the source directory. For example:

²“Off-The-Shelf Software Use in Medical Devices,” <http://www.fda.gov/cdrh/ode/guidance/585.html>.

```
mkdir IGSTK-binary  
cd IGSTK-binary  
cmake Path_To_IGSTK
```

On Windows, the CMake GUI is used to specify the source and build directories (Figure 2.1).

CMake runs in an interactive mode in that you iteratively select options and configure according to these options. The iteration proceeds until no more options remain to be selected. At this point, a generation step produces the appropriate build files for your configuration.

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

2.2.2 Download and Install ITK, VTK, and FLTK

After you have installed CMake, you can download ITK, VTK, and FLTK (or Qt), and use CMake to configure and then build these libraries. The download link for these libraries are given as follows. Choose the right version for this release of IGSTK as specified in Section 2.1.3.

ITK download page:

<http://www.itk.org/HTML/Download.htm>

VTK download page:

<http://www.vtk.org/get-software.php>

FLTK download page (a snapshot of version 1.1):

http://public.kitware.com/IGSTKWIKI/index.php/Download_IGSTK

After downloading the source code of these packages, you need to run CMake on them and then compile the generated “Makefile” or “Solution” file. You can disable the `BUILD_EXAMPLE` and `BUILD_TESTING` options to speed up the build process.

2.2.3 Caveat: Build Options

When you build ITK, VTK, and IGSTK, make sure you follow the same build process for all the libraries.

Do not configure libraries with CMake 2.4 and one library with CMake 2.2.

Do not compile one library with GCC 3.4 and one library with 4.1.

Do not build one library under “Debug” mode and one library under “Release” mode.

Do not build one library as “Static” library and one library as “Shared” library.

Use the same settings when building the libraries. For example, if you want to develop application using “Visual Studio 2005” under “Debug” mode based on IGSTK release 4.0, you should use CMake 2.6 to configure all the libraries with the right version (see Section 2.1.3) for “Visual Studio 8 2005”, and then compile all of them using “Visual Studio 2005” under “Debug” mode with “Static” library. If you want to test the application under “Release” mode, you should repeat the above process again and change the build option for all the libraries from “Debug” to “Release”, and build them again. Make sure you are linking to the right library, if you have multiple versions and builds in your system.

The reason for enforcing this rule is that you may find link errors if you mix build options for these libraries. If you force the compiler to go through the link process, you will get an unstable executable. This is very dangerous, as it might crash at any time. For example, C/C++ structures have different sizes under “Debug” mode versus “Release” mode. When a pointer to a structure increments, because the step size is calculated differently under different build options, it might cause the pointer to reference an invalid memory address and lead to unpredictable behavior or crash the application.

CMake has a built-in mechanism to check ITK build options versus VTK build options when they are used together. Application developers of IGSTK should take extra caution when building their applications and linking them to third-party libraries.

2.3 Configuring IGSTK

Figure 2.1 shows the CMake interface for UNIX and MS-Windows. To speed up the build process you may want to disable the compilation of the testing and examples. This is done with the variables `BUILD_TESTING=OFF` and `IGSTK_BUILD_EXAMPLES=OFF`. The examples distributed with the toolkit are a helpful resource for learning how to use IGSTK components but are not essential for the use of the toolkit itself. The testing section includes a large number of small programs that exercise the capabilities of IGSTK classes. Due to the large number of tests, enabling the testing option will considerably increase the build time. It is not desirable to enable this option for a first build of the toolkit.

Begin running CMake by using `ccmake` on Unix, and `CMakeSetup` on Windows. Remember to run `ccmake` from the binary directory on Unix. On Windows, specify the source and binary directories in the GUI, then begin to set the build variables in the GUI as necessary. Most variables should have default values that are sensible. Each time you change a set of variables in CMake, it is necessary to proceed to another configuration step. In the Windows version this is done by clicking on the “Configure” button. In the UNIX version this is done in an interface using the curses library, in which you can start the CMake configuration by pressing the “c” key.

There might be error messages indicating that ITK or VTK packages cannot be found. You

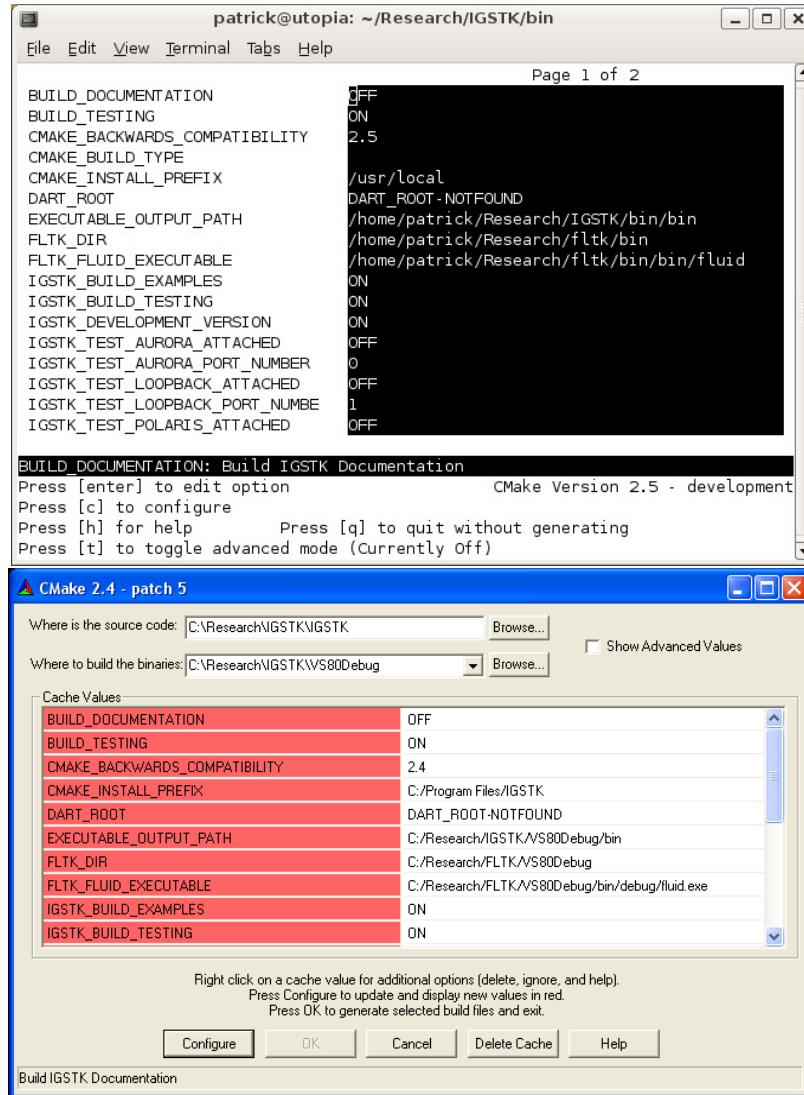


Figure 2.1: CMake Interface. Top) `ccmake`, the UNIX version based on `curses`. Bottom) `CMakeSetup`, the MS-Windows version based on `MFC`.

need to set the path to those libraries manually and verify that you are using the correct versions of them built with consistent compiler options. The following options must be set during the configuration:

- `ITK_DIR`. The path to the ITK binary directory in your system.
- `VTK_DIR`. The path to the VTK binary directory in your system.
- If you set `IGSTK_USE_FLTK=ON`, you also have to set the following two options:
 - `FLTK_DIR`. The path to the FLTK binary directory in your system.
 - `FLTK_FLUID_EXECUTABLE`. The path to the “fluid.exe” (FLtk User Interface Design tool) file in your system.
- If you set `IGSTK_USE_Qt=ON`, you need to specify Qt library and header file directory

When no new options appear in CMake, you can proceed to generate Makefiles or Visual Studio Solution (or appropriate build file(s) depending on your compiler). This is done in Windows by clicking on the “Ok” button. In the UNIX version this is done by hitting the “g” key. After the generation process, CMake will quit silently. To initiate the build process on UNIX, simply type `make` in the binary directory. Under Windows, load the solution named `IGSTK.sln` from the binary directory you specified in the CMake GUI.

The build process will typically take anywhere from 15 to 30 minutes depending on the performance of your system. If you decide to enable testing as part of the normal build process, about 80 small test programs will be compiled. This will verify that the basic components of IGSTK have been correctly built on your system.

If you configured the IGSTK with `IGSTK_BUILD_EXAMPLES=ON` and `IGSTK_USE_FLTK=ON` (and you have FLTK installed in your system), you will have example applications compiled when you build the whole package. In the next section, we will walk you though the “Hello World” example in IGSTK.

2.4 Hello World

This example illustrates the minimal application 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, cylinder and 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 to as a TrackerTool. In this minimal example, we use a MouseTracker, which is a class intended mainly for demonstration and debugging purposes. This tracker gets 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 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 the GUI from the main program of our application, we must include the header file generated by *fluid*. This is done as follows:

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

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

```
#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. Here we include 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 as a real tracking device but with the convenience of running based on the movement of the mouse in the screen. The header file for this class is as follows:

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

Since image-guided surgery applications are used in a safety-critical environment, it is 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 forwards 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 "igstkLogger.h"
#include "itkStdStreamLogOutput.h"
```

We are now ready to write the code of the actual application. Of course, 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 initializes the parameters of the clock. Timing is critical for all operations performed in an IGS application. Timing signals make it possible to synchronize the operation of different components and to ensure that the scene rendered on the screen actually displays a consistent state of the environment on the operating room. The command for the timer initialization is as follows:

```
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, as follows:

```
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);
```

```
// Add the position of the cylinder with respect to the View.
igstk::Transform transform;
transform.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );
cylinder->RequestSetTransformAndParent( transform, m_GUI->View );

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 as follows:

```
m_GUI->View->RequestAddObject( ellipsoidRepresentation );
m_GUI->View->RequestAddObject( cylinderRepresentation );
```

Function RequestEnableInteractions() allows the user to interactively manipulate (rotate, pan, zoom 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 the original VTK key-mouse-combined interactions are not supported. In summary, the mouse events are as follows: left button click triggers pick event; left button hold rotates the camera, in igstk::View3D (in igstk::View2D, camera direction is always perpendicular to image plane, so there is no rotational movement available for it); middle mouse button pans the camera; and right mouse button dollies the camera.

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

The following code instantiates a new mouse tracker and initializes 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->RequestOpen();
tracker->SetScaleFactor( 100.0 );

typedef igstk::MouseTrackerTool           TrackerToolType;
typedef TrackerToolType::TransformType   TransformType;
typedef igstk::TransformObserver         ObserverType;

// instantiate and attach wired tracker tool
TrackerToolType::Pointer trackerTool = TrackerToolType::New();
std::string mouseName = "PS/2";
trackerTool->RequestSetMouseName( mouseName );
//Configure
trackerTool->RequestConfigure();
//Attach to the tracker
trackerTool->RequestAttachToTracker( tracker );
ObserverType::Pointer coordSystemAObs = ObserverType::New();
```

```
coordSystemAObserver->ObserveTransformEventsFrom( trackerTool );  
  
TransformType identityTransform;  
identityTransform.SetToIdentity()  
    igstk::TimeStamp::GetLongestPossibleTime() );  
  
ellipsoid->RequestSetTransformAndParent( identityTransform, trackerTool );  
  
// Attach a viewer to the tracker  
m_GUI->View->RequestSetTransformAndParent( identityTransform, tracker );  
  
m_GUI->SetTracker( tracker );
```

Now a logger is set up. The log output is directed to both the standard output (`std::cout`) and a file (`log.txt`). For the usage of priority level, please refer to Chapter 13 on page 193.

```
typedef igstk::Object::LoggerType           LoggerType;  
  
LoggerType::Pointer logger = LoggerType::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 View and the Tracker, messages from these components are redirected to the logger, as follows:

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

Next, the refresh frequency of the display window is set. The `Show()` method of the GUI will invoke internally the `RequestStart()` method of the View. After the `RequestStart()` function is called, the pulse generator inside the display window will start ticking, and will call the display to update itself 60 times per second, as follows:

```
m_GUI->View->SetRefreshRate( 60 );  
m_GUI->Show();
```

Here we reset the camera position so that we can observe all objects in the scene. We have deferred this call to shortly before the main event loop so that all the coordinate systems have been set up.

We need to have coordinate system connections between the view and each object that we wish to display. For instance, the cylinder's coordinate system is attached directly to the view. The ellipsoid, however, is attached to the tracker tool. The tracker tool is attached to the tracker, which has an attached view.

```
m_GUI->View->RequestResetCamera();
```

All applications should include 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, which will allow the separate tracker thread to update itself. It is very important to include the igstk::PulseGenerator::CheckTimeouts() in the main thread. Otherwise, trackers will not update. After checking time out, the application request the transform of the tracker tool using event through the Coordinate system API (see Chapter 8). The command is as follows:

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

    typedef ::itk::Vector<double, 3>      VectorType;
    typedef ::itk::Vesnor<double>           VesnorType;
    TransformType                           transform;
    VectorType                             position;

    coordSystemAObserver->Clear();
    trackerTool->RequestGetTransformToParent();
    if (coordSystemAObserver->GotTransform())
    {
        transform = coordSystemAObserver->GetTransform();
        position = transform.GetTranslation();
        std::cout << "Trackertool :"
              << trackerTool->GetTrackerToolIdentifier()
              << "\t\t Position = (" << position[0]
              << "," << position[1] << "," << position[2]
              << ")" << std::endl;
    }
}
```

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

```
tracker->RequestStopTracking();
tracker->RequestClose();
delete m_GUI;
ofs.close();
return EXIT_SUCCESS;
```

When IGSTK is built with the CMake option “IGSTK_BUILD_EXAMPLES” on, the “HelloWorld” application will be built automatically with the toolkit. You will see a user interface like Figure 2.2 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 statements similar to the following :

```
...
24445310681.5099 : (DEBUG) draw() called...
24445310681.5114 : (DEBUG) UpdateSize() called...
24445310681.5263 : (DEBUG) Tracker::RequestStartTracking called...
24445310681.5287 : (DEBUG) State transition is being made:(...omited...)
24445310681.5341 : (DEBUG) Tracker::AttemptToStartTrackingProcessing
                     called ...
24445310681.5373 : (DEBUG) State transition is being made:(...omited...)
24445310681.5425 : (DEBUG) Tracker::StartTrackingSuccessProcessing
                     called...
24445310681.5454 : (DEBUG) Tracker::EnterTrackingStateProcessing
                     called...
...
...
```

These are the log messages from different IGSTK classes. They give you information on time stamps, priority levels, function calls, and state machine transitions. This log file can be used for debugging purposes and clinical procedure reviews.

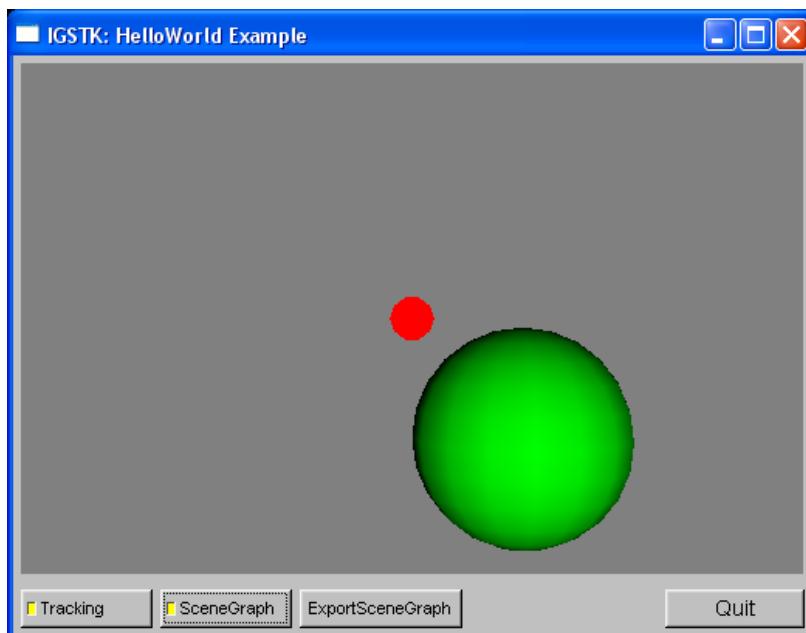


Figure 2.2: Screen Shot of "Hello World" Example.

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.

3.1 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. IGSTK’s 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 codeline policies to satisfy different quality criteria. Codelines with separate policies - for example, a main repository and a sandbox - let developers collaborate on code that might not yet meet stringent requirements. Exploiting source code control approaches early in a project helps document and track quality progress.
5. Focus on 100 percent code and path 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 CDash (<http://www.cdash.org>) 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 CDash, 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; Mantis Bug Tracker (<http://www.mantisbt.org>) 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 and collaborative nature of open source software development. It is not beneficial to create strict process controls that cannot, 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 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, builds, testing, release management are all performed in a highly iterative continuous manner to ensure the quality and safety of the software produced.

3.2 Developer Practices

3.2.1 Code Conventions

Understandability of source code is critical 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 include 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, as the unnecessary use of templates may lead to type-related runtime errors.

IGSTK advocates the use of “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, “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 enforce Set/Get method 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 A.9 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 CDash dashboard. The KWStyle tool removes the time-consuming, tedious process of checking for stylistic conformance and ensures that 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 while “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 CDash 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 Section 3.2.2 below) always include detailed checks to ensure that these conventions are followed.

3.2.2 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 source code control 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 [6, 28]), but they 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 has 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 CDash to facilitate reviewing the code. Defects found by the reviewer are first posted to the Wiki to give the original developer an opportunity 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 in which it is decided that 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.

Just 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.

3.2.3 Managed Communication

IGSTK was developed by a team of developers who are 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 of 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 with the toolkit, and the principal vehicles for this are the communication tools described here. We encourage you to become a participant in the IGSTK community.

3.2.4 Source Code Control

It is not always obvious how 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 with the combination of such tasks whether simultaneously or in sequence.

As an illustration of how much software surround us, here are some modern devices and the

number of lines of code that they contain:

- A laundry appliance has a couple thousand lines of code.
- The Joint Strike Fighter F-35 has 19 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 numbers of lines of code in IGSTK and the toolkits it depends on. The units are “KLOC”, or thousands of lines of code.

- IGSTK: 60 KLOC, 20 of which are for testing
- ITK: 634 KLOC, 116 of which are for testing
- VTK: 1414 KLOC, 148 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 source code control system.

Source code control (SCC) is a segment of configuration management important for enforcing and maintaining the quality of software products³. Most developers are familiar with using SCC 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 into the code repository. If any changes were made to the same files by other developers, the SCC 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 SCC system.

IGSTK uses the popular and well-known Concurrent Version System (CVS) tool for version control of software products⁴. The commands needed for obtaining IGSTK and supporting

¹<http://www.gao.gov/cgi-bin/getrpt?GAO-06-356>

²<http://www.stsc.hill.af.mil/crossstalk/1996/01/Boeing777.asp>

³We distinguish *configuration management* from source code control in IGSTK by including the software *build process* as managed by CMake (see Chapter 2).

⁴Editor’s note: at this writing, we are considering switching to Subversion for source code control.

software packages such as ITK from CVS are described in Chapter 1 and Chapter 2. IGSTK application developers 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 SCC 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 in the above scenarios. However, IGSTK goes further by supporting best practices in SCC, 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. In SCC terms (and specifically in CVS), a *branch*, is a named (or *tagged*) set of source code files that is 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 representatives of all stakeholders of the codeline.

Another reason to branch might be to support custom features for a particular customer or project that is 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, a generally accepted axiom is to branch as late as possible and as seldom as possible to avoid this complexity.

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 in which branch of a codeline a modification can be committed. 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 in 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 that they may only do so after a complete unit test and code review of the source code. An experimental codeline, in which 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 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 enforces quality standards better 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 can prototype high-risk code. For mini-projects that were deemed successful, the code was then subjected to more rigorous quality policies so that 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, as well as 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 SCC patterns when developing and supporting applications that depend on IGSTK. Only through the application of consistent quality policies can the quality and safety of software for surgical environments be achieved. For further reading on best practices for SCC and Configuration Management, we recommend [29].

3.2.5 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. The 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 main and sandbox repositories are frozen and tagged.
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 zip files 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 Section 1.2.1 and Chapter 2.

Finally, we emphasize one related best practice in IGSTK. 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.

3.2.6 Continuous Testing using CDash

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” produced for the U.S. Department of Commerce’s National Institute of Standards and Technology (NIST) found in 2002 that

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.⁵

It is important to note that this report is gathering statistics from application domains that are not considered to be mission-critical. That is, the cost referred to in this study do not include software error events such as the Mars Polar Lander crash in 1999 (\$165 million); the Ariane 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. As the study further explains,

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.

⁵<http://www.nist.gov/director/prog-ofc/report02-3.pdf>

IGSTK Approach

IGSTK relies on extensive automated unit testing to ensure that all lines of code are verified along some execution path. Automation is provided by CMake and CDash integration. CDash is a regression testing system that supports web-based report generation for a variety of test types. The web reports generated by CDash create 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 3.1 and Figure 3.2.

ITK and VTK developers and users should already be familiar with the CDash 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 [15] 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 and VTK 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 *code 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.



Figure 3.1: CDash Dashboard Screenshot: Nightly Builds.

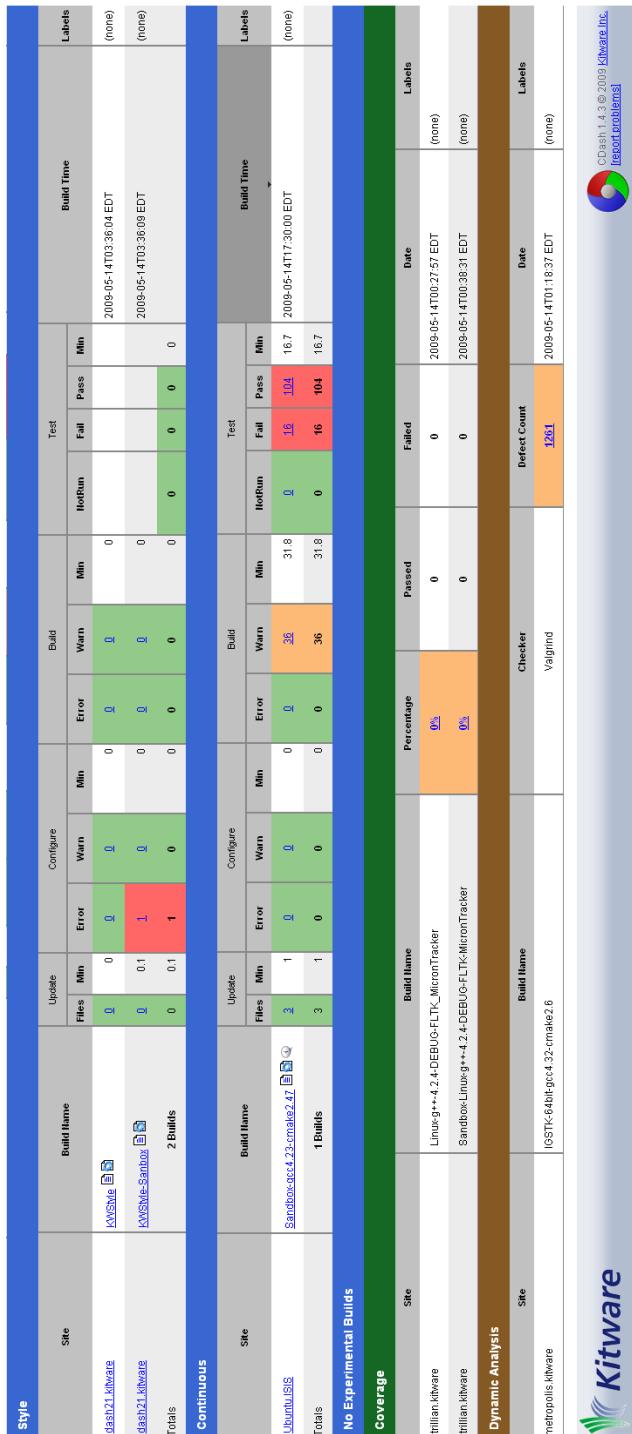


Figure 3.2: CDash Dashboard Screenshot: Code Style, Continuous Builds, Code Coverage, Memory Testing.

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

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 the CDash dashboard provides the best way to do this. CDash is freely available; IGSTK application developers are encouraged to contribute CTest submissions to the IGSTK CDash dashboard, as well as to set up their own CDash 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 set up their own CDash dashboard.

Finally, IGSTK believes in defect tracking , and uses a customized implementation of the open source Mantis BugTracker (<http://www.mantisbt.org>). 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 list is revisited at least once every internal release boundary.

3.3 Agile Methods and Refactoring

The previous sections describe best practices in the IGSTK software development process. These practices are understood in the Agile community, IGSTK’s contribution is in their application to a safety-critical system. The principal benefit is in the continual adherence to community standards for the project, ensuring low-level code quality and transparency of the process, critical in an open source, community-based process.

But what happens when the software requires *rework* instead of just *refactoring*? While pure Agilists might suggest this never should be the case, we have found in practice that eventually one must consider more extensive changes to the code, and maintaining the continual integration quality thresholds on the existing codebase while making these changes is simply counter-productive. Throughout this revised edition of the IGSTK book, new components and major reworkings are described (such as the Tracker). What matters is how the team addressed these changes - they were done with the same diligence as the original creation of the codebase.

IGSTK state machines are tightly coupled to IGSTK components - by intent. The state machine enforces the governance policy on component objects at runtime, therefore it is appropriate from a design standpoint to be highly coupled (though at a class level the StateMachine class remains decoupled). The IGSTK team understood the need to totally rework the Tracker classes to ensure forward support of new Tracker devices, and as such undertook an intrusive reworking of the Tracker component (see Chapter 9).

The IGSTK team at first approached this new development by trying to rework the classes in the sandbox (see Section 3.2.4). However, the purpose of the sandbox was to introduce and validate experimental functionality and algorithms, and as such the sandbox code maintained a

strong build dependency on the mainline. For this rework, attempting to put in new architectural patterns quickly broke these build dependencies.

Practicing the principle of *people before process* (see Section 3.1), the IGSTK team decided to relax the main codeline’s quality threshold constraints to complete this rework. While relaxing other best practices, including the continuous integration constraint, IGSTK applied its first and foremost best practice over others. We also point out that this is the first and only time a process modification and rework has been required in over 5 years of active community development on a rapidly growing project, and this is a pretty good track record!

The struggle, of course, is also having the courage to know when the rework is done and when to return to a daily reinforceable set of quality constraints on the project. IGSTK has completed the rework, updated its code reviews and unit tests, and turned control back to the high quality constraints evidenced on the IGSTK CDash dashboard.

3.4 Conclusion

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 one 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 to ensure the continued high-quality and safety goals of IGSTK. The IGSTK development community welcomes not only questions on the design aspects embodied in the source code, but also on the proper execution of agile and test-driven methods to attain these goals.

Requirements

“What we do is never understood, but only praised and blamed.”

—Friedrich Nietzsche

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 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 clinicians and other SMEs. As such, agile development approaches are more inline with this domain. Approaches to requirements engineering and management are not well-defined within typical agile development methodologies. In our work, we introduce a customized extension to agile development techniques through the introduction of an *agile* requirements management approach.

4.1 What is an IGSTK Requirement?

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

IGSTK requirements are classified into three areas that represent the three different end users of the system. These three types of requirements are *architecture requirements*, *style guidelines requirements*, and *application requirements*. 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 define how the target system will execute with respect to its use in a surgical environment. It is inevitable that some overlap will exist between these categories.

These classifications were derived from earlier work which defines a product line development process called Component-Based Product Line Analysis and Development (C-PLAD, see Figure 4.1). Interestingly, as applications are built from the underlying components, the application requirements extend the component-level requirements.

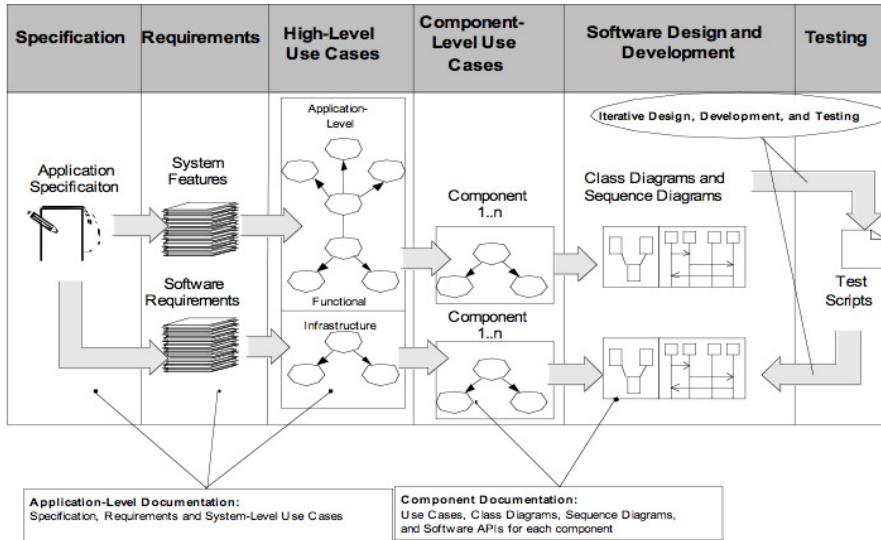


Figure 4.1: C-PLAD Conceptual Process Model.

4.2 Lightweight Requirements Management Process

4.2.1 Defined Requirements Management Process

IGSTK defined requirements management process follows two scenarios. The "formal requirements" follows a formal approach for introducing new software whereas the "fast-tracked requirements" is inline with quick changes or modifications. Step-by-step details of the process are described below:

Steps Common to Both Scenarios:

1. Framework 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.

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 formal, during weekly teleconferences.
5. When a requirement is discussed and approved in a preliminary state, then the requirements lead will move the requirement to the Mantis BugTracker.
 - The requirement should be classified (i.e. Severity Field) as either "Style Guideline", "Design Guideline", or "Functional Requirement".
 - The requirement should be put in the assigned status ("New"/"Unconfirmed"/"Assigned").
 - The number should be picked based on REQ XX.XX.XX format (e.g. REQ 06.02.10).
 - The requirement number should be appended in bold to the requirement text in the Wiki.

Formal Requirements Development Process Steps:

6. If the requirement is tagged for an iteration, then the software should be developed that corresponds to the requirement.
7. Once the code is developed it is moved to the Sandbox.
8. As the code is tested and reviewed, this may require changes to the requirements.
 - Changes to the requirement text should be entered to the Wiki and discussed during the teleconference.
 - Once a change is agreed upon, again the Requirements Lead makes the correction to the bug tracker by adding a comment.
 - The changed requirement should be stated as NEW_REQ_TEXT: *New requirements text* (e.g. 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.

Fast-Tracked Requirements Development Process Steps:

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 teleconference. Any changes to the requirement text should be entered to the Wiki as discussed during the teleconference.
9. Code should be moved to the main repository and requirements text updated in the bug tracker.

4.2.2 Managing Concurrent Change in Requirements and Code

IGSTK is a framework upon which surgical applications are constructed, therefore requirements are ever-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 of users applying the toolkit to different applications. Every new application brings new requirements in the form of desirable new features, or the modification of existing features.

The process for requirements management is significantly integrated with code management. IGSTK sees 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, integrated with application development. This process is illustrated as a UML state diagram in Figure 4.2.

In the initial requirements phase, general requirements for the framework are discovered. As the components for these initial requirements are developed, we often discover that additional requirements exist (e.g. specific validation requirements). Once a developer identifies new potential requirements (Initialized box in Figure 4.2), the developer posts a textual description (Defined) on the shared web site (Wiki). At the same time, the initial code that fulfills the requirement is entered into the Sandbox. The requirement then undergoes an iterative review subprocess where team members review, discuss, and potentially modify the requirement. Based on the team’s 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.

The IGSTK project uses an open source bug tracker (Mantis Bug Tracker) 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.

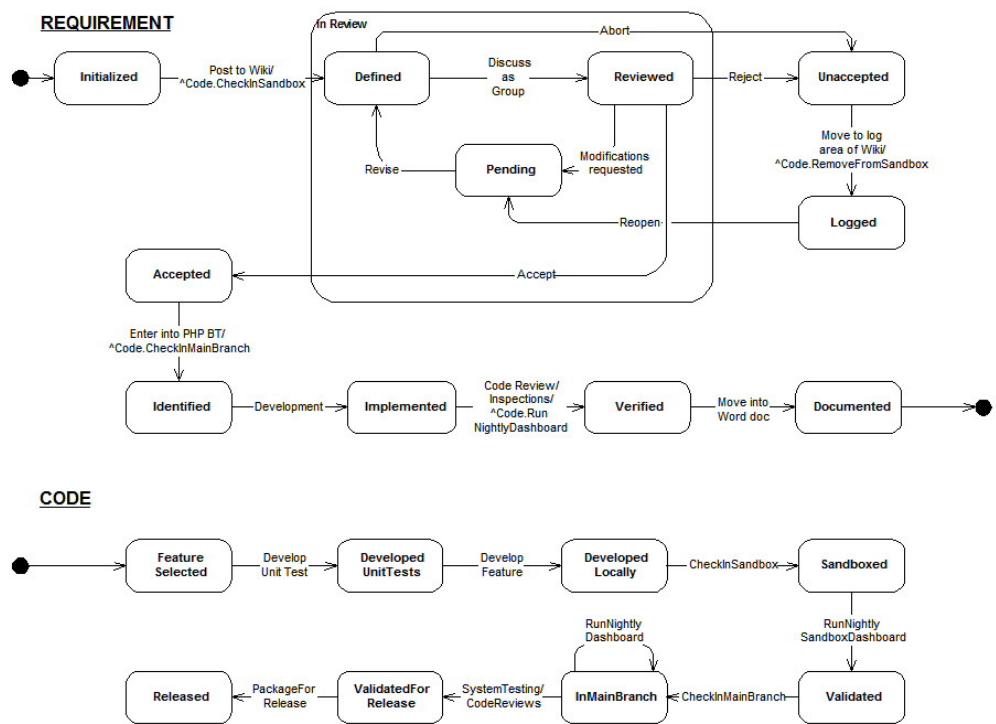


Figure 4.2: Lightweight Requirements Management Process in IGSTK.

Table 4.1: Sample Scenario for Guidewire 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 patient's 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 3D 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. |

4.3 Conceptualizing Application Requirements through Activity Modeling

A barrier to conceptualizing requirements is the disparity of knowledge between software engineers and clinicians. These two groups 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 4.1 shows a sample scenario of the use of IGSTK applications for guidewire placement. A stepwise, temporal review of this scenario is illustrated in Figure 4.3. This example is presented in detail with its initial implementation in Chapter 25.

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 addition, requirements can be inferred from this activity diagram for all

three requirements classifications as defined in the earlier section.

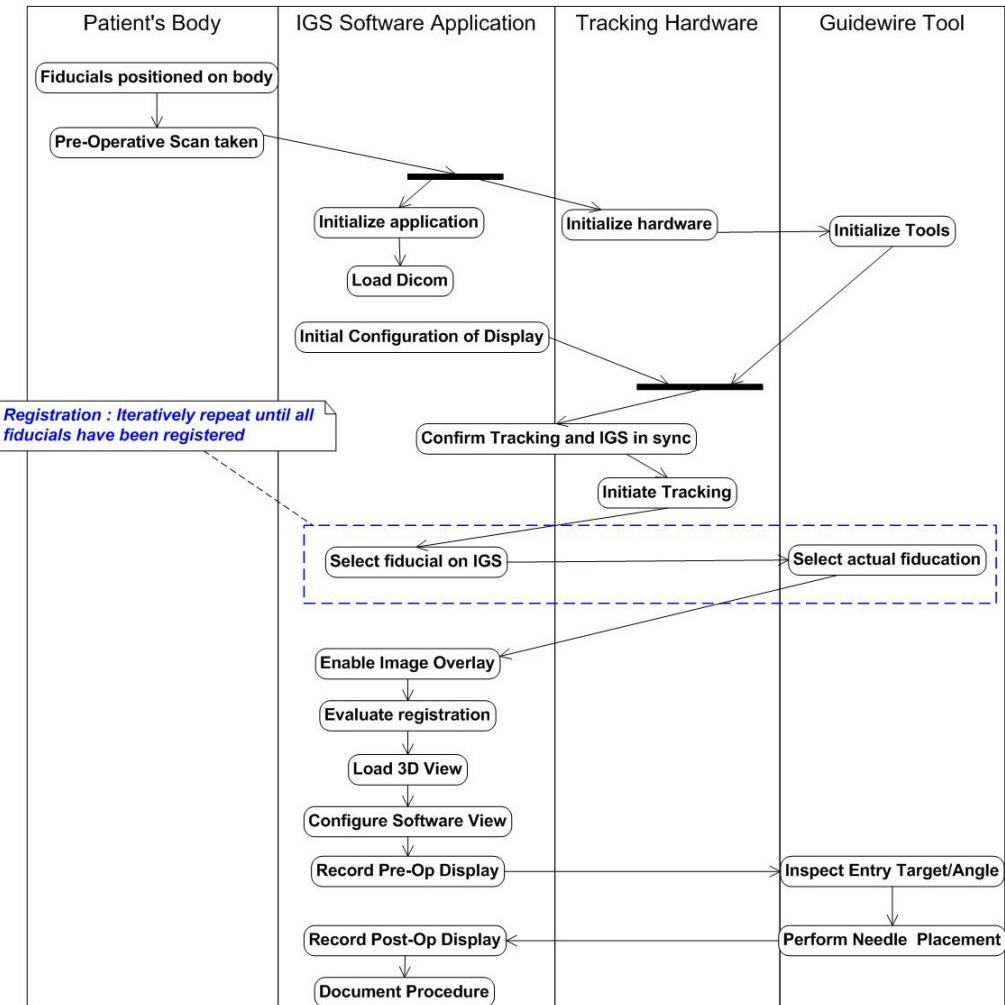


Figure 4.3: Guidewire Tracking Activity Diagram.

4.4 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.

The original IGSTK requirements were captured in the IGSTK BugTracker. Figure 4.4 presents a query page for filtering information from the bug tracker.

The screenshot shows the Mantis BugTracker interface. At the top, there is a logo of a green mantis and the word "MANTIS". Below the logo, there are links for "Anonymous", "Login", and "Signup for a new account". The date and time are shown as "2009-05-21 16:07 EDT". The project name is "IGSTK". There are buttons for "Switch" and "Logout". Below the header, there is a navigation bar with links: "Main", "My View", "View Issues", "Change Log", "Roadmap", and "Docs". There are also buttons for "Issue #", "Jump", "Search", "Apply Filter", "[Simple Filters]", and "[Create Permalink]".

Below the navigation bar is a search form with fields for "Reporter", "Monitored By:", "Assigned To:", "Category", "Severity", "Resolution", and "Profile". There are dropdown menus for "Status", "Product Build", "Priority", and "Target Version". Other fields include "Show", "View Status", "Show Sticky Issues", "Changed(hrs)", "Use Date Filters", and "Relationships".

The main area displays a table titled "Viewing Issues (1 - 50 / 555) [Print Reports] [CSV Export]". The table has columns: P, ID, #, Category, Severity, Status, Updated, and Summary. The rows show various issues, such as:

P	ID	#	Category	Severity	Status	Updated	Summary
	0009048		IGSTK	crash	assigned (cheng)	2009-05-18	Navigator example crashes when the scout image is the first series in the directory
	0009045	1	IGSTK	minor	closed (andinet.enqu)	2009-05-18	IGSTK dashboard configuration scripts outdated
	0009001		IGSTK	minor	resolved (andinet.enqu)	2009-05-11	Missing classes in StateMachineExportTest
	0008936		IGSTK	minor	resolved (cheng)	2009-04-27	uninitialized calibration transformation
	0008935		IGSTK	minor	resolved (cheng)	2009-04-27	igstkTrackerController used the flag IGSTKSandbox_USE_MicronTracker instead of IGSTK_USE_MicronTracker
	0008921		IGSTK	minor	resolved (zivy)	2009-04-22	no error message available when attaching tool fails in the tracker controller
	0008912	2	IGSTK	major	closed (cheng)	2009-04-20	Confusion with igstkConfigure.h and igstkSystemInformation.h

Figure 4.4: Requirements Management using Mantis BugTracker. Customized states and transitions are scripted into the BugTracker to enable developers and the user community easy, consistent access to defects and changing requirements.

Part II

Framework Design

Architecture

“As to diseases, make a habit of two things - to help, or at least, to do no harm.”

—Hippocrates, Epidemics Book 1, Section XI.

5.1 General Background

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 take advantage of every possible software mechanism that could prevent errors when the application was 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 used

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 it possible to guarantee that the application is never in an error state, and they make explicit their behavior 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 [3, 5].

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 thoroughly. 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 thoroughly tested [21].

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 it possible to have 100 percent 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 features 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 to offer options to application developers. This tendency has to be reversed in the context of 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 3.

Further motivation for the emphasis on safety came from the development team's awareness that clinical conditions are already tense, even before software-based systems are 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 background of what is publicly accepted as the evidence of continuous danger in a clinical environment, and what is known about the frailties and vulnerabilities of software development endeavors.

5.2 Medical Errors

The following is a quote from the report “*To Err is Human*” prepared by the U.S. Institute of Medicine¹ in 2001 [17].

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 errors 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 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 economic terms was kept in mind during the design and development process of the toolkit. It became relevant every time arguments were raised in favor of relaxing the safety rules of the toolkit with the purpose of making the work of developers easier or more convenient. 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.

¹<http://www.iom.edu>

In the analysis of “*Why errors happen?*” the report emphasizes the following:

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 be perceived as a weakness in a general software toolkit, are indeed the strength of IGSTK because it is then more likely that the toolkit will be safe and reliable enough for clinical settings.

The 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 more common than one might think.
- Medical errors are sometimes irreversible.
- Medical errors are expensive.
- Good design practices can help prevent medical errors.

5.3 Layered Architecture

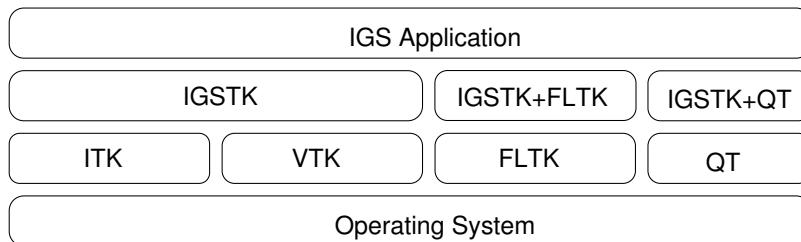


Figure 5.1: IGSTK Layered Architecture.

Figure 5.1 illustrates the layered architecture of IGSTK, starting from the operating system layer as a foundation. IGSTK is built on other widely used open source toolkits. The Insight Toolkit

(ITK) is used for providing all the image analysis functionalities as well as a good portion of the infrastructure classes. The Visualization Toolkit (VTK) is used for supporting the display of images and geometrical models as well as managing a good portion of the user interaction. The Fast Light Toolkit FLTK is one option for the GUI. 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.

Image-Guided Surgery (IGS) applications will be built on top of IGSTK. The application code can only have access to IGSTK classes, not to the ITK or VTK classes that are used underneath. IGSTK internally makes use of ITK and VTK classes but does not expose any of them in its API. The purpose of this encapsulation of toolkit functionalities is to improve the safety of the final application by preventing developers from directly manipulating objects without passing first through the many safeguards that have been included in IGSTK.

5.4 The Main Components

The toolkit was designed as a collection of medium size components. Each one of these components was implemented as a C++ class. In many cases an IGSTK class contains an aggregation of ITK and VTK classes that are configured for solving a specific task in image-guided surgery.

The boundaries between two IGSTK components were defined by exploring the number and significance of interactions between the internal ITK and VTK classes of both components. The general guideline was to define the boundary in places where minimal interactions were needed between two IGSTK components.

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 safety by design approach in IGSTK. The idea is that derived classes allow one 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 [21].

A typical example of this implementation of safety by design is the family of the `ImageReader` classes. While an abstract `DICOMImageReader` class implements the basic actions required for reading any DICOM files, its derived classes the `CTImageReader` and `MRIImageReader` 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 does not match the expected modality, the reader will report the failure of the action and it will remain in a safe state declaring that no image has been read.

As illustrated in Figure 5.2, the components of the toolkit can be classified in the following groups

- Display
- Geometric Representation

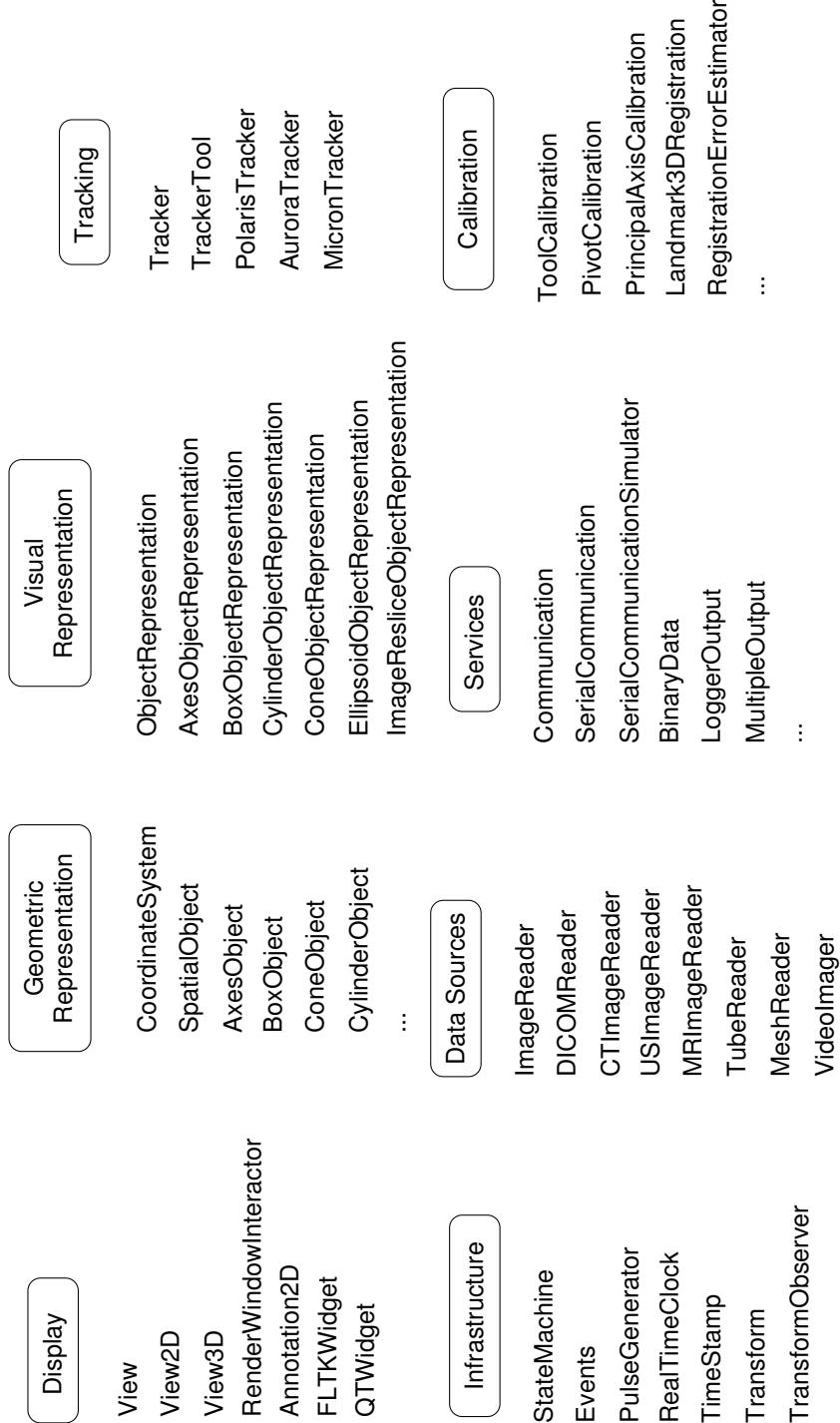


Figure 5.2: Categories of the Main IGSTK Components.

- Visual Representation
- Tracking
- Data Sources
- Calibration
- Infrastructure
- Services

These categories are described in the following sections.

5.4.1 Display

The Display category includes the components that are closest to the application's GUI. These are the classes that will be displaying renderings of the surgical scene inside one or more specific windows of the GUI. The main classes in this group are listed below.

- `igstk::View`
- `igstk::View2D`
- `igstk::View3D`
- `igstk::Annotation2D`
- `igstk::RenderWindowInteractor`
- `igstk::FLTKWidget`
- `igstk::QWidget`

The interactions between IGSTK and the applications's GUI happen at the level of visualization windows. They are typically based on OpenGL-driven windows that are created by the GUI library and then are offered to VTK classes in order to use them as the output for the generated renderings.

The classes that abstract the interface to the GUI library are at the bottom of the list. They are the `FLTKWidget` and `QWidget`. They are intended to be used along with the FLTK and Qt libraries respectively. These Widget classes connect the OpenGL window created by the GUI library to the `vtkRenderWindowInteractor` class that translates GUI events, typically Mouse and Keyboard events, into VTK internal events. The Widget classes are the only ones in this group that are aware of the GUI's existence. By abstracting the OpenGL window and the user interaction events, the Widget classes make it possible to encapsulate all the GUI dependencies away from the rest of IGSTK classes intended for display. These details, however, are hidden

to the application developer. From the point of view of application design, the developer should simply anticipate when and where to show the instances of the Widget classes.

Developers who want to use IGSTK with a new GUI library (different from FLTK and Qt) should start by creating a new customized Widget class that will interface to the implementation of an OpenGL window in that GUI library.

The central class in this Display group is the `View` class. It encapsulates the VTK classes that produce the actual renderings. The `View` contains instances of the `vtkRenderer`, `vtkRenderWindow` and `igstk::RenderWindowInteractor` classes. An instance of the `View` class is intended to be connected to one and only one instance of the Widget classes. The `View` class is further specialized into two subclasses, the `View2D` class and the `View3D` class. They differ on the type of user interactions that they allow. These classes are described in detail in Chapter 12.

The renderings generated by `View` classes are refreshed at a rate set by the application developer. Each time that a refresh request is triggered, the request is propagated to other scene representation classes with the purpose of composing a scene that is consistent for a given point in time.

The `RenderWindowInteractor` class is a wrapping around the VTK class of the same name. This class encapsulates the services of the VTK class and facilitates the integration of the `View` class with a particular GUI library.

The `Annotation2D` class is intended to display messages on the `View` windows. A typical usage is to display the patient name in one of the window corners.

5.4.2 Geometric Representation

The surgical scene is represented at two levels. On one hand the geometrical aspects of scene objects are managed by the `SpatialObject` classes. On the other hand, the visual representation aspects of these objects are managed by the `Representation` classes.

The family of `SpatialObjects` include the following classes.

- `igstk::SpatialObject`
- `igstk::GroupObject`
- `igstk::AxesObject`
- `igstk::BoxObject`
- `igstk::ConeObject`
- `igstk::CylinderObject`
- `igstk::EllipsoidObject`
- `igstk::ImageSpatialObject`

- `igstk::CTImageSpatialObject`
- `igstk::MRISpatialObject`
- `igstk::USISpatialObject`
- `igstk::MeshObject`
- `igstk::TubeObject`
- `igstk::TubeGroupObject`
- `igstk::UltrasoundProbeObject`
- `igstk::ReslicerPlaneSpatialObject`
- `igstk::CoordinateSystem`

The classes in this group define the shape and size characteristics of physical objects that are commonly present in a surgical scenario. These classes are intended to be used as basic elements for composing a scene. For example, a tumor biopsy application could use the `CylinderObject` for representing a surgical needle, and an `EllipsoidObject` for representing a tumor. A liver resection application could use the `MeshObject` to display the segmentation results of different liver lobes. The `TubeGroupObject` could typically be used for representing a vascular tree or the airways.

The `ImageSpatialObject` family of classes hold the image data from CT, MRI and Ultra sound scans. This 3D data is commonly displayed by extracting a 2D slice from the dataset. The `ReslicerPlaneSpatialObject` encapsulates the information describing the position and orientation of the plane used to extract that slice.

This group includes basic geometrical shapes such as boxes, cones and ellipsoids, as well as more complex shapes such as vascular trees, ultrasound probes, and surfaces resulting from the segmentation of anatomical structures. These are the main objects that will be presented to the clinician as elements of the surgical scene. It is anticipated that particular image-guided surgery applications will add more complex and specialized objects for specific clinical applications.

To define the position and orientation of all these objects in 3D space, an implicit *Scene Graph* structure is provided by using instances of the `igstk::CoordinateSystem` class as nodes. Spatial objects, Trackers, Tracker Tools and Views contain an internal instance of the coordinate system class, and in this way they can be associated to the nodes of the scene graph structure. The relative position and orientations of connected nodes in this graph are defined by `igstk::Transforms`. When writing an IGS application, one of the first considerations of the developers should be to identify the proper scene graph that represents the relationships between the objects present in the surgical scene. When defined properly, the scene graph provides a reliable platform for consistently representing the relative positions and orientations of all the objects, regardless of whether they are being tracked or not.

Except for the `CoordinateSystem`, all the classes in this group derive from the `SpatialObject` class and have been constructed as safe encapsulations around the `itk::SpatialObject` classes. The classes listed in this group are described in detail in Chapter 10.

5.4.3 Visual Representation

The classes in the Geometric Representation group do not specify the visual aspects of how they will be presented in the scene. They focus on the size, shape and position aspect of these objects.

The visual characteristics of the object are managed by the visual representation classes listed below.

- `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`
- `igstk::ImageResliceSpatialObjectRepresentation`

These classes specify properties such as color and reflectivity of surfaces, the thickness of lines, and the resolution of discretization for cones and spheres.

The classes in this group are usually associated one-to-one to the classes in the geometrical representation group. However, it is always possible for a single geometrical class to be represented by different visual classes. For example, a geometrical CylinderObject could be represented by the actual surface rendering of cylindrical shape, or it could also be simplified as a line on the screen.

Most of the classes in the family of the ImageSpatialObjectRepresentation can extract orthogonal slices from 3D datasets for the purpose of displaying them as planes in space. The ImageResliceSpatialObjectRepresentation class has the particularity of extracting 2D slices at arbitrary positions and orientations, a functionality that is commonly used for displaying an anatomical context for tracked instruments that have been inserted in the patient.

It is up to the application developers, in collaboration with their clinical partners, to explore the most effective ways of representing the element of the surgical scene. Just as with the previous group, here we also anticipate that specialized applications will develop specific variants of the representation classes that will be customized to the needs of a clinical application. These classes are described in detail in Chapter 11.

5.4.4 Tracking

Despite the fact that we refer to this domain as *image-guided* surgery, there is indeed a very large fraction of the *guidance* that is provided by devices that track the position and orientation of surgical instruments and anatomical structures in 3D space. These devices, known as *Trackers*, are typically based on optical or electromagnetic principles.

Image-guided surgery applications require support for incorporating information provided by tracking devices. IGSTK provides this support in the form of classes that represent the abstraction of the physical tracker device. The main members of this category are the `Tracker` and `TrackerTool` classes. The `Tracker` class is an abstract class that embodies the interface of a physical tracking device and its interactions with a software application. This class is then specialized for brand and model-specific variations of the Trackers. Many commercial tracking devices use the concept of *Tools* as an abstraction of the different physical objects that they can track. A single tracker, for example, can gather information corresponding to three or four different objects, by associating a tool to each one of them. The `TrackerTool` is used in IGSTK for mapping this concept of the physical trackers into the software level.

The main classes in this category are presented in the following list.

- `igstk::Tracker`
- `igstk::TrackerTool`
- `igstk::NDITracker`
- `igstk::PolarisTracker`
- `igstk::PolarisTrackerTool`
- `igstk::AuroraTracker`
- `igstk::AuroraTrackerTool`
- `igstk::MicronTracker`
- `igstk::MicronTrackerTool`
- `igstk::AscensionTracker`
- `igstk::AscensionTrackerTool`

We anticipate that as more tracking devices become available in the medical market, additional tracker classes will be added to the toolkit to facilitate their use from IGS applications. The design of the Tracker classes allows one to easily integrate new variations of trackers, just by creating a new class deriving from the base Tracker class. This evolution is illustrated by the recent addition of the MicronTracker² and the AscensionTracker³.

The Tracker classes are described in detail in Chapter 9.

5.4.5 Data Sources

The category of data sources encompasses all the classes that are used for bringing data into the scene representation. The major subgroups of the data sources are image readers, mesh readers, and transformation readers. The full list is presented below.

- `igstk::DICOMImageReader`
- `igstk::CTImageReader`
- `igstk::GenericImageReader`
- `igstk::MRIImageReader`
- `igstk::MeshReader`
- `igstk::TransformFileReader`
- `igstk::SpatialObjectReader`
- `igstk::TubeReader`
- `igstk::USImageReader`
- `igstk::VascularNetworkReader`
- `igstk::VideoImager`

One of the most important classes is the `DICOMReader`. This will be used for reading images from preoperative scans of the patient. The `DICOMImageReader` is further specialized in three classes, the `CTImageReader`, `MRIImageReader` and the `USImageReader`. Each one of these classes imposes a certain number of restrictions intended to mitigate the risk of reading an incorrect image during the execution of the surgical intervention. The image reader classes are described in detail in Chapter 14.

Additional readers are available for loading segmentations of anatomical structures in the form of surfaces and tree structures. It is also possible to read information related to the calibration of

²Claron Technologies (<http://www.clarontech.com>).

³Ascension Technology Corporation (<http://www.ascension-tech.com>).

tracking devices with respect to references in the patient, for example, pre-operatively inserted fiducials.

The GenericImageReader⁴ makes it possible to load images from any of the file formats supported by ITK. This reader is provided mostly for debugging purposes and for prototyping applications at research level. It is strongly discouraged to use this reader in an IGS application intended to be deployed in a clinical setting.

The VideoImager is analogous to a type of reader that gathers image information via a digitization card or from actual digital acquisition devices such as fluoroscopy, endoscopy, and ultrasound machines.

5.4.6 Calibration and Registration

One of the typical tasks of image-guided surgery interventions is to combine information from pre-operative images with data from intra-operative tracking of surgical instruments. To combine these two sources of information in an effective and safe manner it is necessary to determine the geometrical transformation that relates the coordinate system of the pre-operative image with the coordinate system of the tracking device. This is commonly done by performing registration between points taken from the image and positions read from the tracker.

The classes in this category are used for computing such registrations, as well as for computing the relationships between the tips of tracker tools and the coordinate system of the tracking device itself. The classes of this category are listed below.

- `igstk::PivotCalibration`
- `igstk::Landmark3DRegistration`
- `igstk::Landmark3DRegistrationErrorEstimation`

The classes in this group are described in detail in Chapter 15 and Chapter 16.

5.4.7 Infrastructure

The category of infrastructure aggregates classes that are used inside many other IGSTK components. These infrastructure classes provide the supporting fabric that allows for the coordination of activities between different IGSTK components. The main classes of this category are listed below.

- `igstk::StateMachine`
- `igstk::Events`

⁴This class is under development and is not included in this release.

- `igstk::PulseGenerator`
- `igstk::RealTimeClock`
- `igstk::TimeStamp`
- `igstk::Transform`
- `igstk::TransformObserver`

IGSTK components have been designed to have a high level of autonomy. Although, of course, they do not operate independently, they still have a high degree of self-sufficiency when it comes to executing their tasks.

The RealTimeClock and PulseGenerator classes are intended to provide self-generated time ticks to the classes that need to ensure their execution at a specific rate. In particular, this is done in the View and Tracker classes.

The Event classes provide a common layer of communications that facilitate the decoupling of different IGSTK components and still allow them to effectively work together. Events are described in detail in Chapter 7.

The StateMachine class provides a formal framework for programming the logic that governs the behavior of individual IGSTK classes. They add determinism, safety, and higher levels of encapsulation to the IGSTK components. State machines are described in detail in Chapter 6.

The Transform class is used to represent the relative positions and orientations of nodes in the surgical scene. All transforms in IGSTK are currently 3D rigid transforms. That is, they only represent combinations of rotations and translations in 3D space. The rationale for limiting the transform to this mathematical group is that most of the time in image-guided surgery applications, the focus is on coordinating the positions of different objects in 3D space. For example, displaying the position of surgical instrument as reported by a tracking device, with respect to the position of a organ segmented from a pre-operative image.

Given that it is very common to pass Transforms as the payload of Events, it is convenient to have a class that can be used as an observer for such events. This class is the TransformObserver. It provides an easy-to-use mechanism for receiving events and extracting the transforms that they can potentially carry as payload.

5.4.8 Services

This category contains all the classes that provide services to other IGSTK components. The major classes of this group are listed below.

- `igstk::Communication`
- `igstk::SerialCommunication`
- `igstk::SerialCommunicationSimulator`

- `igstk::SocketCommunication`
- `igstk::TrackerToolObserverToSocketRelay`
- `igstk::TrackerToolObserverToOpenIGTLLinkRelay`
- `igstk::BinaryData`
- `igstk::LoggerOutput`
- `igstk::MultipleOutput`

Most of the classes in this group were the result of identifying large sections of code in existing IGSTK components that were performing complex tasks and for which a reduced number of interactions were needed. For example, the PolarisTracker and AuroraTracker classes required information to be sent through the serial port to communicate with the tracking device. It was therefore convenient to remove from these trackers any code implementing serial port communications, and to encapsulate it into a service class that could be used from the trackers.

A similar situation arose with the `SocketCommunication`, which happened to be necessary for interfacing to a robotic device via TCP/IP sockets. By encapsulating in this class the functionalities of opening, closing, sending and receiving data through sockets, it was possible to reuse this functionality outside of the framework of this robotic device.

One of the most important subgroups of classes in this category is the set of Logging classes. These classes facilitate the systematic gathering of information from IGSTK components as they are executed at run time. Information sent to the log files can be used for debugging purposes, performance profiling, retrospective optimization, reconstruction of a previously executed surgical intervention and for animating recreations of the state machine executions. The logging classes are described in detail in Chapter 13.

The `TrackerToolObserver-relay` classes are intended to be used for implementing the OpenIGTLLink⁵ communications protocol. These classes listen to Transform Events produced by a `TrackerTool` and reformat the Transform information according to the encoding defined in the OpenIGTLLink standard⁶. This makes it possible for IGSTK applications to communicate with other applications running in different computers in the network, and to even make IGSTK applications collaborate with others that are not based on IGSTK. For example, by setting up an IGSTK application to be a Tracker-server while having a visualization application acting as a client.

5.5 Timing

*In action, timing is everything.
Force doesn't matter.*

⁵<http://www.na-mic.org/Wiki/index.php/OpenIGTLLink>

⁶<http://www.na-mic.org/Wiki/index.php/OpenIGTLLink/Protocol>

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

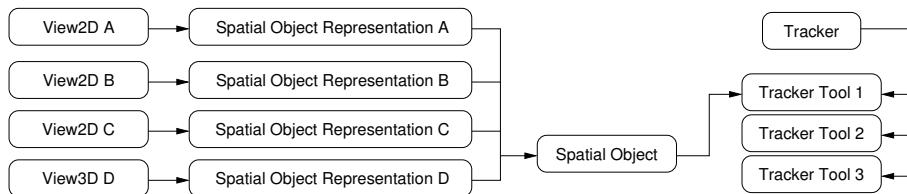


Figure 5.3: Timing Collaboration Between the Main IGSTK Components.

5.5.1 Timing Collaborations

When a display is presented to the surgeon, it is implicitly stated that it is the view of the operating room at a very recent time. Since many of the objects in the scene are in continuous movement, 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 9:06 am. The toolkit design takes measures for preventing the accidental display of the position of the needle where it was at 9:06 am along with the patient's liver where it was at 8:54 am. 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 management of synchronism and timing in IGSTK is an integral part of the architectural design. The timing aspects of collaboration between the main component of the toolkit are illustrated in Figure 5.3. Tracker and View classes have their own pulse generators that will keep them updating at a rate specified by the application developer. These may well be different rates. Presumably the tracker will use higher rates than the instantiations of the view class, but this may change depending on the specific clinical application at hand.

A tracking device can usually drive multiple tracked tools that, in turn, are associated to different surgical instruments or markers in the patient. The updated information of position and orientation of tracked objects is carried through the application in the form of transforms. The

transforms originate in the tracker class under the control of continuous updates regulated by the pulse generator. Transforms produced by the tracker device are specific to each one of the tracked objects and therefore are stored in the associated instantiations of the tracker tool classes. Once information about transform updates is passed from the tracker into the tracker tool, it becomes available to be queried by the spatial objects. This querying will typically happen at the time that a view requests its representation objects to update their appearance. At that point the associated spatial object will compute their current position and orientation with respect to the View that originated the update. The computation of relative position and orientation is done internally by the CoordinateSystem classes that implement the scene graph.

Objects that are tracked in the surgical scene should be rendered in the display to be presented to the surgeon. This visual display is performed by the spatial object representation classes in association with the spatial object classes. One spatial object instance can be attached to one and only one tracker tool instance. A TrackerTool can only be attached to a single SpatialObject.

Figure 5.4 presents a UML sequence diagram of the timing interactions between the major classes in IGSTK. The diagram can be better understood by dividing it in two sections. The division line must be drawn between the SpatialObject and the TrackerTool. The section to the right of the SpatialObject describes the flow of information and events from the PulseGenerator class that is associated with a Tracker, up to the instance of the TrackerTool 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. This is the pulse generator that is contained inside the Tracker.

The application developer defines the reading rate of the tracker by setting the rate of operation of the pulse generator. At every pulse, the Tracker class will 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 uses the TrackerTool instances as the helpers that are associated to each one of the tracked surgical instruments. When the Tracker class receives the updated information from the hardware, it stores the new transforms, which 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 `RequestUpdateStatus()` in the Tracker. 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. At that point, the TrackerTool is updated and the refreshing cycle of the Tracker concludes. Events are further described in Chapter 7. The detailed description of the Tracker is presented in Chapter 9.

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

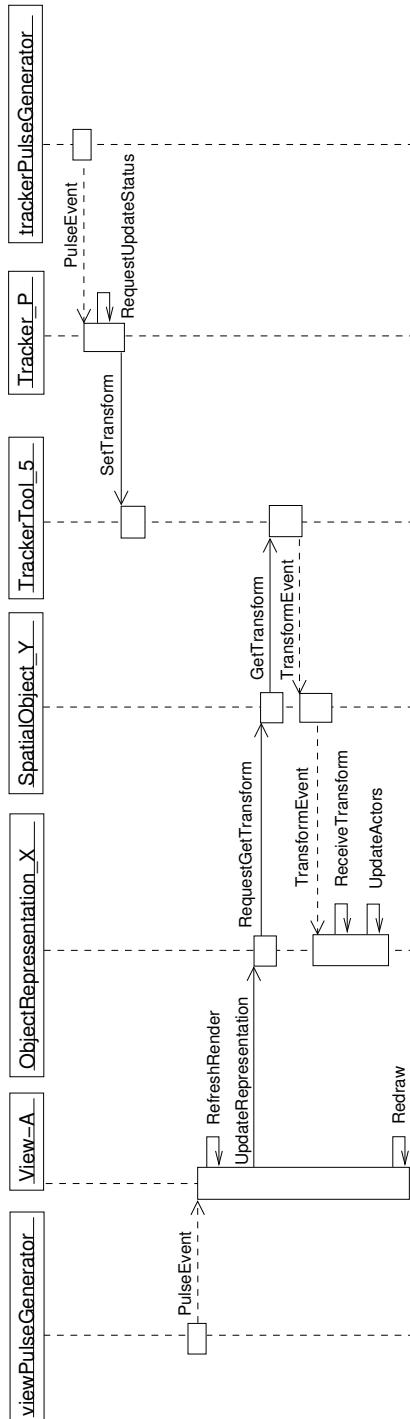


Figure 5.4: Timing Collaboration Between the Main IGSTK Components.

should be available and when to stop using an old transform. A very similar mechanism is used in the TCP/IP protocol 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 not to 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 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. In either case, it is very important to let the surgeon know that the position of that particular object in the display can not be trusted.

The section of the diagram at the left side of the TrackerTool represents another cycle of information refreshing, this time in two stages. First, requests 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 method `RequestUpdateRepresentation()`. During that invocation, the View class notifies these objects that it is scheduling a refreshing at a specific time in the future. The representations and their associated spatial objects will compare this time to the validity range of their transforms to verify that the transforms will not be expired at the time associated with the rendered scene.

Note that, for the sake of simplicity, only the cycle of the `RequestUpdateRepresentation()` method is presented in this diagram. When the `RequestUpdateRepresentation` method is invoked in the SpatialObjectRepresentation object, it triggers a call to the `RequestComputeTransformTo()` method on the SpatialObject. Typically, the argument of this call will be the View that is being refreshed. This in its turn triggers a call to the methods of the CoordinateSystem nodes in the scene graph that will compose all the Transforms between this spatial object and the View that originated the refresh call. If a valid Transform results from the composition, it will be sent via an Event to the SpatialObjectRepresentation, that will receive it by using an Observer defined by a transduction macro.

The reception of the TransformEvent in the SpatialObjectRepresentation triggers the `ReceiveSpatialObjectTransformProcessing()` method, that then updates the transform of all its VTK actors. This last method also calls the `RequestVerifyTimeStampAndUpdateVisibility` method to modify the graphical representation of the objects as presented to the surgeon, 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 SpatialOb-

jectRepresentation objects, it will proceed to actually redraw the scene by triggering a `Render()` call in its internal `vtkRenderWindowInteractor`. This will trigger a Render action on the VTK pipeline that is maintained by classes located inside the View and the `SpatialObjectRepresentation` classes.

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).

The timing infrastructure described here 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.

5.5.2 Pulse Generator Implementation

Figure 5.5 illustrates the State Machine of the `igstk::PulseGenerator` class. This state machine is set up as a simple oscillator. The pulse generator, when active, is alternating between the `PulsingState` and the `WaitingEventReturnState`. The actions corresponding to the transitions out of each one of these two states take care of inserting an input in the state machine queue, that will trigger the next alternation of states. The cycle can be broken by the reception of a stop input pushed via the `RequestStop()` method.

The pulse generator relies on an internally maintained queue of timeout events. As a consequence, it is not possible to guarantee that the rates requested by the application developer could actually be honored in practice. It is therefore extremely important for application developers to verify the actual rate at which their class configurations manage to perform at run time. Several iterations of fine tuning may be required to find rates that produce an appropriate response time for the needs of the specific surgical intervention.

The refresh rates of View classes must be selected according to the capabilities of the graphics hardware available, the complexity of the scene to be rendered, and the actual rate of interaction required by the clinician.

The refresh rate of the Tracker classes must be selected according to the actual tracker hardware rate of refreshing position information, the rate of transmission set on the communication layer(for instance, the baud rate of the serial link), and the purpose of displaying the tracked object in the surgical scene.

5.6 Conclusion

The architecture of the IGSTK toolkit was designed based on a concern for robustness and safety. The code was organized in medium-size components with a reduced set of features.

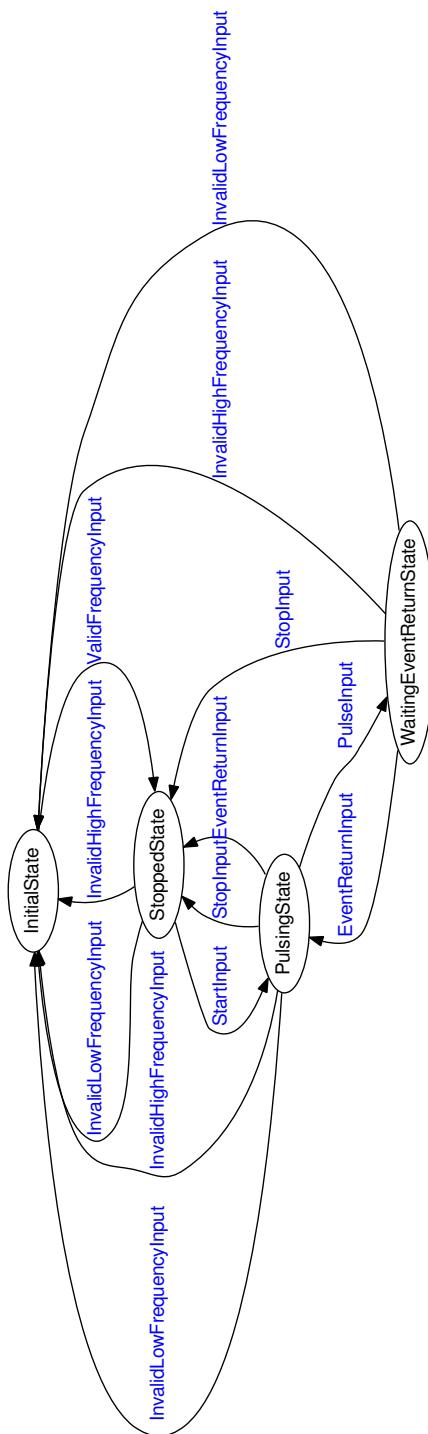


Figure 5.5: State Machine Diagram of the PulseGenerator Class.

Strong encapsulation and C++ typing was used to further reduce opportunities for components misuse. State machines were introduced to closely control the logic of each component and to enforce determinism and manage error conditions. Communications between components were implemented using the event observer pattern.

Although the development team went to great lengths to make the toolkit a very robust piece of software, there are always unforeseeable situations that may require the refactoring of components in the future. The combination of testing-base agile development and requirements-driven development used in our software development process will facilitate such refactorings with an appropriate level of quality.

State Machine

“We may regard the present state of the universe as the effect of its past and the cause of its future. An intellect which at a certain moment would know all forces that set nature in motion, and all positions of all items of which nature is composed, if this intellect were also vast enough to submit these data to analysis, it would embrace in a single formula the movements of the greatest bodies of the universe and those of the tiniest atom; for such an intellect nothing would be uncertain and the future just like the past would be present before its eyes.”

“—*Essai Philosophique sur les Probabilités*,” 1810, Pierre-Simon Laplace

6.1 General Background

The *State machine* is a fundamental concept in computer programming. It was introduced by Alan Turing in 1936 as a formalism for supporting his work on determining whether the execution of an algorithm will ever stop; a theoretical question also known as the *Entscheidungsproblem*. 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 lead to a unique state [5, 18].

All computers can be modeled as state machines; however their number of possible states is so large that, for practical purposes, they can barely be considered 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; 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, and unfortunately, the one that better describes 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 thus to protect patients from harm.

Computer programs, in particular those that are modeled using object-oriented programming, are naturally 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, in which 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 necessity of reliability and robustness in IGSTK led the development team at the very early stages of the project to decide to use the state machine model. State machines are an excellent way of limiting the number of possible behaviors and ensuring both that a program will always be in a valid condition, and that all possible behaviors have been considered in advance by the developers team to anticipate appropriate responses. A well-defined architecture based on state machines makes it possible to guarantee repeatability and deterministic behavior [3, 13].

The introduction of state machines makes it 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

A computer program is said to display *Deterministic Behavior* if it provides exactly the same response every time that it is fed with a given specific input and under the same conditions. The main reason why determinism is a desirable feature of software is that it gives meaning to the effort of intensively testing the software. In the absence of deterministic behavior, the process of software testing would become pointless because non-deterministic programs could behave fine during the testing process and could still fail during the execution of a surgical intervention. The whole purpose of performing testing is to build confidence in the quality of the software, and to expect that if the software behaves correctly during the testing process, it will do just as well when it is used for guiding the execution of a surgical procedure.

6.2.2 Preclude Wrong Use

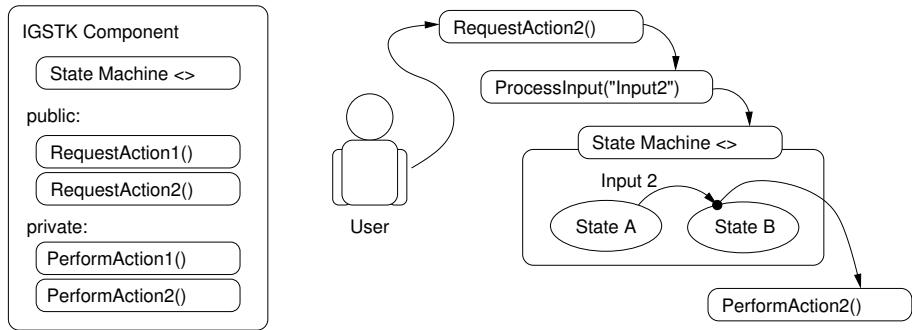


Figure 6.1: Illustration of the Separation Between Public and Private Interface of IGSTK Components. Every IGSTK component based on a state machine contains an instance of the state machine class. To invoke actions, users have access to public methods that *request* the IGSTK component to perform an action. The request is passed to the state machine, where, depending on the current state of the component, the request may or may not be satisfied. The methods that actually perform actions are all declared private and can only be called by the state machine. In this way, the state machine logic provides a layer of protection that separates the user's request from the actual execution of actions. The fact that the logic of the state machine verifies that the conditions for performing an action are appropriate before triggering such execution prevents the incorrect usage of the IGSTK components. Note that the action method is called after the state machine assumes the destination state.

The introduction of state machines in IGSTK makes it possible to implement 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 to decide 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 untold assumptions regarding the order and the conditions in which they should be invoked. Developers, being aware of such untold assumptions and conditions, tend to test the classes by following a set of non-explicit rules to make sure that the conditions are satisfied. Users 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. In this traditional context, when a user of the software invokes any of the public methods, the class obliges blindly to the request. This blind obedience makes the software fragile to users that are interacting naively with the system.

The introduction of a state machine separates user requests from the actions executed by the class. This means that instead of blindly responding to every invocation of a method, now a class simply takes a function call as a “*request*” from the user. This request is translated into an input to the state machine of the class, and depending on its 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 satisfy the necessary conditions, then the state machine may reply with an error notification via an Event (see Chapter 7), or may simply ignore the request. In either case, the class will be left in a valid state after returning from considering the user's request.

Figure 6.1 illustrates the separation between the public and private interface of an IGSTK component. 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 prevents users from using the class incorrectly. This diagram also illustrates the fact that the action method is called after setting the state machine to the destination state. This means that at the time the action method is being executed, the state machine is already in the destination state. For the cases where the successful termination of the action method is a requirement for reaching a destination state, IGSTK implements the “*Attempting*” pattern, as described in Section 6.3.2.

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 way to protect 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, if 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 will the state machine go into an erroneous state.

Note that this property does not arise automatically from the introduction of the state machine inside the IGSTK component. Instead, it requires the developers of the IGSTK class to consider all possible combinations of states and inputs, and to anticipate the appropriate responses for each case. In this context, the state machine is simply a convenient recipient of the logic defined by the developers. The relevance of the state machine for providing this property is lost if developers skip the full analysis of inputs and states combinations. Unfortunately, developers that are not properly trained tend to underestimate the importance of considering all combinations of situations in the state machine, and tend to code only for the trivial *expected* cases that represent good scenarios. Such lax practices result in fragile code that will break at execution time as soon as it is exposed to any of the non-standard situations that were not considered by the developers. The IGSTK software development best practices described in Section 3.1 require developers to define responses to all inputs from all states in classes that use a state machine.

6.2.4 Managing Complexity

Very early in the development cycle it became clear that simplicity of the classes API is fundamental for promoting 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 things to go wrong during the surgical intervention.

The introduction of state machines makes complexity management possible by forcing developers to make the logic of the application 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. When state machines are not used, it is easy for programmers to deny or to ignore the existence of these details because they are spread across the code in the form of *latent logic*, implemented in hundreds of “if” statements as well as in the conditionals of “for,” “while,” and “switch” statements. This diffusion of the application’s logic produces the false impression that the source code is simple and that all the important combinations of situations are being managed even if they are not. State machines force developers to concentrate the logic of a component in a single place within the class, where it is made explicit and it is clearly visible. At that point it becomes possible to apply a realistic approach for managing complexity 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 algorithmic complexity contained in a particular class.

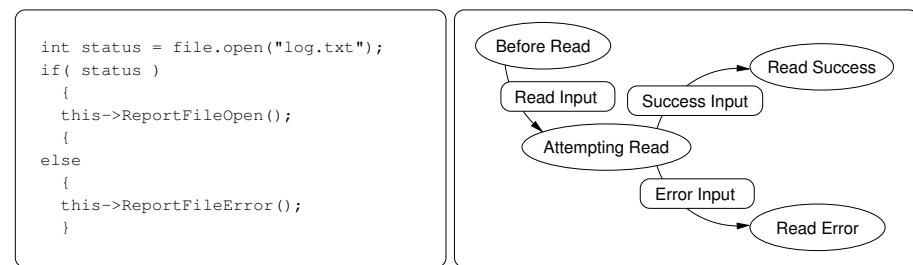


Figure 6.2: Comparison of How a Traditional “if” Statement is Implemented Using a State Machine in IGSTK. The source code on the left is typically used for attempting to open a file, and depending on the success or failure of that operation, to execute a different action. The state diagram on the right illustrates the set of four states and three inputs that correspond to the logic of this operation.

Figure 6.2 illustrates the case of a commonly used “if” statement. Implementing this code as a state machine using IGSTK guidelines will require a set of four states and three associated inputs. The first state “Before Read” is the one existing before we call the `open()` method. The “Read Input” is the signal that triggers the action of opening the file. During the time that the IO operations are being performed, the program is set in an “Attempting Read” state.

At some point, the IO operation will return with a status value indicating success or failure. The two possible values of the status variable correspond to two possible inputs to the state machine, “Success Input” and “Error Input.” Each one of those inputs will produce a transition to a different state, “Read Success” or “Read Error,” and as a consequence will also trigger the execution of a different action for each case.

The fundamental message of this section is that the state machine representation is the one that describes the real level of complexity of the source code. The state machine by itself does not introduce any additional algorithmic complexity.

6.2.5 Traceability

When a class is exercised at run-time, the use of an internal state machine makes it 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 directly or indirectly maintain the state of the class.

The implementation of state machines in IGSTK classes facilitates logging information about the inputs, states, and transitions taken by the state machine at every step of its execution. Such information simplifies the task of testing and debugging the class during development. It also make it easier to monitor the behavior of a full IGS application after it has been deployed.

To send tracing information from an IGSTK class to the logger, it is enough to configure the logger to accept messages at least at the DEBUG priority level, and to connect the logger to the IGSTK class by using its `SetLogger()` method. Section 13.2.1 provides details on the logger priority levels and their use.

6.2.6 Suitability for Testing

From the point of view of testing, state machines make it possible to exercise full coverage at the unit testing level. The goal of reaching 100 percent code coverage becomes realistic when state machines are used. Otherwise, the logic of the program diffuses over many lines of code and it becomes very hard to attempt to follow every possible combination of circumstances. When state machines are used, performing 100 percent code coverage becomes a matter of exercising all possible transitions in the transition 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 percent of the application’s code. In honest terms, such a testing suite should be claiming that the applications passed *only* 15 percent of the tests, and the remaining tests have not even been performed.

Test suites with low code coverage have the dangerous effect of providing a false sense of security. 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. A report with the number of passing or failing tests must also be accompanied by the percent of code coverage associated

with the tests.

One of the coding situations in which state machines make a great difference in the effort of raising the code coverage of a test suite is in the removal of “if” conditionals related to error management. Typical error management done with “if” statements makes it difficult for a test suite to simulate the error conditions that will exercise the error management code. Not being able to test error management code has dire consequences in critical applications in which error conditions correspond to particularly dangerous situations. Failing to systematically test the error managing code of your application is as foolish as never checking the fire extinguisher of your house or office, or as never having inspected the seat belts and airbags of your car. Those are items that must work perfectly in emergency situations and for which poor quality is not an acceptable option. State machines, by providing a more systematic approach to error management, make it easier to include this code in the coverage of the testing suite. For most of its development, IGSTK sustained a code coverage of 94 percent or better.

In addition to applying unit-testing practices for verifying the correct behavior of the software driven by state machines, it is also important to verify the correctness of the state machines themselves. For example, by making sure that every state can be reached by at least one transition path that originates in the initial state. This additional verification is known as *State Machine Validation* and it is discussed in detail on Chapter 20.

6.2.7 Consistent Documentation

State machines in IGSTK are capable of exporting their internal logic in formats that are suitable for generating state diagrams. In particular, they can export their list of possible states, possible inputs, and the content of the transition matrix. This functionality makes it possible to automatically generate “State Diagrams” [5] 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 these diagrams and use them as supporting documentation for understanding the behavior of the code.

The state diagrams can be exported in the format of the following applications

- Graphviz format [<http://www.graphviz.org>]
- LTSA format [<http://www.doc.ic.ac.uk/ltsa>].
- SCXML format [<http://www.w3.org/2005/07/scxml>]

Graphviz

Graphviz is a suite of open source visualization tools intended for representing structural information as diagrams of abstract graphs and networks. It is extensively used for documenting software systems. Graphviz provides multiple command line tools. One of them is called “dot” and it is intended for reading the description of a graph from a text file then generating a diagram in graphic formats such as PNG, TIFF, JPEG, or EPS. The IGSTK state machine class

uses the method `ExportDescription()` to generate a text file with the description of the state machine in terms of the syntax that “dot” expects as input. This textual description provides the list of states and the list of possible transitions in the state machine. Examples of the diagrams generated with Graphviz are presented in Figures 5.5, 9.6, 15.1, and 24.2. The state machine diagrams of IGSTK classes are generated using Graphviz as part of the Doxygen documentation of the toolkit.

LTSA

The Labelled Transition System Analyzer (LTSA) is a verification tool for concurrent systems. Given a description of a system in terms of finite state machines (FSM), LTSA can verify whether the system satisfies properties required for its proper behavior. For example, it can determine whether some states can be reached by a sequence of transitions from the initial state, a property known as *reachability*. LTSA also provides functionality for generating animations of the state machine specification to make it easier for developers to interactively explore the system. The IGSTK state machine class provides the method `ExportDescriptionToLTS()` for exporting the state machine description to LTSA.

SCXML

State Chart XML (SCXML) is a working draft published by the World Wide Web Consortium (W3C). SCXML defines an XML notation suitable for describing the semantics of complex state machines. The IGSTK state machine class provides the method `ExportDescriptionToSCXML()` for exporting the state machine description to SCXML.

Since it is common to find many transitions that loop back to the same state, and because such loops tend to obscure the state diagram representation, a boolean flag is available in the `ExportDescription` methods for excluding loops from the state machine diagram. This option must be used carefully. It should not be interpreted as an indication that the loop transitions are irrelevant.

6.3 Implementation

A generic state machine class `igstk::StateMachine` is available in the toolkit and provides an abstraction of the set of states, the set of inputs, and the set of transitions. Every IGSTK component instantiates internally its own state machine and at construction-time it programs the full behavior of the state machine. This organization makes it possible to anticipate how the classes will work when their methods are invoked in any order. When using standard programming techniques, it is rare to find objects that will behave correctly or at least without run-time failures when their methods are invoked in random order. The introduction of state machines in IGSTK is intended to minimize the occurrence of such run-time failures.

6.3.1 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.

In general, application developers do not need to use the state machine API when building their applications. This is because the state machines are not exposed in the public API of IGSTK classes. State machines are only used inside of IGSTK classes. This section is of interest for developers of new IGSTK classes, and for application developers who wish to control the logic of their application using a state machine, a practice that is strongly encouraged here.

The state machine API consists of three main groups of methods:

- Methods for programming the state machine logic
- Methods for executing the logic
- Methods for querying the logic

The following methods are used for programming the state machine

- `AddInput(InputType, DescriptorType)`
- `AddState(StateType, DescriptorType)`
- `AddTransition(StateType, InputType, StateType, ActionType)`
- `SelectInitialState(StateType)`
- `SetReadyToRun()`

The `StateType` and `InputType` are instantiations of the `igstk::StateMachineState` and `igstk::StateMachineInput` classes respectively. These two classes are templated and must be instantiated using as template argument the IGSTK class that owns the state machine. The `DescriptorType` is a textual identifier of the state or input and it is typically a string. Note that this string is purely descriptive. It is not used as the identifier of the state or input. The descriptor string is mainly used in state diagrams that are exported from the state machine for documentation purposes.

The `AddInput` and `AddState` methods insert new inputs and states in the internal arrays of the state machine. Each one of the Inputs and States must also be declared as member variables of the IGSTK class that owns the state machine. This may seem redundant, but it serves the purpose of ensuring that states and inputs are not interchangeable among different classes, and that each one of them has a unique identifier. The standard C++ type-safety mechanism will prevent developers from using the state machine inputs of one IGSTK class into the state machine of a different IGSTK class. The value of the identifier is produced at construction time of the Input and State via the constructor of their base class, the `igstk::Token`.

The `AddTransition` method expects as arguments the initial state of the transition, the input that will trigger the transition, the state to which the state machine will move after the transition, and the pointer to the method that will be executed as the action of the transition.

The `SelectInitialState` method defines the state that will be used for initializing the state machine. This method is intended to be called only once. The `SetReadyToRun` method indicates the completion of the programming stage of the state machine. This method sets an internal flag in the state machine to reject any subsequent calls to `AddInput`, `AddState`, `AddTransition`, and `SelectInitialState` as errors. The purpose of this method is to separate the stage of programming the state machine from the stage of running the state machine. Ideally, the call to `SetReadyToRun` should be the last call in the constructor of the `IGSTK` class that owns the state machine.

The logic programming methods described above must be called in the constructor of the `IGSTK` class that owns the state machine. To simplify the writing of the code and to enforce style consistency, a set of equivalent C++ macros are defined. They are:

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

These macros enforce the convention that states must end with the string “State” and inputs must end with the string “Input”. They also enforce the convention that the descriptor string must be the same name as the state or input. For example, the macro

```
igstkAddInputMacro( TransformModified );
```

is expanded to the code

```
this->m_StateMachine.AddInput(
    this->m_TransformModifiedInput, "TransformModifiedInput" );
```

and the macro

```
igstkAddStateMacro( TrackerInitialized );
```

is expanded to the code

```
this->m_StateMachine.AddState(
    this->m_TrackerInitializedState, "TrackerInitializedState" );
```

The macros are expected to be used in the following specific order:

1. the set of `igstkAddInputMacro()` calls
2. the set of `igstkAddStateMacro()` calls
3. the set of `igstkAddTransitionMacro()` calls
4. the single call to `igstkSetInitialStateMacro()`
5. the single call to `SetReadyToRun()`

Once the state machine logic has been programmed in the constructor of the IGSTK class, it is possible to start executing this logic by performing calls to the “Request” methods of the IGSTK class. These methods internally will make calls to the group of state machine methods that control execution. These internal methods are used for passing an input to the state machine and for triggering the execution of the transition. They are:

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

Inputs passed to the state machine with the `PushInput` and `PushInputBoolean` methods are stored internally in a queue. They are processed later, in the same order of insertion, by the `ProcessInputs` methods. Pushing inputs in the queue does not immediately trigger the execution of the state machine.

The typical use of these methods will be:

```
void RequestStartTracker()
{
    this->m_StateMachine.PushInput( this->m_StartTrackerInput );
    this->m_StateMachine.ProcessInputs();
}
```

Again, for the purpose of enforcing the consistency of the coding style, a macro was created for pushing inputs into the state machine, called `igstkPushInputMacro`. The previous call will be replaced with:

```
void RequestStartTracker()
{
    igstkPushInputMacro( StartTracker );
    this->m_StateMachine.ProcessInputs();
}
```

It is common for the “Request” methods, when they receive arguments, to perform some level of validation in those arguments and, depending on the result, to select the input that will be passed to the state machine. For example, a method for requesting to set the number of iterations of an optimizer would look like:

```
void RequestSetNumberOfIterations( unsigned long n )
{
    this->m_NumberOfIterationsToBeSet = n;
    if( n > 100000L )
    {
        igstkPushInputMacro( InvalidNumberOfIterations );
    }
    else
    {
        igstkPushInputMacro( ValidNumberOfIterations );
    }
    this->m_StateMachine.ProcessInputs();
}
```

Since this case is quite common, the method `PushInputBoolean` was added for convenience. The method allows to push in the state machine queue one of two possible inputs. It will be used in the following way:

```
void RequestSetNumberOfIterations( unsigned long n )
{
    this->m_NumberOfIterationsToBeSet = n;
    this->m_StateMachine.PushInputBoolean( ( n > 100000L ),
                                            InvalidNumberOfIterations,
                                            ValidNumberOfIterations );
    this->m_StateMachine.ProcessInputs();
}
```

The final groups of methods in the public state machine API are intended to export the state diagram of the state machine logic. These methods are public and can export the description of the state machine to Graphviz, LTSA, and SCXML respectively (see Section 6.2.7):

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

6.3.2 Safe States: Attempting Pattern

The approach followed for programming the state machines of the different IGSTK components is the one for “walking in a mine field”. This means that before executing every operation, IGSTK components verify 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 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 6.3 illustrates the use of this pattern of interaction in the context of the `igstk::DICOMImageReader`.

6.3.3 Communication Protocols

The premise behind this implementation feature of the state machine is to 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 at that point, 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.

6.3.4 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 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 always the same, a set of helper Macros were defined in the toolkit 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. The transduction mechanism is extremely powerful and provides a natural layer of integration between the state machine based implementation of IGSTK components and the event-based communication layer.

For a more detailed discussion of Events and their use in IGSTK see Chapter 7.

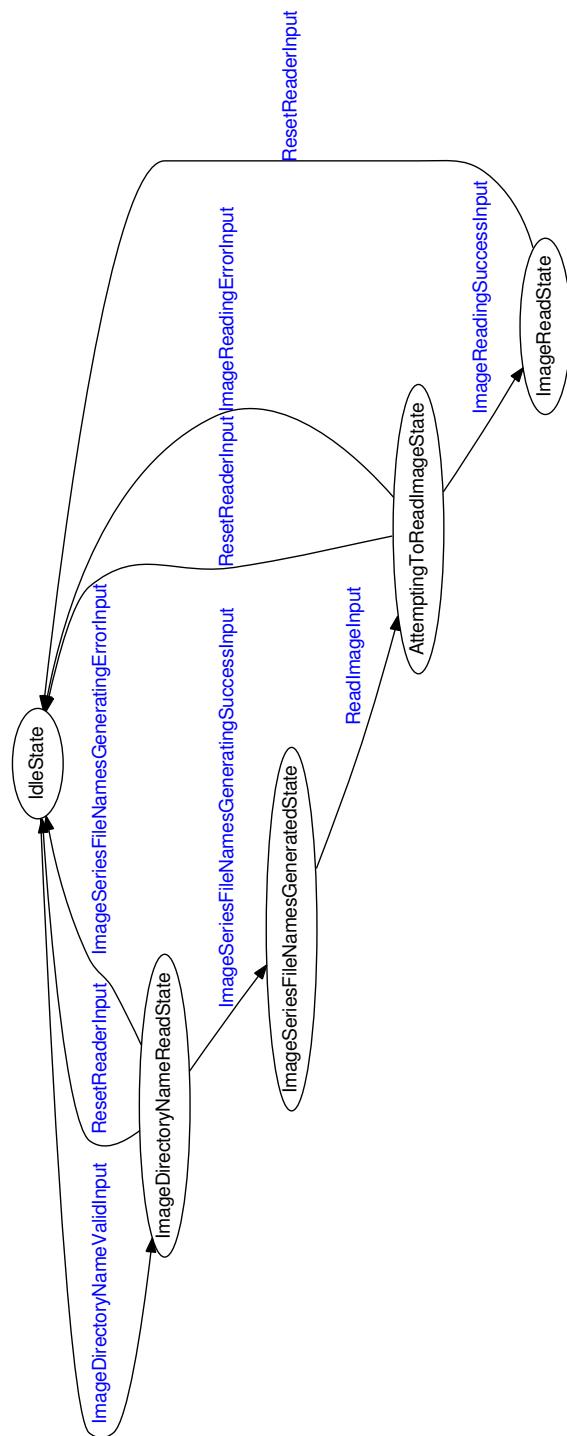


Figure 6.3: Example of the State Machine *Attempting* Pattern in the `DICOMImageReader` Component.

6.3.5 Integration Inside a Class

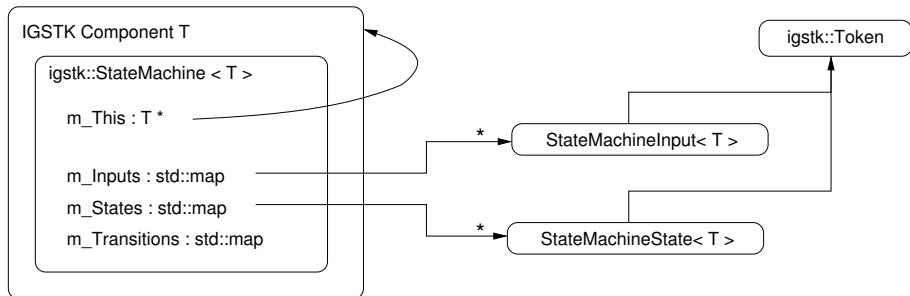


Figure 6.4: Illustration of the State Machine Main Structure and How It is Integrated Inside an IGSTK Class.

State machines are hidden inside IGSTK classes, as illustrated in Figure 6.4 . Every major class in the toolkit 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 of inputs and states unique for this particular class. In this way, there is no possibility of erroneously sending to the state machine of type “A” the inputs that were intended for the 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.

The state machine is instantiated as a member variable of the IGSTK class, and holds a pointer to the IGSTK class itself. Using this pointer, the state machine will be able to invoke methods of its owner class. Since most of the methods that the state machine will call are declared `private`, the state machine is declared as `friend` of the IGSTK class to grant it access to such methods.

Once the state machine is instantiated as a type of the IGSTK class, it defines a number of traits, or types, for representing States, Inputs, and Transitions. The instantiation of the state machine and the declarations of its traits are made via C++ preprocessor macros to enforce their consistency across the toolkit. The main macro that instantiates the State machine is the `igstkStateMachineMacroBase`. This macro defines for the developer the types listed in Table 6.1. The `StateMachineType` is the type that results from instantiating the state machine class using the current IGSTK class as template argument. The `StateType` and `InputType` are traits defined in the state machine itself, and they correspond to the `StateMachineState` and `StateMachineInput` classes instantiated using the IGSTK class as template argument. These two types provide compilation-time verification that the states and inputs passed to a specific state machine will only be the ones associated with the specific IGSTK class that owns the state machine.

The `ActionType` is also a trait of the state machine and it is defined as a pointer to a method of the IGSTK class. The signature of this method is `void f(void)`, which means that it does not take any arguments and it does not return any value. All the methods of the IGSTK class that are intended to be called from the state machine will have this same signature. Each one of such methods corresponds to an action to be performed as the result of a transition in the state

Table 6.1: Traits Defined by the `igstkStateMachineMacroBase`.

<code>StateMachineType</code>	<code>::igstk::StateMachine< Self ></code>
<code>StateType</code>	<code>StateMachineType::StateType</code>
<code>InputType</code>	<code>StateMachineType::InputType</code>
<code>ActionType</code>	<code>StateMachineType::TMemberFunctionPointer</code>
<code>OutputStreamType</code>	<code>StateMachineType::OutputStreamType</code>
<code>ReceptorObserverType</code>	<code>::itk::ReceptorMemberCommand< Self ></code>
<code>ReceptorObserverPointer</code>	<code>ReceptorObserverType::Pointer</code>

machine. The `OutputStreamType` is used as the byte stream where the description of the state machine logic will be exported, as discussed in Section 6.2.7. This stream is typically an output file.

The `ReceptorObserverType` is the type of Observer that will be used for listening to Events from other classes and converting them into inputs to the current state machine. This conversion is explained in detail in Section 7.3.3. The type `ReceptorObserverPointer` defines a smart pointer to the `ReceptorObserverType`.

The macro `igstkStateMachineMacroBase` also adds to the IGSTK owner class the declaration of an instance of the state machine as a member variable of the class. The instance is declared private with the goal of preventing application developers from tampering with the internal structure of the state machine. As a result, users of IGSTK classes are not exposed at all to the internal state machines. Application developers should be able to use the class without being aware of the existence of the state machine as the element controlling the logic of the class. Not even the IGSTK classes are allowed to query the current state of their own state machine. Allowing such access would have permitted the logic of the state machine to leak into the IGSTK class in the forms of “if” statements that make decisions based on the current state of the class.

The macro also calls internally the `igstkFriendClassMacro` to specify that the state machine type is a friend of the current IGSTK class and therefore has access to its private methods. The declaration of friendship must be done through a macro to deal with differences in how the C++ standard has been implemented in different compilers.

Finally, the macro adds to the IGSTK owner class the declaration of the functions that will export the description of the state machine. These methods simply delegate the call to the internal instance of the state machine; therefore, they are simply a mechanism for exposing to the public API of the IGSTK class the functionality that is already provided by the state machine itself. This is necessary because, as stated earlier, the state machine is defined as a private member variable of the IGSTK class. The methods that export the description of the state machine have been listed in section 6.2.7.

The state and inputs of the State machine must be declared as member variables of the owner IGSTK class. The helper macros `igstkDeclareInputMacro` and `igstkDeclareStateMacro`

are available for this purpose. These macros are intended to be used in the header files of the IGSTK classes and will typically take the following form:

```
class PulseGenerator : public Object
{
...
private:
    igstkDeclareInputMacro( ValidFrequency );
    igstkDeclareInputMacro( InvalidLowFrequency );
    ...
    igstkDeclareStateMacro( Initial );
    igstkDeclareStateMacro( Stopped );
    ...
};
```

If a developer forgets to declare any of the inputs or states in the header file, the compiler will refuse to compile the macros that add the same inputs or states in the constructor of the IGSTK class. Such compilation errors will appear in the expansions of `igstkAddStateMacro` and `igstkAddInputMacro`.

The IGSTK class owning the state machine passes its “this” pointer to the state machine at construction time. The state machine needs the “this” pointer to invoke the methods of the owner. The initial lines of the typical construction methods of an IGSTK class will appear as follows:

```
/** Constructor */
Tracker::Tracker(void) : m_StateMachine( this )
{
    // Set the state descriptors
    igstkAddStateMacro( Idle );
    igstkAddStateMacro( AttemptingToEstablishCommunication );
    ...
    igstkAddInputMacro( ActivateTools );
    igstkAddInputMacro( StartTracking );
    ...
    igstkAddTransitionMacro( AttemptingToEstablishCommunication,
                            Success,
                            CommunicationEstablished,
                            CommunicationEstablishmentSuccess );
    ...
    igstkSetInitialStateMacro( Idle );
    m_StateMachine.SetReadyToRun();
```

```
}
```

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 of the owner's class *Processing* methods, if any, to call as the action to be performed during its state transition.

A typical *Request* method could be the following:

```
public:  
void RequestStartTracking()  
{  
    igstkPushInputMacro( StartTracking );  
    this->m_StateMachine.ProcessInputs();  
}
```

and its associated *Processing* method could be the following:

```
private:  
void StartTrackingProcessing()  
{  
    // ... actually trigger the tracking process  
}
```

The *Processing* method can only be invoked if it is listed as the action method in any of the state transitions defined during the programming stage of the state machine. The state transitions will look like the following lines of code

```
igstkAddInputMacro( StartTracking );  
  
igstkAddStateMacro( Tracking );  
igstkAddStateMacro( ToolsActive );  
  
igstkAddTransitionMacro( ToolsActive, StartTracking, Tracking, StartTracking );
```

Where `igstkAddInputMacro` declares a state machine input, `igstkAddStateMacro` declares a state machine state, and `igstkAddTransitionMacro` declares a state machine transition. The arguments of the `igstkAddTransitionMacro` are the initial state, the input, the final state and the action method respectively. In the case of the first transition listed here, the macro will expand the names in the following way

- The initial state expands to `m_ToolsActiveState`
- The input expands to `m_StartTrackingInput`
- The final state expands to `m_TrackingState`
- The action to be executed expands to `StartTrackingProcessing()`

This illustrates how the separation between the Request and the actual Processing methods introduces a layer of protection in the form of the state machine logic. The `StartTrackingProcessing` method will only be called by the state machine if all the conditions are appropriate. The `RequestStartTracking` method simply submits a petition to the state machine for its consideration.

Note that Request and Processing methods do not have to be associated one to one. There may be cases where three different Request methods may eventually result in the execution of the same Processing method. The fundamental point here is that the Request methods only submit petitions to the state machine, and it is up to the state machine logic to decide what Processing method to invoke as a response or to ignore the request all together.

6.4 Conclusion

The introduction of state machines in IGSTK provided a level of formalism that can rarely be achieved with the simple use of object-oriented techniques. However, to fully benefit from this programming style, application developers must assimilate the elements of this methodology, and avoid at all cost to try to bend IGSTK classes back into a procedural type of programming. Such attempts will not only make it harder to develop an application but also undermine the advantages that were gained by using a state machine based formalism in the IGSTK toolkit.

The state machine based design of IGSTK makes it very natural to integrate with GUI libraries, and in particular with applications using event-based communications. We strongly encourage application developers to use state machines for driving the top-level logic of their applications, to achieve at the application level the same degree of determinism, testability, and safety that was reached in IGSTK at the toolkit component level.

Events

“You only need to know things on a need-to-know basis.”

– Yes, Prime Minister - 1986

7.1 General Background

In the context of Object-Oriented Programming, “*encapsulation*” is a technique that keeps an object from exposing the details of its implementation to other objects that interact with it. Encapsulation of components is one of the techniques used in IGSTK for addressing the concern of improving safety and robustness. The effectiveness of encapsulation is further improved when combined with decoupling of software components.

In practical terms, decoupling means that one object should need to know as little as possible about any other object. Of course, the ideal case is when an object does not need to know anything at all about other objects that will interact with it. This seems to contradict with the need for objects to pass information between themselves during their interactions. However, the use of the event and observer pattern [5, 9] establishes a bridge for passing information between two objects while still preserving their encapsulation and their full decoupling.

The use of events is 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 as follows:

- Decoupling of collaborating classes.
- Containment of API modifications.

- Improving code reuse.
- Providing a mechanism for error handling.

7.2.1 Class Decoupling

Decoupling classes that need to collaborate one with another is probably the strongest motivation for the use of events in the context of the safety-critical applications. Decoupling strives to reduce the interdependency between classes. The less a class “A” knows about the internal implementation of another class “B,” the less this first class “A” depends on assumptions about class “B,” and therefore, the more robust it will be when faced with changes to the implementation of “B.” Decoupling also makes it easier to perform unit testing of individual classes, by making it possible to focus on the contractual interface of the class, as opposed to considering specific features of the collaborating classes.

7.2.2 API Containment

It is a common mistake to assume that software is written once and that the task can be brought to a state of completion. In practice, software is continuously evolving, according to the needs of the user community and to new available technologies. Software that is not designed to facilitate this evolution becomes easily degraded when changes start to happen. One thing that often changes in an object-oriented programming library is the public interface (API) of specific classes. This means that the methods of classes may be renamed or removed, or they may change the type of their arguments. When such changes take place, all of 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 aggravation that it is cumulative over time. When method calls are replaced with events as the mechanism for passing information between classes, the API modifications become more localized and do not affect many other pieces of software. Of course, the mere adoption of event-based communications cannot completely solve the problems inherent to an evolving API. They can, however, mitigate its complications.

7.2.3 Code Reuse

Code reuse is also improved by introducing events and observers as the mechanism for passing information between objects of different classes. In particular, by using events, it becomes easier to connect two objects 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. Variations of classes that perform similar tasks can be replaced one for another as long as they respond and emit the same types of events. This sort of reuse is more flexible than polymorphism because it does not require the two classes to derive from a common base class. They simply need to support a similar dialog based on events.

7.2.4 Error Handling

Using events as a mechanism for error handling is a very natural approach in the context of an event-driven architecture as well as in the context of programming based on state machines. When errors are reported in the form of events, it becomes possible to configure as event-observers the classes that must be notified of error occurrences. These observer classes can translate the received error-events into inputs to their respective state machines and, by this mechanism, integrate error handling as a natural aspect of the class logic. This approach formalizes error handling as a normal activity of the classes instead of relegating it to an exceptional activity that “should never happen.” Once integrated into the state machine logic, error handling becomes receptive to formal analysis by using state machine validations tools. This also prevents the dispersion of logic across the code in the form of conditional statements that check for error conditions after every action is executed. Such logic dispersal is detrimental for enforcing safety because it makes it more difficult to analyze and validate the code, and it also prevents the testing framework from reaching high levels of code coverage.

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.

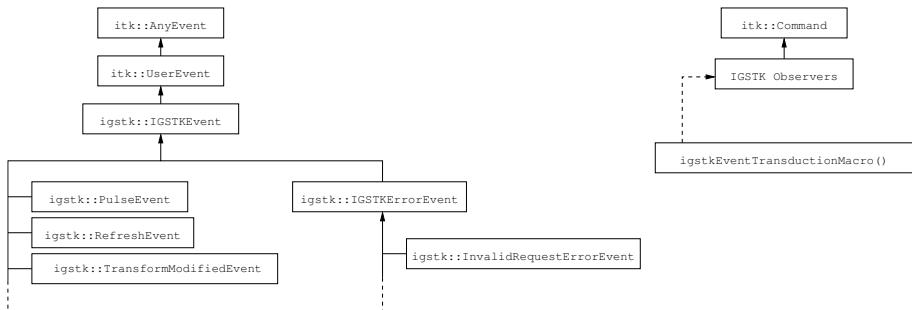


Figure 7.1: Events Class Hierarchy. On the left, the class hierarchy of ITK events starts with the `itk::AnyEvent` class. ITK provides the generic `itk::UserEvent` as a hook for deriving application specific events. IGSTK takes advantage of this existing hook to derive from it the base class of IGSTK events. On the right side is the hierarchy of IGSTK observers. They derive from the `itk::Command` class that is the generic observer in ITK. The use of Macros facilitates the consistency on the definition of Events and Observers.

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 deriving it from the `itk::UserEvent`.

Figure 7.1 shows a small subset of the many events that derive from the `IGSTKEvent`. For example, 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. Also, it is worth noting that recent versions of IGSTK contain the `igstk::IGSTKErrorEvent`, as shown in Figure 7.1. This event is the parent of all error events in IGSTK. `igstk::InvalidRequestErrorEvent`, also shown in Figure 7.1, is just one of the error events in IGSTK. `igstk::IGSTKErrorEvent` allows a developer to register for all errors on an object, not just those expected at development time.

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 have only a type are used to notify other objects 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 objects of different class without requiring these two classes to know about each other. To facilitate the creation of events with payload, a set of Macros and Templated classes are provided in the following files:

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

The more important group of these macros is defined in the `igstkEvents.h` file. This group includes the following macros:

- `igstkEventMacro`
- `igstkLoadedEventMacro`
- `igstkLoadedObjectEventMacro`
- `igstkLoadedTemplatedObjectEventMacro`
- `igstkLoadedTemplatedConstObjectEventMacro`

The `igstkEventMacro` declares 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 SmartPointers¹. The `igstkLoadedTemplatedObjectEventMacro` should be used when the payload class uses SmartPointers and is also a templated class. The `igstkLoadedTemplatedConstObjectEventMacro` should be used when the payload class uses SmartPointers, is a templated class and is const. 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 &)`

Events with payload created by the `igstkLoadedObjectEventMacro` have the following standard methods:

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

Events of this last type store internally a Smart Pointer to a `PayloadType` object.

The version of macros for templated classes do not have any API differences with the `igstkLoadedObjectEventMacro`. They are needed only to deal with the syntax differences in C++ `typedef` declarations of traits taken from templated classes.

Events with payload are particularly 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` defined in the `igstkMacros.h` file. When this macro is used inside an `IGSTK` class, it will create an observer as a member variable of that class, and will also define methods that make it possible for an object of that class to connect its internal observer to an instantiation of 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 object that owns the observer. In this way,

¹Smart pointers automatically increase and decrease an instance's reference count, deleting the object when the count goes to zero.

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 mechanism of the transduction macro. An object of class “B” is intended to observe events “E” sent from an object of class “A.” Upon reception of any event “E,” the object of 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 into which we want to translate that 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 `CallbackEventInput` 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 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 suffix *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.”

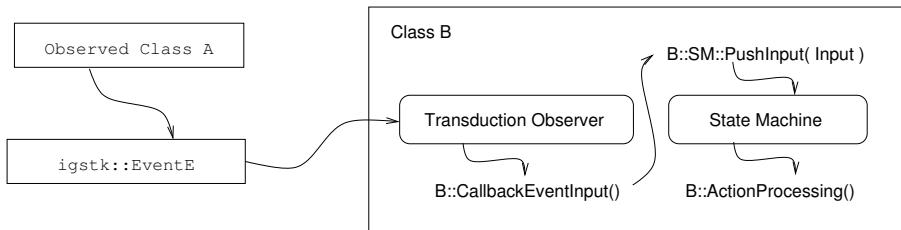


Figure 7.2: Events/Input Transduction. A set of macros are available to simplify the process of converting an event into a state machine input. In this way, event-messages and state machine logic can be integrated. In this diagram, an object of class A produces an event E that is observed by an object of class B. The transduction macro has been used in class B for generating code that will convert the incoming event E into a specific input to the internal state machine of class B.

7.3.4 Observers

IGSTK does not define 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 flexibility is required by the internal implementation of the IGSTK class. The second option is used in the `igstk::Tracker` and `igstk::View` classes 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 developers who want to modify or maintain the source code of IGSTK. Developers of image-guided surgery applications based on IGSTK need only be concerned with defining observers when they query information from particular IGSTK classes, mostly for the purpose of monitoring the behavior of those classes. The communication between application code and the toolkit classes is described further 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, communication between IGSTK objects is done internally through events. Developers of applications do not need, in general, to deal with the events that are being emitted from IGSTK objects. It is useful, however, to know how information is flowing between the different IGSTK components. We present here some details about how the mechanism works.

First of all, to pass an event “E” from an object of class “A” to an object of 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`.
- An instantiation of class “B” must add its observer to an instantiation of class “A” using the `AddObserver` method of class “A” and specifying that it is interested in event “E” or one of its derived classes.

Figure 7.3 illustrates these requirements. The left side 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 instantiations of this class that events of type “E” will be sent. The middle of the figure illustrates the hierarchy of event “E,” which must derive directly or indirectly from the `IGSTKEvent`. The right side of the figure illustrates the collaboration between class “B,” which wants to know about event “E” and its helper class, the IGSTK observer, which 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 from an object of class “A.” Finally, the bottom of the figure shows the invocation of the `AddObserver()` method from class “A.” This invocation informs an object of class “A” that the observer is interested in being notified whenever event “E” occurs. Since the IGSTK observer

is a generic class, objects of class “A” do not need to know about the existence of objects of class “B,” and objects of class “B” do not need to know about the existence of objects of class “A.” In this way, decoupling between classes “A” and “B” is completely achieved. Of course, the application developer will be responsible for calling the `AddObserver` method from the application’s code.

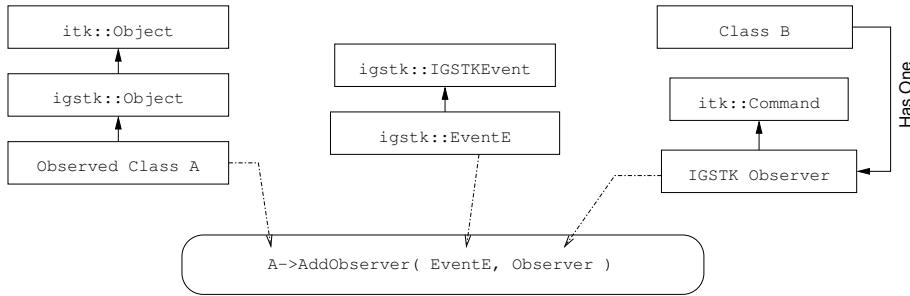


Figure 7.3: Events Usage. To use events in IGSTK, the observed class must derive from the `igstk::Object` class, an event must be defined deriving from the `IGSTKEvent` class, an observer class must be created deriving from the `itk::Command` class, and an instantiation of the observer class must be registered with an instantiation of the observed class by calling the `AddObserver` method.

7.4.2 External Usage: The Request/Observe Pattern

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

The Request/Observe pattern is intended to deal with a scenario in which 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 combines it with one of the three 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 safety-critical applications, the first option has the drawback that every time the “Get()” method is called by the customer of the class, he or she has to check whether the data object is valid. This results in code plagued with “if” conditions that are difficult to debug and hard to maintain, and it becomes very difficult to enforce testing with 100 percent 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 *precondition* and it is common in *contract-based programming*. There are two drawbacks to this method. The first is that it relies on application developers to call the “IsValid()” method before calling the “Get()” method. In C++ there is no way to make sure that the “IsValid()” method is 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 is returned true at any given point, then 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, which may or may not be valid. The validity of the data object is indicated by the boolean that is simultaneously returned by the function. This is equivalent to combining the “IsValid()” and “Get” functions into a single atomic operation. This option still has the drawback that the customer’s code is plagued with “if” conditions that are detrimental to the robustness and safety of the code.

As an alternative to these three typical options, IGSTK introduced 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 states, inputs, and transitions in the state machines of the two objects involved in the transaction of the data object. Figure 7.4 shows an intuitive (non-UML conforming) sequence diagram of the transaction between objects of two different classes. On the left side of the figure, the “Customer” class is the one that needs 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 of the interaction is for the “Customer” object to connect its internal observers to the “Provider” object. 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 has no payload and signals an error condition. These 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. These connections will typically be performed at initialization time of the application.

The next step is for the “Customer” to call a method in the “Provider” in order to ask it to send 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” object 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.” The “Customer” goes into a “Waiting” state, also called an “Attempting” state in IGSTK. Depending on the current state of the “Provider,” that input may result in different transitions and associated actions in the state machine of the “Provider.” Two typical occurrences 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 it is

sent to the “Customer” by using the `InvokeEvent` method. The second occurrence 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 indicating the non-validity of the requested value is sent 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,” which 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 Transition 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, as discussed in Section 7.3.2. If, by the time the event gets to the “Customer,” its state machine is in a state that cannot process the data object or it is not interested in processing the data object anymore, then the data is simply ignored. Note, however, that it is up to the developer of the state machine of the “Customer” class to define the specific action to be taken for every possible state and input pairs of the class.

In the case of “State Transition B,” when the “Provider” object is unable of produce a valid data object and invokes an “ErrorEvent”, the Error event is received by the observer in the “Customer” object 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 a very explicit transaction in which the reactions to both valid and invalid data objects are carefully analyzed. Since this mechanism directly takes care of routing the responses of both objects to the condition in which the data object is invalid, it ensures that none of the objects will be put in an invalid state during the transaction. This is a great advantage 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.

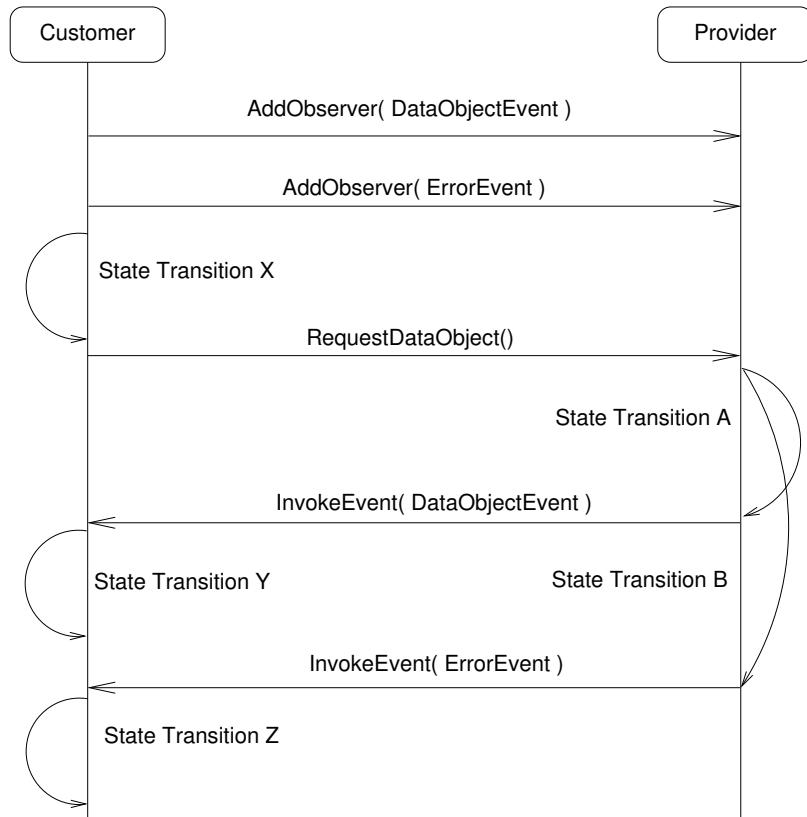


Figure 7.4: Request/Observe Pattern. Sequence diagram indicating the typical interactions between instantiations of two IGSTK classes when one requests a data object from the other. Although this is not a conforming UML sequence diagram, it has been depicted in a similar way.

7.5 Complete Event Hierarchy

Figure 7.5 is the complete event class hierarchy. To get a detailed view of this graph, please refer to the electronic version (pdf) of the book and zoom in for a larger picture.

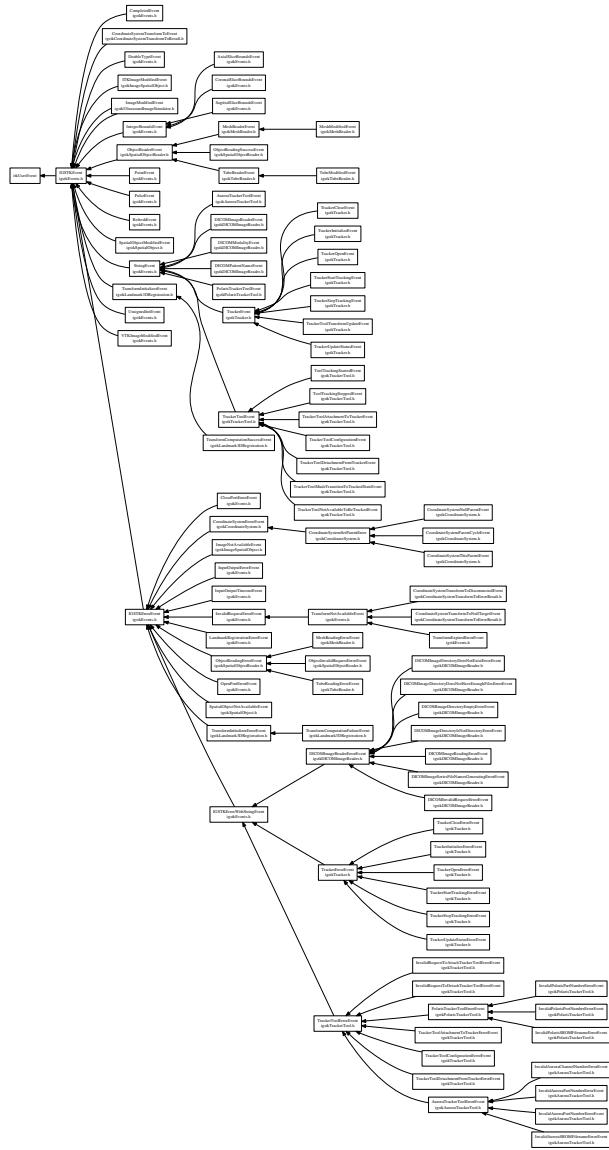


Figure 7.5: Events Class Hierarchy. Complete hierarchy of events in IGSTK, with the source file containing the event.

7.6 Conclusion

Events are an extremely valuable programming technique that is used in many different places in IGSTK. To take advantage of the strengths of events and state machines, application developers should strive to design their own applications based on the same principles. In that way, the safety and robustness of IGSTK design will percolate to the application.

Attempting to introduce IGSTK code into an application with a more traditional design may 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, and that the application architecture itself may have a fragile logical structure.

Part III

Core Components

Coordinate Systems

“In mathematics you don’t understand things. You just get used to them.”

—John Von Neumann

8.1 The Role of Coordinate Systems in IGSTK

Coordinate systems are at the heart of any image-guided surgery toolkit. Coordinate systems in an image-guided surgery system describe the spatial-temporal relationships between the objects involved in a procedure. For example, they provide the framework for moving between pre-operative and intraoperative imagery; between image coordinates and tracking tools; and between tracking tools and the display. Every object involved in the procedure must be located correctly in relation to every other object, not just in space, but also in time. For example, tracker tool positions and the corresponding transformation are sampled at an instant in time. If these relationships are not correctly constructed, the validity of any information or visualization produced by the system is in question.

IGSTK uses a scene graph to manage coordinate systems. The coordinate systems are nodes in a graph that defines the transformations between coordinate systems. The graph may be a single tree, or a forest of trees. This approach is very flexible, allowing the developer great freedom in designing the coordinate system graph.

8.1.1 Advantages and Disadvantages of the IGSTK Scene Graph

The inclusion of a scene graph in IGSTK has several advantages and a few drawbacks. On the positive side, a developer can now specify a complex set of coordinate system relationships in a straightforward manner. Furthermore, the developer only needs to provide a transformation from one coordinate system to another, instead of having to convert every coordinate system into a world coordinate system. The system will automatically use these local transformations to compute the transformation to other, connected coordinate systems. The coordinate scene graph allows easy construction of the previously more complicated features, including having

an `igstk::View` display a scene using the coordinate system of a `igstk::TrackerTool` or an image coordinate system.

On the negative side, there is now more code and functionality to test. Furthermore, the new coordinate system framework is more complex than previous versions of IGSTK. Developers now need to understand the interplay of the coordinate system scene graph and rendering to correctly display surgical scenes. (See Section 8.5 for more information regarding the role of coordinate systems in rendering.)

In the remainder of this chapter, we first provide a more rigorous mathematical definition of transforms in IGSTK. Next, we discuss the basics of the scene graph used to manage coordinate systems in IGSTK. Finally, we describe the C++ classes that provide the IGSTK coordinate system functionality.

8.2 Mathematical Definition and Notation

Consider a point $\vec{p} \in P$, where P is a coordinate system. In IGSTK, \vec{p} is a vector in a three-dimensional Cartesian coordinate system. We wish to find the location of \vec{p} in another three-dimensional Cartesian coordinate system Q . We will denote the result of the transformation of \vec{p} into Q as \vec{q} . Mathematically, we may express this as:

$$\vec{q} = T_{P,Q}(\vec{p}) \quad (8.1)$$

where $T_{P,Q}$ is a transformation which transforms a point from coordinate system P to coordinate system Q . In IGSTK, T is a 3-dimensional, rigid-body transformation, specifically a rotation followed by a translation. Computationally, we use versors to find the resulting transformation. Note that T is a point transformation from a point \vec{p} in a three-dimensional Cartesian coordinate system P to a point \vec{q} in another three-dimensional Cartesian coordinate system Q ; T is *not* a coordinate system transformation.

8.2.1 Transform Composition

Transforms may be composed to move through multiple coordinate systems to a final coordinate system. For example, if we have the transform $T_{P,Q}$ from P to Q and the transform $T_{Q,R}$ from Q to R , then we may compute the resulting transformation from P to R as

$$\vec{r} = T_{P,R}(\vec{p}) = T_{Q,R}(T_{P,Q}(\vec{p})). \quad (8.2)$$

The transformation of \vec{p} into R is first computed by transforming \vec{p} into Q and then transforming the resulting point from Q into R . We compose the transformations by applying the transformation from P to Q and then from Q into R . When composing transforms like this, we must be mindful of the input and output coordinate systems of each transformation to ensure that the transformations are chained together in the correct order.

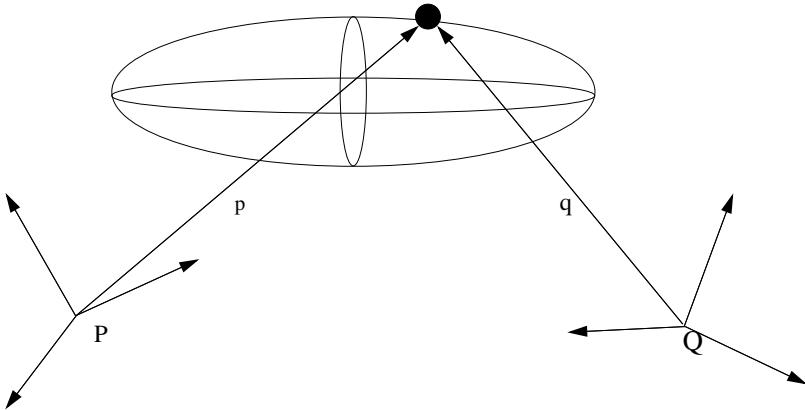


Figure 8.1: Transformation Example.

If we wish to transform from R back into P , we may apply the inverse transformations in the reverse order:

$$\vec{p} = T_{P,R}^{-1}(\vec{r}) = T_{P,Q}^{-1}(T_{Q,R}^{-1}(\vec{r})). \quad (8.3)$$

Since we have restricted ourselves to valid rigid body transformations, the inverse always exists.

8.3 Coordinate System Scene Graph

As mentioned earlier, IGSTK uses a scene graph to describe the spatial-temporal relationships between coordinate systems. A graph is a collection of nodes, also called vertices, and edges that connect nodes. In the context of the IGSTK scene graph, nodes are coordinate systems and the edges are directional and define a relationship, the transformation, between the nodes.

In IGSTK, we restrict the graph to be a tree or a forest. A tree is a graph where each node has a single parent and zero or more children. A tree has no cycles, that is, one cannot traverse the directional edges from an initial node and return to the initial node. A forest is a collection of unconnected trees.

We restrict the IGSTK scene graph to a tree or forest structure to prevent cycles in the coordinate system graph. Cycles could cause ambiguity in the definition of the transformation between coordinate systems. We also do not specifically indicate the root of the tree.

The scene graph in IGSTK is built by connecting a coordinate system to a parent coordinate system. We use parent here to indicate the parent-child relationship typical of nodes in a tree. At the same time the parent is specified, a transform that takes a point from the current coordinate system to corresponding point in the parent coordinate system is specified. Connecting several individual coordinate systems builds the application scene graph. The complete scene graph defines the relationships between all the coordinate systems in IGSTK. It is not necessary that all coordinate systems be connected in the scene graph. However, it is not possible to compute

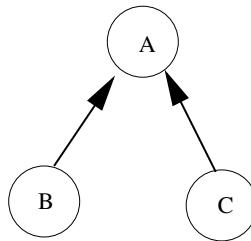


Figure 8.2: Coordinate Systems Tree. A, B, C are coordinate systems. A is the parent of B and C. B and C are children of A. Note that the edges in this figure have the reverse direction from what is often used in illustrations of tree data structures. In our case, the direction of the arrows represent the direction of the transformation on the edge. In IGSTK, the scene graph is constructed by specifying the transformation from the child node to the parent node. As a result, the edges in the figure go from the child nodes to the parent node.

the transformation between two coordinate systems that are not connected in the scene graph.

8.3.1 Transform Computation

To compute the transformation from one coordinate system in IGSTK to another coordinate system, we must determine if there is a spatial relationship between the two coordinate systems. The spatial relationship may be simple, such as the parent-child relationship, or a more indirect relationship defined by a larger subset of the scene graph.

In general, two coordinate systems have a relationship if they are connected in the graph. That is, there exists a path which connects the two coordinate systems in the scene graph. In this context, we treat the edges as if they were undirected, or, in other words, a path does not need to follow the directions of the graph edges. While the directionality of the edges is not used to restrict the path, it does specify the "direction" of the transformation. A forward edge implies the transform on the edge is used directly, while traversing an edge in the reverse direction implies the use of an inverse transformation. (Note, since transforms are restricted to valid rigid body transformations, the inverse transform always exists.) If the two coordinate systems are not connected in the coordinate system graph, it is not possible to compute a transformation between them based on the graph.

In Figure 8.2, the transform from coordinate system B to coordinate system C may be computed by first transforming a point from coordinate system B into coordinate system A. Then, once the point is in A, the inverse transform from C to A may be applied taking the point from A back into C. We can write this as:

$$T_{B,C} = T_{A,C}^{-1}(T_{B,A}). \quad (8.4)$$

In this example, A is a common ancestor of both B and C. In general, if two nodes in the coordinate system graph are connected, they will have a common ancestor. As in the example above,

the transformation between the two coordinate systems can be computed by concatenation of the transforms from each coordinate system to the common ancestor. One of the transforms will need to be inverted; which transform depends on the directionality of the desired transform.

In IGSTK, while computing a transform between two coordinate systems, the implementation attempts to find the lowest common ancestor (LCA) between two coordinates. The lowest common ancestor is an ancestor in the coordinate system graph shared by both coordinate systems. Furthermore, the ancestor is closest to both coordinate systems in the sense that it is the ancestor lowest in the tree. For a graph which is a single tree, the root of the tree is always a common ancestor. However, two nodes in a tree may have an ancestor which is lower in the tree than the root. We wish to find this lowest common ancestor because it reduces the number of transform concatenations. Finding the path between the two coordinate systems through the lowest common ancestor will help reduce both computation time and numerical issues, since there will be few transforms concatenated to compute the final transformation.

8.4 Coordinate System API

8.4.1 Transforms and Time Stamps

Earlier in the chapter, it was mentioned that objects needed to be localized by both position and time. Both a coordinate transformation and time are stored in a `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

The 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 that 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, its internal measurement rate, the number of tools that it is tracking, and the data transmission speed between the tracking system and the computer. Latency is very important when measurements are to be made simultaneously with different tracking systems (for instance, with optical and magnetic systems): 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.

8.4.2 Coordinate System Classes

There are two main classes the make up the coordinate system API in IGSTK. These classes are: `igstk::CoordinateSystem` and `igstk::CoordinateSystemDelegator`.

The `igstk::CoordinateSystem` class is the heart of the implementation. This class provides the basic scene graph and transform computation implementation.

The `igstk::CoordinateSystemDelegator` class exists to provide an isolated, templated version of the coordinate system API. The templated API exists to support the design goal of making it easy to add coordinate system functionality to an existing object.

The `igstk::CoordinateSystem` contains the following API.¹

1. `RequestSetTransformAndParent(const Transform & t, const CoordinateSystem* parent)` : Connect to a parent node in the IGSTK scene graph and specify the transformation from a point in the current coordinate system to a point in the parent coordinate system.
2. `RequestUpdateTransformToParent (const Transform &t)` : Replace the transformation from a point in the current coordinate system to a point in the parent coordinate system with the transform supplied as a function argument.
3. `RequestGetTransformToParent()` : Request the transform to parent.
4. `RequestComputeTransformTo (const CoordinateSystem *targetCoordSys)` : Request that a transform be computed to targetCoordSys. This method generates three possible events: CoordinateSystemTransformToEvent, CoordinateSystemTransformToNull-TargetEvent, CoordinateSystemTransformToDisconnectedEvent.
5. `RequestDetachFromParent()` : Request that the coordinate system be detached from its parent. This detaches the coordinate system from the IGSTK scene graph.
6. `SetName(const char* name)` : Set the name of the coordinate system.
7. `GetName()` : Get the name of the coordinate system.

¹ For the most up-to-date API documentation, please consult <http://igstk.org/documentation.htm>

The `igstk::CoordinateSystemDelegator` contains an nearly identical API to `igstk::CoordinateSystem`. The main difference is that key methods are templated to support adding coordinate system functionality to other classes with minimal effort.

1. *template < class TParentPointer > void RequestSetTransformAndParent(const Transform & transformToParent, TParentPointer parent)* : Templated version of `RequestSetTransformAndParent` in `igstk::CoordinateSystem`.
2. *void RequestUpdateTransformToParent(const Transform & transformToParent)* : Replace the transformation from a point in the current coordinate system to a point in the parent coordinate system with the transform supplied as a function argument.
3. *void RequestGetTransformToParent()* : Request the transform to parent.
4. *void RequestDetachFromParent()* : Request that the coordinate system be detached from its parent. This detaches the coordinate system from the IGSTK scene graph.
5. *template <class TTTarget> void RequestComputeTransformTo(const TTTarget & target)* : Request that a transform is computed to targetCoordSys This method generates three possible events: `CoordinateSystemTransformToEvent`, `CoordinateSystemTransformToNullTargetEvent`, `CoordinateSystemTransformToDisconnectedEvent`.
6. *bool IsCoordinateSystem(const CoordinateSystemType*) const* : Allows another object to verify which coordinate system an object owns. This does not check that the coordinate systems are equivalent, but whether the instances of the coordinate systems are identical. This method is useful for verifying the identity of the source and destination coordinate systems in a `igstk::CoordinateSystemTransformToEvent` .
7. *void PrintSelf(std::ostream& os, itk::Indent indent) const* : Print out object information.
8. *void SetName(const char* name)* : Set the name of the coordinate system.
9. *void SetName(const std::string& name)* : Set the name of the coordinate system. Same as `SetName` above, but different signature.
10. *const char* GetName() const* : Return the name of the coordinate system.

8.4.3 Coordinate System Macros

There are two coordinate system macros which exist to make adding coordinate system functionality to an object easy. These macros are located in `igstkCoordinateSystemInterfaceMacros.h`.

The first macro is `igstkCoordinateSystemClassInterfaceMacro`. This macro should be placed in the header of a class that wishes to implement the coordinate system API.

The second macro is `igstkCoordinateSystemClassInterfaceConstructorMacro`. This macro should be called in the constructor of an object that wishes to implement the coordinate system API.

8.4.4 Classes that Have Coordinate Systems

Many of the classes in IGSTK now make use of coordinate systems. These classes include:

1. `igstk::SpatialObject` and subclasses;
2. `igstk::View` and subclasses;
3. `igstk::Tracker` and subclasses;
4. `igstk::TrackerTool` and subclasses;
5. `igstk::ObjectRepresentation` and subclasses.

As a result, it is possible to compute transformations between any of these objects in the scene graph, provided that they are connected in the graph. Furthermore, the coordinate systems for these objects may be specified in terms of the coordinate system of any of the other objects through a call to `RequestSetTransformAndParent`.

8.4.5 Coordinate System Events

1. *CoordinateSystemErrorEvent* - This is the parent class for any coordinate system error.
2. *CoordinateSystemSetParentError* - This is the parent class for any coordinate system error in setting the parent of a coordinate system.
3. *CoordinateSystemNullParentEvent* - This error event occurs when `NULL` pointer is passed to `RequestSetTransformAndParent`.
4. *CoordinateSystemThisParentEvent* - This error event occurs when `this` pointer is passed to `RequestSetTransformAndParent`. This would cause a coordinate system to be its own parent.
5. *CoordinateSystemParentCycleEvent* - This error event occurs when the parent passed to `RequestSetTransformAndParent` would cause a cycle in the scene graph. The code tests whether the coordinate system on which `RequestSetTransformAndParent` is called is reachable by traversing the directed edges in the scene graph from the parent.
6. *CoordinateSystemTransformToEvent* - This event carries the result of computing a transformation between two `igstk::CoordinateSystems`. The payload for this event is `igstk::CoordinateSystemTransformToResult` which contains a pointer to the source coordinate system, a pointer to the destination coordinate system, and a transformation. Before using the resulting transformation, a developer should check that the payload points to the expected source and destination coordinate systems, especially when using the same listener for multiple `igstk::CoordinateSystemTransformToEvent` events.

8.5 Coordinate Systems and Object Display

`igstk::CoordinateSystem` is used by `igstk::View`, `igstk::SpatialObject`, `igstk::Tracker`, and `igstk::TrackerTool`. The connections between these objects define the coordinate system relationships between them. These relationships may be static or dynamic. That is, the relationships may be fixed at application development time (for instance, a view that is fixed in space) or they may be dynamic (for example, the relationship between a tracker tool and a tracker).

One facet of IGSTK that causes new IGSTK developers trouble is the interplay between the coordinate system scene graph and object display. Each `igstk::View` has a list of objects that may be displayed in that view. At rendering time, the view traverses the list of objects and asks them to compute a transformation from that object's coordinate system into the view's coordinate system. This transformation is used to properly position the object in the rendered view. *If the object does not supply a valid transformation at render time, it will not be displayed in the view. If the object is not connected to the view through the coordinate system scene graph, it can not supply a valid transformation.* It is critical then, to not only add an object to the view, but also to make sure that there is a path through the coordinate system scene graph which connects the object and the view.

While debugging, one quick way to check if the object and the view are connected is to ask the object to compute a transformation to the view (or vice-versa). For example, one might try: `mySpatialObject->RequestComputeTransformTo(myView);` to see if `mySpatialObject` and `myView` are connected in the coordinate system scene graph.

Another debugging approach could be to print `mySpatialObject` and `myView`. When a `igstk::CoordinateSystem` is printed (by itself or as a member of another object), it will traverse the parent pointer internal to the `igstk::CoordinateSystem`, printing the pointer and name of each coordinate system it encounters until it reaches the root of the tree. Closely examining the output may indicate if the view and spatial object are connected. It is best to print both objects, since the coordinate system graph is only printed (and traversable) from child to parent, and it may not be known which object is a child of the other in the coordinate system graph.

8.6 Scene Graph Visualization

The scene graph is a data structure whose composition must be correct. The path between any two nodes represents a computational process involving an intended set of transforms. IGSTK provides some methods to inspect that computational process (the path), but much as a surgeon visualizes a patient's anatomy internally, a developer must internally visualize this data structure. To prevent miscomputations - correct computations but along unintended paths in the scene graph - we built a scene graph visualization tool. This tool allows developers to see, as IGSTK is running, the structure of the scene graph and the computational paths used for rendering a scene. Essentially, this helps developers explicitly see the scene graph to rendered object view the previous section describes.

Section 8.4.4 shows the types of objects that have a coordinate system associated with them in IGSTK. In any given instance of IGSTK, there will be (at runtime) many coordinate system objects and transforms present in the application. To render the potentially volatile scene graph data structure and the computations performed over its paths, our design includes a central repository holding on to these coordinate system objects and the transforms between them. This is essentially an instance of the Blackboard architecture style [23], as the scene graph changes or transformations are computed across its structure, events are generated that the visualizer components respond to by updating its rendered depiction of the structure.

8.6.1 Displaying Static Snapshots of the Scene Graph

To enable this functionality, you need to configure IGSTK with `IGSTK_USE_SceneGraphVisualization` turned ON.

The scene graph is exportable with a button click to the DOT format, which can be read and displayed in GraphViz.

In the "HelloWorld" example (Section 2.4)

```
igstk::SceneGraph *m_SceneGraph = igstk::SceneGraph::getInstance();
m_SceneGraph->ExportSceneGraphToDot(fileName);
```

This is the content of the output dot file.

```
digraph G {
"Tracker 0x016441B0" -> "View3D 0x015853C8";
"Tracker 0x016441B0" -> "MouseTrackerTool 0x01640B00";
"Tracker 0x016441B0"[shape=polygon, sides=4, distortion=-0.7, color=yellow, style=filled];
"View3D 0x015853C8"[shape=polygon, sides=8, color=green, style=filled];
"MouseTrackerTool 0x01640B00"[shape=polygon, sides=6, color=red, style=filled];
}
```

To generate a picture using the dot file, you need the dot tool from the GraphViz package

```
$dot -Tjpg igstkHelloWorld.dot Co igstkHelloWorld.jpg
```

Figure 8.3 is the result as rendered by dot.

8.6.2 Display Scene Graph in FLTK

We can also display the scene graph in FLTK window by using the following command.

```
igstk::SceneGraph* m_SceneGraph = igstk::SceneGraph::getInstance();
m_SceneGraph->ShowSceneGraph(showingSG);
igstk::SceneGraphUI * m_SceneGraphUI = igstk::SceneGraphUI::getInstance();
m_SceneGraphUI->DrawSceneGraph(m_SceneGraph);
```

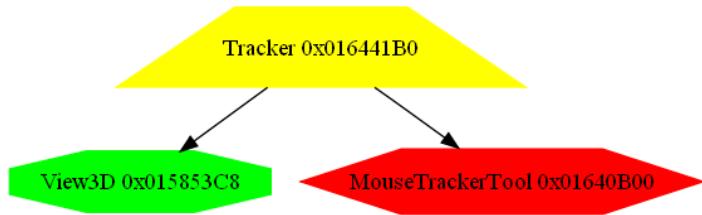


Figure 8.3: Rendering the Scene Graph Using GraphViz.

The screen capture of the display in the "Hello World" example application is shown in Figure 8.4.

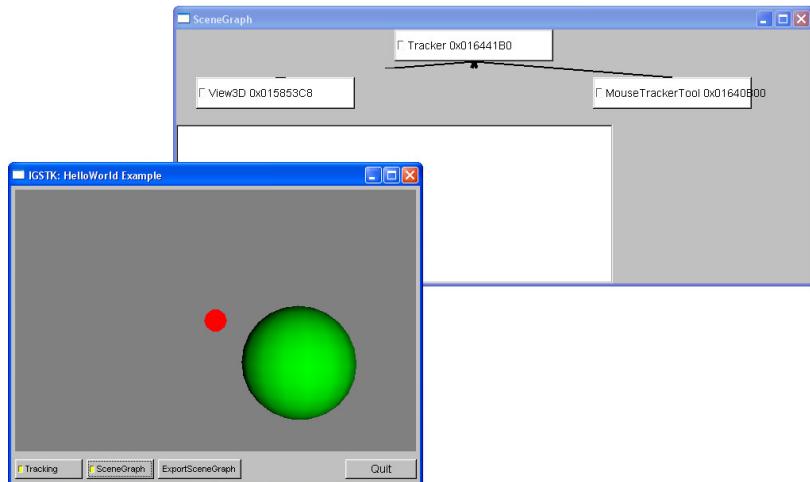


Figure 8.4: Scene Graph Displayed in FLTK GUI.

8.6.3 Displaying Dynamic Changes and Transformations of the Scene Graph

The static snapshot section above describes a process for capturing the structure of the scene graph at any point in time. This by itself is useful as the developer can visually inspect and validate her/his understanding of the scene graph "anatomy". However, the scene graph visualization tool goes further - it also has a mode where it highlights the nodes and edges in the graph involved in computing a transform, and can do so while IGSTK is running.

The code is very simple. A flag was added to each of the nodes in the scene graph visualization structure. If set, then any computation that involves that node results in the path being highlighted on screen. This is a simple filter mechanism that ensures that not every transform computation is highlighted - these are too frequent and too fast to keep track of. Developers can

flag the paths they want to see and control how long to see them so they can match observable IGSTK application behavior with a white-box inspection of the scene graph structure and path used to compute a transform.

Figure 8.5 shows a scene graph with a path highlighted indicating a pairwise transform computation.

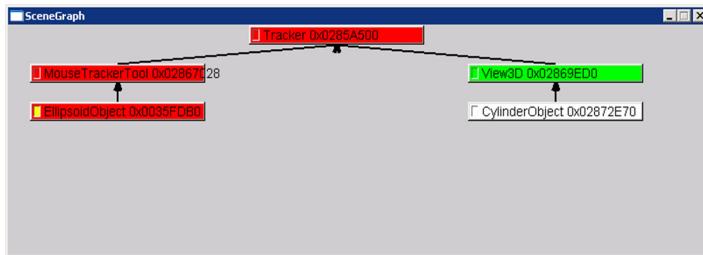


Figure 8.5: Displaying the Scene Graph with a Transform Computation Path.

8.7 Coordinate System Examples

8.7.1 Basic Example

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

This example shows the basic techniques for working with coordinate system objects.

First, we include the appropriate header files.

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

Then we construct a cone object and set the radius and height.

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

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

We construct a transform which will hold the relative position to the parent. We also set some initial values.

```
const double validityTimeInMilliseconds = 200.0;
```

```
igstk::Transform::VectorType translation;
translation[0] = 10.0;
translation[1] = 0.0;
translation[2] = 0.0;
igstk::Transform::VesnorType rotation;
rotation.Set( -0.707, 0.0, 0.707, 0.0 );

igstk::Transform::ErrorType errorValue = 0.01; // 10 microns

igstk::Transform transform;

transform.SetTranslationAndRotation(
    translation, rotation, errorValue, validityTimeInMilliseconds );
```

Next, we construct a cylinder object with a radius and height.

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

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

Then, we build a very simple scene graph by connecting the cylinder to the cone. The cone is the parent and transform specifies the transformation from the cylinder's coordinates into the cone's coordinates.

```
cylinder->RequestSetTransformAndParent( transform, cone );
```

Now we can modify the transform.

```
translation[0] = -10.0;
translation[1] = 0.0;
translation[2] = 0.0;
rotation.Set( 0.707, 0.0, 0.707, 0.0 );

transform.SetTranslationAndRotation(
    translation, rotation, errorValue, validityTimeInMilliseconds );
```

And then we can reconnect the cylinder to the cone with the new transform.

```
cylinder->RequestSetTransformAndParent( transform, cone );
```

8.7.2 View From Tracker Coordinates Example

This example illustrates viewing a scene from the perspective of the tracker. The example is almost identical to the one in Section 8.7.3, but the view is connected to the stationary tracker instead of being connected to the moving tracking tool.

The source code for this section can be found in the file
Examples/CoordinateSystems/CoordinateSystems1.cxx.

This example describes how to build a scene graph with a View, Tracker, and TrackerTool. This example will display the scene from the perspective of the Tracker.

First, we include the appropriate header files.

```
#include "igstkAxesObject.h"
#include "igstkAxesObjectRepresentation.h"
#include "igstkBoxObject.h"
#include "igstkBoxObjectRepresentation.h"
#include "igstkCylinderObject.h"
#include "igstkCylinderObjectRepresentation.h"
#include "igstkView3D.h"
#include "igstkFLTKWidget.h"
#include "igstkRealTimeClock.h"
#include "igstkSimulatedTrackerTool.h"
#include "igstkCircularSimulatedTracker.h"
```

Then we initialize the RealTimeClock. This is necessary to manage the timed events in IGSTK.

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

Now, we create an AxesObject with an AxesObjectRepresentation to provide a visual reference for the Tracker's coordinate system.

```
typedef igstk::AxesObject    AxesObjectType;
AxesObjectType::Pointer axesObject = AxesObjectType::New();
axesObject->SetSize(3.0, 3.0, 3.0);

typedef igstk::AxesObjectRepresentation  RepresentationType;
RepresentationType::Pointer axesRepresentation = RepresentationType::New();

axesRepresentation->RequestSetAxesObject( axesObject );
```

Next, we create a View3D to show our 3D scene.

```
typedef igstk::View3D  View3DType;
View3DType::Pointer view3D = View3DType::New();
```

Then we create a minimal FLTK GUI and display it.

```
typedef igstk::FLTKWidget      FLTKWidgetType;  
  
Fl_Window * form = new Fl_Window(600,600,"Circular Tracker");  
FLTKWidgetType * fltkWidget = new FLTKWidgetType(0,0,600,600,"FLTKWidget");  
  
fltkWidget->RequestSetView( view3D );  
  
// form->begin() is automatically called in the Fl_Window constructor.  
// form->end() : Add widgets after this point to the parent of the group.  
form->end();  
  
form->show();  
  
Fl::wait(0.5);
```

Now we make a simulated tracker and tracker tool. This way we can run the example without having to rely on having a particular tracker.

```
typedef igstk::CircularSimulatedTracker      TrackerType;  
typedef igstk::SimulatedTrackerTool        TrackerToolType;  
  
TrackerType::Pointer   tracker    = TrackerType::New();  
  
const double speedInDegreesPerSecond = 36.0;  
const double radiusInMillimeters = 2.0;  
  
tracker->RequestOpen();  
tracker->SetRadius( radiusInMillimeters );  
tracker->GetRadius(); // coverage  
tracker->SetAngularSpeed( speedInDegreesPerSecond );  
tracker->GetAngularSpeed(); // coverage;  
tracker->RequestSetFrequency( 100.0 );  
  
tracker->Print( std::cout );  
  
TrackerToolType::Pointer trackerTool = TrackerToolType::New();  
  
trackerTool->RequestSetName("Circle1");  
trackerTool->RequestConfigure();  
trackerTool->RequestAttachToTracker( tracker );
```

Now we will make a BoxObject and a BoxObjectRepresentation that will represent the TrackerTool in the scene. In a real application, one could use a CAD model of the actual tracker tool.

```

typedef igstk::BoxObject           ToolObjectType;
typedef igstk::BoxObjectRepresentation ToolRepresentationType;

ToolObjectType::Pointer toolObject = ToolObjectType::New();
toolObject->SetSize( 1.0, 1.0, 1.0 );

ToolRepresentationType::Pointer
    toolRepresentation = ToolRepresentationType::New();
toolRepresentation->RequestSetBoxObject( toolObject );
toolRepresentation->SetColor( 0.5, 1.0, 0.5 );

```

Here we build up the scene graph. We define an identity transform, just to make things simple.

Then, we attach the AxesObject to the tracker using the identity transformation. As a result the AxesObject will be displayed in tracker coordinates.

Next, we connect the object representing the tracker tool to the actual tracker tool. We use an identity transformation so that the toolObject is in the same position as the tracker tool.

```

igstk::Transform identity;
identity.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );

axesObject->RequestSetTransformAndParent( identity, tracker );
toolObject->RequestSetTransformAndParent( identity, trackerTool );

```

This next line differentiates the two examples, CoordinateSystems1.cxx (View From Tracker Coordinates Example) and CoordinateSystems2.cxx (View Follows Tracker Tool Example). Here, we view the scene from the tracker's perspective.

```
view3D->RequestSetTransformAndParent( identity, tracker );
```

Before we render, we also setup the view3D. We specify a refresh rate, background color, and camera parameters. Then we add the objects we wish to display.

```

view3D->SetRefreshRate( 30 );
view3D->SetRendererBackgroundColor( 0.8, 0.8, 0.9 );
view3D->SetCameraPosition( 10.0, 5.0, 3.0 );
view3D->SetCameraFocalPoint( 0.0, 0.0, 0.0 );
view3D->SetCameraViewUp( 0, 0, 1.0 );

view3D->RequestAddObject( axesRepresentation );
view3D->RequestAddObject( toolRepresentation );

```

Start the pulse generators for the view3D and tracker.

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

Now, run for a while.

```
for( unsigned int i = 0; i < 10000; i++ )
{
    Fl::wait(0.001);
    igstk::PulseGenerator::CheckTimeouts();
    Fl::check();
}
```

Finally, we cleanup before we exit.

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

tracker->RequestClose();

form->hide();

delete fltkWidget;
delete form;

return EXIT_SUCCESS;
```

8.7.3 View Follows Tracker Tool Example

One often-requested feature is a view into the image data from the perspective of the tool attached to a tracker. The new coordinate system design provides an easy way to implement this functionality. The main idea is to connect a `igstk::View` to a `igstk::TrackerTool`. The following example illustrates this construction.

The source code for this section can be found in the file
`Examples/CoordinateSystems/CoordinateSystems2.cxx`.

This example describes how to build a scene graph with a View, Tracker, and TrackerTool. This example will display the scene from the perspective of the TrackerTool.

First, we include the appropriate header files.

```
#include "igstkAxesObject.h"
#include "igstkAxesObjectRepresentation.h"
#include "igstkBoxObject.h"
#include "igstkBoxObjectRepresentation.h"
```

```
#include "igstkCylinderObject.h"
#include "igstkCylinderObjectRepresentation.h"
#include "igstkView3D.h"
#include "igstkFLTKWidget.h"
#include "igstkRealTimeClock.h"
#include "igstkSimulatedTrackerTool.h"
#include "igstkCircularSimulatedTracker.h"
```

Then we initialize the `RealTimeClock`. This is necessary to manage the timed events in IGSTK.

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

Now, we create an `AxesObject` with an `AxesObjectRepresentation` to provide a visual reference for the Tracker's coordinate system.

```
typedef igstk::AxesObject      AxesObjectType;
AxesObjectType::Pointer axesObject = AxesObjectType::New();
axesObject->SetSize(3.0, 3.0, 3.0);

typedef igstk::AxesObjectRepresentation  RepresentationType;
RepresentationType::Pointer axesRepresentation = RepresentationType::New();

axesRepresentation->RequestSetAxesObject( axesObject );
```

Next, we create a `View3D` to show our 3D scene.

```
typedef igstk::View3D   View3DType;
View3DType::Pointer view3D = View3DType::New();
```

Then we create a minimal FLTK GUI and display it.

```
typedef igstk::FLTKWidget      FLTKWidgetType;
FL_Window * form = new FL_Window(600,600,"Circular Tracker");
FLTKWidgetType * fltkWidget = new FLTKWidgetType(0,0,600,600,"FLTKWidget");

fltkWidget->RequestSetView( view3D );

// form->begin() is automatically called in the FL_Window constructor.
// form->end() : Add widgets after this point to the parent of the group.
form->end();

form->show();

Fl::wait(0.5);
```

Now we make a simulated tracker and tracker tool. This way we can run the example without having to rely on having a particular tracker.

```

typedef igstk::CircularSimulatedTracker      TrackerType;
typedef igstk::SimulatedTrackerTool         TrackerToolType;

TrackerType::Pointer   tracker     = TrackerType::New();

const double speedInDegreesPerSecond = 36.0;
const double radiusInMillimeters = 2.0;

tracker->RequestOpen();
tracker->SetRadius( radiusInMillimeters );
tracker->GetRadius(); // coverage
tracker->SetAngularSpeed( speedInDegreesPerSecond );
tracker->GetAngularSpeed(); // coverage;
tracker->RequestSetFrequency( 100.0 );

tracker->Print( std::cout );

TrackerToolType::Pointer  trackerTool = TrackerToolType::New();

trackerTool->RequestSetName("Circle1");
trackerTool->RequestConfigure();
trackerTool->RequestAttachToTracker( tracker );

```

Now we will make a BoxObject and a BoxObjectRepresentation that will represent the TrackerTool in the scene. In a real application, one could use a CAD model of the actual tracker tool.

```

typedef igstk::BoxObject           ToolObjectType;
typedef igstk::BoxObjectRepresentation ToolRepresentationType;

ToolObjectType::Pointer toolObject = ToolObjectType::New();
toolObject->SetSize( 1.0, 1.0, 1.0 );

ToolRepresentationType::Pointer
    toolRepresentation = ToolRepresentationType::New();
toolRepresentation->RequestSetBoxObject( toolObject );
toolRepresentation->SetColor( 0.5, 1.0, 0.5 );

```

Here we build up the scene graph. We define an identity transform, just to make things simple.

Then, we attach the AxesObject to the tracker using the identity transformation. As a result the AxesObject will be displayed in tracker coordinates.

Next, we connect the object representing the tracker tool to the actual tracker tool. We use an identity transformation so that the toolObject is in the same position as the tracker tool.

```
igstk::Transform identity;
identity.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );

axesObject->RequestSetTransformAndParent( identity, tracker );
toolObject->RequestSetTransformAndParent( identity, trackerTool );
```

This next line differentiates the two examples, `CoordinateSystems1.cxx` (View From Tracker Coordinates Example) and `CoordinateSystems2.cxx` (View Follows Tracker Tool Example). Here, we view the scene from the `trackerTool`'s perspective.

```
view3D->RequestSetTransformAndParent( identity, trackerTool );
```

Before we render, we also setup the `view3D`. We specify a refresh rate, background color, and camera parameters. Then we add the objects we wish to display.

```
view3D->SetRefreshRate( 30 );
view3D->SetRendererBackgroundColor( 0.8, 0.8, 0.9 );
view3D->SetCameraPosition( 10.0, 5.0, 3.0 );
view3D->SetCameraFocalPoint( 0.0, 0.0, 0.0 );
view3D->SetCameraViewUp( 0, 0, 1.0 );

view3D->RequestAddObject( axesRepresentation );
view3D->RequestAddObject( toolRepresentation );
```

Start the pulse generators for the `view3D` and `tracker`.

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

Now, run for a while.

```
for( unsigned int i = 0; i < 10000; i++ )
{
    F1::wait(0.001);
    igstk::PulseGenerator::CheckTimeouts();
    F1::check();
}
```

Finally, we cleanup before we exit.

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

tracker->RequestClose();

form->hide();

delete fltkWidget;
delete form;

return EXIT_SUCCESS;
```

8.7.4 "World" Coordinate System Example

In previous releases of IGSTK, applications were often written in terms of a "world" coordinate system. Developers, then, had to provide the transformations for each object into the world coordinate system. One downside to this approach is that it forces the developer to work in one coordinate frame. It is possible to still work in this way and it may be advisable for quick porting of applications written using the previous IGSTK API.

To mimic the previous behavior, the developer can create a node in the coordinate system scene graph that represents the previous world coordinate system. This node may be a `igstk::CoordinateSystem` object or one of the other objects with coordinate systems, such as a `igstk::Tracker`, `igstk::TrackerTool`, or `igstk::SpatialObject`. The following example illustrates this construction, using a `igstk::AxesObject` (a subclass of `igstk::SpatialObject`), to represent the world coordinates.

The source code for this section can be found in the file
`Examples/CoordinateSystems/CoordinateSystems3.cxx`.

This example describes how to build a scene graph with a View, Tracker, and TrackerTool. This example will place all the objects in a developer specified world coordinate system.

First, we include the appropriate header files.

```
#include "igstkAxesObject.h"
#include "igstkAxesObjectRepresentation.h"
#include "igstkBoxObject.h"
#include "igstkBoxObjectRepresentation.h"
#include "igstkCylinderObject.h"
#include "igstkCylinderObjectRepresentation.h"
#include "igstkView3D.h"
#include "igstkFLTKWidget.h"
#include "igstkRealTimeClock.h"
#include "igstkSimulatedTrackerTool.h"
#include "igstkCircularSimulatedTracker.h"
```

Then we initialize the RealTimeClock. This is necessary to manage the timed events in IGSTK.

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

Here we create an AxesObject to function as the world coordinate system. We do not create a representation since we will not be displaying the world coordinate system object. The worldCoordinateSystem just provides a placeholder node in the scene graph.

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

Now, we create an AxesObject with an AxesObjectRepresentation to provide a visual reference for the Tracker's coordinate system.

```
AxesObjectType::Pointer axesObject = AxesObjectType::New();
axesObject->SetSize(3.0, 3.0, 3.0);

typedef igstk::AxesObjectRepresentation  RepresentationType;
RepresentationType::Pointer axesRepresentation = RepresentationType::New();

axesRepresentation->RequestSetAxesObject( axesObject );
```

Next, we create a View3D to show our 3D scene.

```
typedef igstk::View3D  View3DType;

View3DType::Pointer view3D = View3DType::New();
```

Then we create a minimal FLTK GUI and display it.

```
typedef igstk::FLTKWidget      FLTKWidgetType;

Fl_Window * form = new Fl_Window(600,600,"Circular Tracker");
FLTKWidgetType * fltkWidget = new FLTKWidgetType(0,0,600,600,"FLTKWidget");

fltkWidget->RequestSetView( view3D );

// form->begin() is automatically called in the Fl_Window constructor.
// form->end() : Add widgets after this point to the parent of the group.
form->end();

form->show();

Fl::wait(0.5);
```

Now we make a simulated tracker and tracker tool. This way we can run the example without having to rely on having a particular tracker.

```

typedef igstk::CircularSimulatedTracker      TrackerType;
typedef igstk::SimulatedTrackerTool         TrackerToolType;

TrackerType::Pointer   tracker   = TrackerType::New();

const double speedInDegreesPerSecond = 36.0;
const double radiusInMillimeters = 2.0;

tracker->RequestOpen();
tracker->SetRadius( radiusInMillimeters );
tracker->GetRadius(); // coverage
tracker->SetAngularSpeed( speedInDegreesPerSecond );
tracker->GetAngularSpeed(); // coverage;
tracker->RequestSetFrequency( 100.0 );

tracker->Print( std::cout );

TrackerToolType::Pointer trackerTool = TrackerToolType::New();

trackerTool->RequestSetName("Circle1");
trackerTool->RequestConfigure();
trackerTool->RequestAttachToTracker( tracker );

```

Now we will make a BoxObject and a BoxObjectRepresentation that will represent the TrackerTool in the scene. In a real application, one could use a CAD model of the actual tracker tool.

```

typedef igstk::BoxObject           ToolObjectType;
typedef igstk::BoxObjectRepresentation ToolRepresentationType;

ToolObjectType::Pointer toolObject = ToolObjectType::New();
toolObject->SetSize( 1.0, 1.0, 1.0 );

ToolRepresentationType::Pointer
    toolRepresentation = ToolRepresentationType::New();
toolRepresentation->RequestSetBoxObject( toolObject );
toolRepresentation->SetColor( 0.5, 1.0, 0.5 );

```

Here we build up the scene graph. We define an identity transform, just to make things simple. Then, we attach the AxesObject to the tracker using the identity transformation. As a result the AxesObject will be in the same position as the tracker.

Next, we connect the object representing the tracker tool to the actual tracker tool. We use an identity transformation so that the toolObject is in the same position as the tracker tool.

```
igstk::Transform identity;
```

```
identity.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );

axesObject->RequestSetTransformAndParent( identity, tracker );
toolObject->RequestSetTransformAndParent( identity, trackerTool );
```

These next lines are key differences from CoordinateSystems1.cxx (View From Tracker Coordinates Example) and CoordinateSystems2.cxx (View Follows Tracker Tool Example). In this example, we place the view and the tracker into world coordinates using an identity transformation. In a real-world application, the transformation would not be the trivial identity transform.

```
tracker->RequestSetTransformAndParent( identity, worldCoordinateSystem );
view3D->RequestSetTransformAndParent( identity, worldCoordinateSystem );
```

Before we render, we also setup the view3D. We specify a refresh rate, background color, and camera parameters. Then we add the objects we wish to display.

```
view3D->SetRefreshRate( 30 );
view3D->SetRendererBackgroundColor( 0.8, 0.8, 0.9 );
view3D->SetCameraPosition( 10.0, 5.0, 3.0 );
view3D->SetCameraFocalPoint( 0.0, 0.0, 0.0 );
view3D->SetCameraViewUp( 0, 0, 1.0 );

view3D->RequestAddObject( axesRepresentation );
view3D->RequestAddObject( toolRepresentation );
```

Start the pulse generators for the view3D and tracker.

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

Now, run for a while.

```
for( unsigned int i = 0; i < 10000; i++ )
{
    Fl::wait(0.001);
    igstk::PulseGenerator::CheckTimeouts();
    Fl::check();
}
```

Finally, we cleanup before we exit.

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

tracker->RequestClose();

form->hide();

delete fltkWidget;
delete form;

return EXIT_SUCCESS;
```


Tracker

“The trouble with measurement is its seeming simplicity.”

—Anonymous

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 the same features and locations on (and inside of) a patient’s body during surgery. In the simplest case, this process can be purely visual; such as when a surgeon uses x-ray fluoroscopy to look inside a patient during surgery. However, 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 most other modern image-guided surgery systems, the third action is achieved by tracking the positions of the surgeon’s tools and superimposing the tools on the images of the patient, so that the surgeon can see the positions of the tools relative to the target location.

Surgical tools are tracked 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 many times per second to make it possible to track the motion of the tool. Several models of tracking systems are available on the market. The trackers that are supported by the current version of IGSTK are the NDI POLARIS family of optical tracking systems, the NDI AURORA magnetic tracking system, the Micron Tracker, and the Ascension Flock of Birds magnetic tracking system. We anticipate adding support for additional trackers in the future and welcome user contributions in this regard.

9.1 The Role of the Tracker Component in IGSTK

The IGSTK Tracker component communicates with a tracking system to get position measurements, 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 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 possible.

9.2 Structure of the Tracker Component

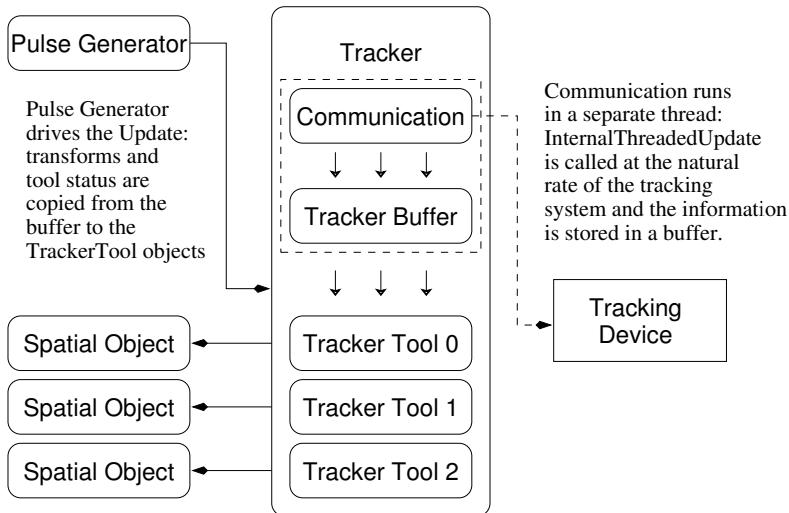


Figure 9.1: The IGSTK Tracker Component.

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 IGSTK software components, but the tracking system is a physical device located in the operating room. The tracked tools are surgical tools held by the clinician. The updating of the tool position and status information in the software occurs because there is a communication link, such as a serial or network connection, between the computer and the tracking system. The interface to this link is provided by an IGSTK `Communication` object, and each `Tracker` object has a `Communication` object that provides exclusive access to the tracking system. No other IGSTK components can access the tracking system except via the `Tracker` object.

The tracking device has its own processor and has a specific “frame rate,” which is the rate at which it makes tool position measurements (typically somewhere between 15 Hz and 100 Hz).

The IGSTK software components run as a synchronous system, with each component executing in turn, and there are no guarantees that the processing and rendering that correspond to one position measurement will be complete before the next position measurement is made. Furthermore, 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 that is completely independent of anything going on in the software; therefore, it was decided that the IGSTK Tracker component should spawn a separate thread to communicate with the tracking device. This thread runs an IGSTK Communication object asynchronously from the rest of IGSTK - that is, this Communication object is able to talk to the tracking device even when other IGSTK objects are busy.

It is crucial to note that most of the IGSTK Tracker component code (including the state machine and the entire public interface) is executed exclusively within the main IGSTK application thread. Only the code that uses the Communication object is run within the tracker communication thread.

These data records are not used immediately as they are received; rather, they are buffered until the application is ready for them - that is, 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 changes only at times when other IGSTK components are expecting it to change.

9.2.1 Communication

Collecting data from the tracking system is one of the two fundamental duties of the Tracker (the other duty is relaying this data to the application). Early in the development of IGSTK, it was decided that communication between the tracker component and the tracking system should be handled by an `igstk::Communication` class that meets the following 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 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 fourth requirements facilitate the testing of the tracker component and the tracking system. In 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 during testing to simulate the presence of a tracking device. 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.

9.2.2 Threading

As stated in Section 9.2, 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, in which the device will physically resist any motion in certain directions.

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, then 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`, 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, performance criteria such as frame rate and lag, which are so important to 3D video game enthusiasts, are of some consequence in image-guided 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 measurement back.

By contrast, the IGSTK Tracker has been implemented such that the tracking system continuously streams position measurements to it and uses its extra thread to listen for each of these measurements and to 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.

9.2.3 Buffering

In the previous section, we described how threading and buffering are used to improve the performance of IGSTK. In future versions of IGSTK, the buffer will store not only the most recent tool positions, but a history of all position measurements that are made by the tracking system during the surgery. A diagram of the tracker buffer is shown in Figure 9.2.

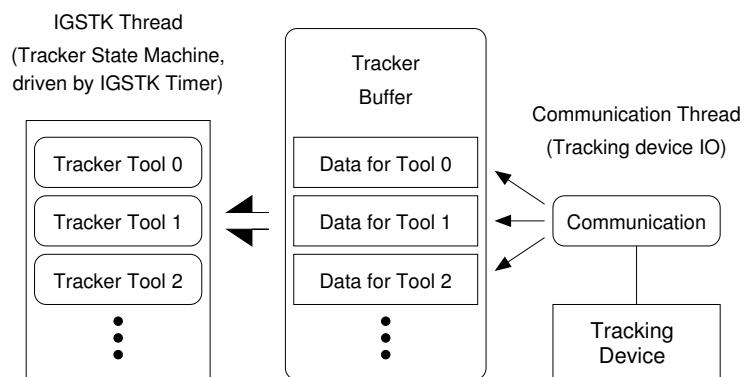


Figure 9.2: The Tracker Buffer.

When IGSTK displays a surgical scene with the positions of all of the surgical tools superimposed on the medical images, it is displaying the location of those tools at a particular instant in time. That instant in time is the latest instant for which valid data is available for all tools, i.e. the latest instant for which the tracker buffer holds a valid transform for each tool.

In a future version of IGSTK, the tracker buffer will store the position history for each tool, and it will be possible to determine where a tool was at any time in the past after the instant when the

tracker began tracking the tools. This feature will make it possible to replay the positions of the tools, and more importantly, it will allow for accurate synchronization of the tool positions with video data. For example, if every video frame (generated by a planned “video” component for IGSTK) has a timestamp, it will be possible to search the tracker buffer to get the tool positions that best match the video frame’s timestamp and use those positions to update the display.

Since the TrackerBuffer contains data that is shared between the tracking thread (within which the Tracker places 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.

9.2.4 Coordinate Transformations

The Tracker component applies transformations to the position measurements (raw data) read from the tracking device before passing it to the rest of IGSTK components. The following formula will be applied to the raw transform data:

$$T_{\text{comp}} = T_{\text{ref}}^{-1} T_{\text{raw}} M_{\text{cal}} \quad (9.1)$$

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_{comp}	tool transform that will be used by other IGSTK components
T_{raw}	raw transform data from the tracking system
T_{ref}^{-1}	raw transform data from a tracked reference (if present)
M_{cal}	tool calibration transform (Identity transform by default)

The tracking component returns the composed transform (T_{comp}).

The *reference* is a 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 - 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 calibration transformation is used as a correction for the tip location and shaft orientation for a particular tool. This is discussed in great detail in Chapter 16.

Figure 9.3 shows typical coordinate systems that are involved in image-guided surgery application. The transformation that we want is T , which relates the position and orientation of the tool tip to the patient frame of reference. For this we need patient registration transform (M_{reg}). 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. This is discussed in detail in Chapter 15.

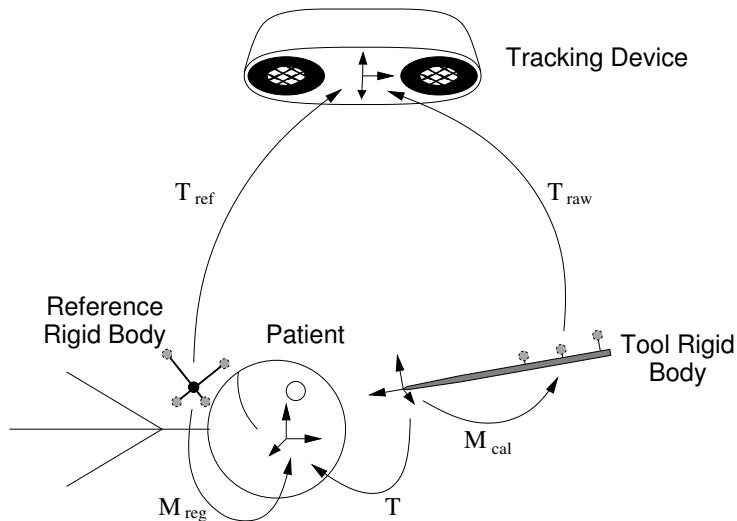


Figure 9.3: Tracker Coordinate Transformations.

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

$$\mathbf{I} = \mathbf{T}^{-1} \mathbf{M}_{\text{reg}} \mathbf{T}_{\text{ref}}^{-1} \mathbf{T}_{\text{raw}} \mathbf{M}_{\text{cal}} \quad (9.2)$$

$$\mathbf{I} = \mathbf{T}^{-1} \mathbf{M}_{\text{reg}} \mathbf{T}_{\text{comp}} \quad (9.3)$$

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

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

Figure 9.3 is drawn for an optical tracking system, which uses a camera that tracks sets of 3 or 4 markers that define a “rigid body,” or 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.

9.3 Class Hierarchy

The tracker component contains two main superclasses i.e `igstk::Tracker` and `igstk::TrackerTool`. Vendor-specific classes are derived from these classes. Figure 9.4 and 9.5 show the class hierarchy diagram for the tracker and tracker tool classes.

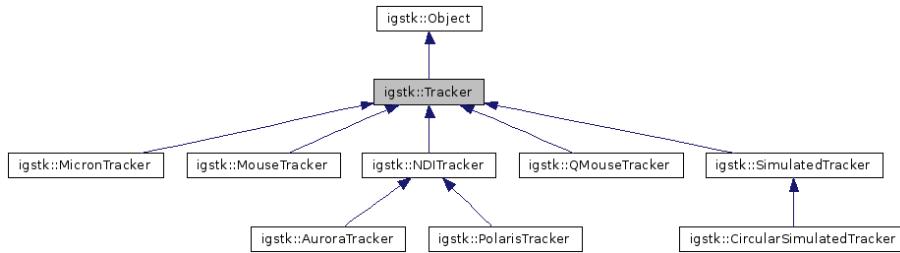


Figure 9.4: Tracker Class Hierarchy.

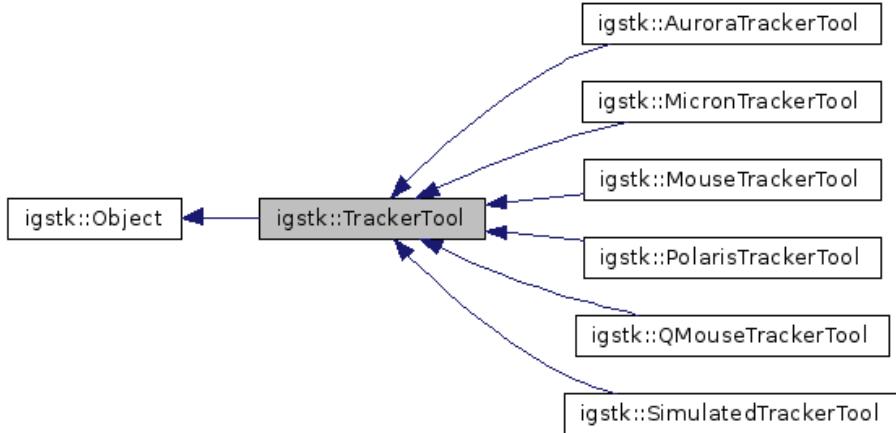


Figure 9.5: Tracker Tool Class Hierarchy.

In the next sections, state machine configuration, public interface methods and events for `igstk::Tracker` and `igstk::TrackerTool` classes are presented.

9.3.1 Tracker

State Machine

Figure 9.6 illustrates the state machine diagram of the `igstk::Tracker`.

The major states are as follows:

1. *Idle* : Initial state. The Tracker will return to this state after the `RequestClose()` method is called.
2. *TrackerToolAttached*: Tracker tool is attached to the tracker.
3. *CommunicationEstablished* : Communication is established. This state is entered if a call to the Tracker's `RequestOpen()` was successful. The tracker can be returned to this state from any state except the *Idle* state by calling the `RequestReset()` method.
4. *Tracking* : Tracking component is ready to send position data. This state is entered if a call to `RequestStartTracking()` was successful.

The following are transitional states, which occur when the tracker class is waiting to see if a particular request has been fulfilled:

1. *AttemptingToEstablishCommunication* : Waiting for the communication link to be opened to device.
2. *AttemptingToCloseCommunication* : Waiting for the communication link to close.
3. *AttemptingToAttachTrackerTool* : Waiting for the tracker tool to be attached to the tracker.
4. *AttemptingToTrack* : Waiting to make a transition to tracking state.
5. *AttemptingToStopTracking* : Waiting till tracking is stopped.
6. *AttemptingToUpdate* : Checking for new data records.

Upon transitioning out of one of these “attempt” states, the Tracker will generate an event that will indicate whether the attempt succeeded or failed.

Interface Methods

The public interface to the tracker component consists of the following methods:

1. `RequestOpen()` : Initiate communication with the device.
2. `RequestClose()` : Close communication with the device.

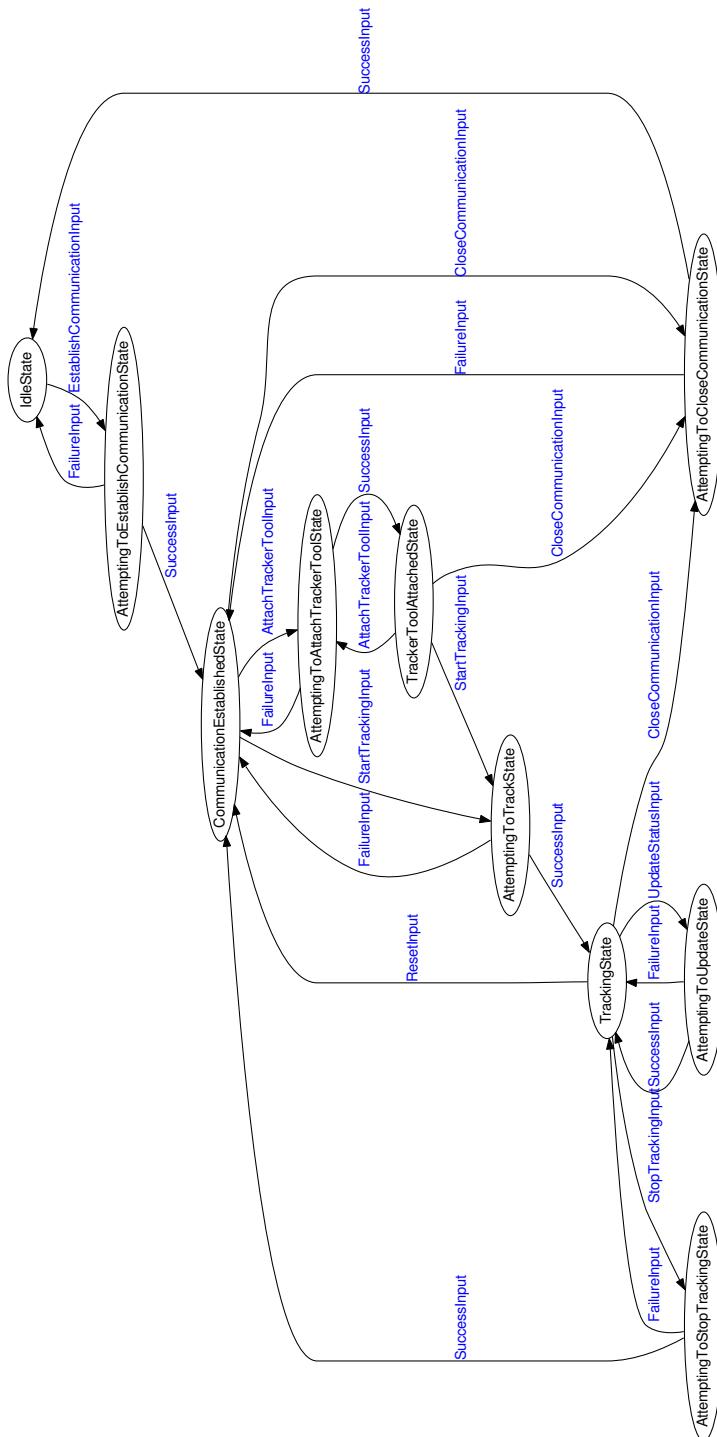


Figure 9.6: State Machine Diagram of the Tracker Component.

3. *RequestReset()* : Attempt to reset the device to its initial state.
4. *RequestStartTracking()* : Put the device into its tracking state, from which it will generate tracking data for each tool. In this state, the tracker buffer will be continually updated with new data from the tracking device.
5. *RequestStopTracking()*: Stop tracker tool transform gathering
6. *RequestSetFrequency()*: Set the frequency at which the transform information will be queried from the tracking device.
7. *RequestSetReferenceTool()*: Set the tracker tool that will be used as a reference. This reference tool has to be first attached to the tracker similar to the other tracker tools.

Events

The following events are generated by the Tracker object.

1. *TrackerOpenEvent* : Generated when a call to *RequestOpen()* was successful.
2. *TrackerOpenErrorEvent* : Generated when a call to *RequestOpen()* failed to initiate communication with the device.
3. *TrackerCloseEvent* : Generated when a call to *RequestClose()* was successful.
4. *TrackerCloseErrorEvent* : Generated when a call to *RequestClose()* failed to close communication with the device.
5. *TrackerInitializeEvent* : Generated when a call to *RequestInitialize()* was successful.
6. *TrackerInitializeErrorEvent* : Generated when a call to *RequestInitialize()* failed.
7. *TrackerStartTrackingEvent* : Generated when a call to *RequestStartTracking()* was successful.
8. *TrackerStartTrackingErrorEvent* : Generated when a call to *RequestStartTracking()* failed.
9. *TrackerStopTrackingEvent* : Generated when a call to *RequestStopTracking()* was successful.
10. *TrackerStopTrackingErrorEvent* : Generated when a call to *RequestStopTracking()* failed.
11. *TrackerUpdateStatusEvent* : Generated when a call to *RequestUpdateStatus()* was successful.
12. *TrackerUpdateStatusErrorEvent* : Generated when a call to *RequestUpdateStatus()* failed.

13. *TrackerEvent* : Superclass of all events generated by the Tracker for successful request completions.
14. *TrackerErrorEvent* : Superclass of all events that are generated by the Tracker as a result of a failed request or other error.

9.3.2 Tracker Tool

State Machine

Figure 9.7 illustrates the State Machine Diagram of the `igstk::TrackerTool`.

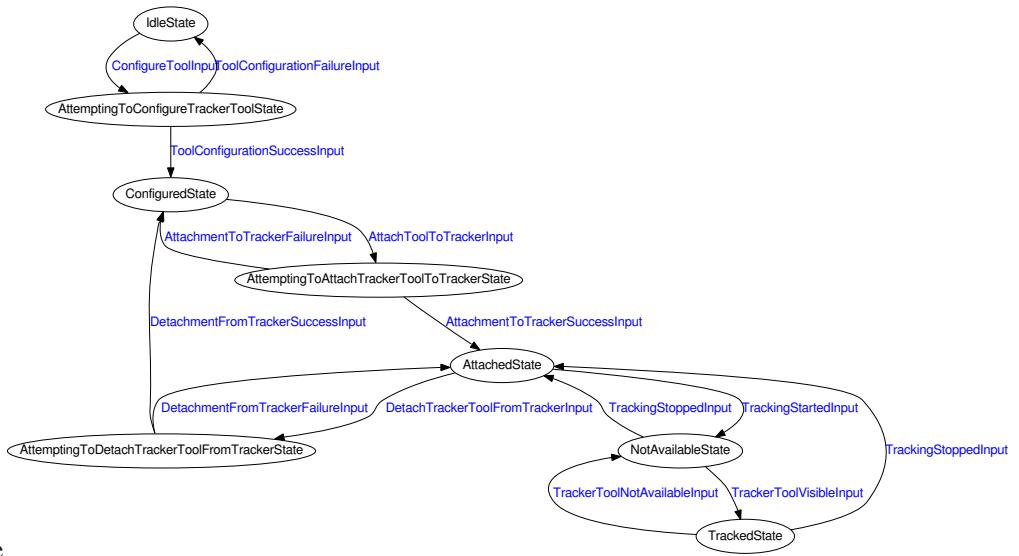


Figure 9.7: State Machine Diagram of the `TrackerTool` Class.

The major states are as follows:

1. *Idle* : Initial state.
2. *Configured*: Tracker tool ready for attachment. After all the required parameters are set for the tracker tool, the tracker tool make the transition to this state.
3. *Attached*: Tracker tool attached to the tracker
4. *NotAvailable*: Tracker tool is not available to be tracked. The tool could be outside the field of measurement of the tracking device.

5. *Tracked*: The tool is being tracked by the tracker class.

The following are transitional states for the tracker tool class.

1. *AttemptingToConfigureTrackerTool*: Waiting until the tracker tool is configured. The tool can be attached to the tracker only after configuration.
2. *AttemptingToAttachTrackerTool*: Waiting until the tracker tool is attached to the tracker
3. *AttemptingToDetachTrackerToolFromTracker*: Waiting until the tracker tool is detached from the tracker.

Interface Methods

1. *SetCalibrationTransform()*: Set the calibration transform for the tool (Identity transform by default).
2. *GetCalibratedTransform()*: Get the calibration transform for the tool.
3. *RequestConfigure()*: Request the tracker tool to be configured.
4. *RequestAttachToTracker()*: Request the tracker tool to be attached to the tracker.
5. *RequestDetachFromTracker()*: Request the tracker tool to be disconnected from the tracker.
6. *GetTrackerToolIdentifier()*: Get the unique tracker tool identifier auto-generated by the tracker tool class.

Events

The following events are generated by the Tracker object.

1. *TrackerToolConfigurationEvent*: Generated when a call to *RequestConfigure()* method is successful.
2. *TrackerToolConfigurationErrorEvent* : Generated when a call to *RequestConfigure()* method fails.
3. *InvalidRequestToAttachTrackerToolErrorEvent* : Generated when a call to *RequestAttachToTracker*: is made when the tracker tool is not yet configured.
4. *InvalidRequestToDetachTrackerToolErrorEvent* : Generated when a call to *RequestDetachFromTracker*: is made when the tracker tool is not attached to the tracker.
5. *TrackerToolAttachmentToTrackerEvent* : Generated when a call to *RequestAttachToTracker* is successful.

6. *TrackerToolAttachmentToTrackerErrorEvent* : Generated when a call to *RequestAttachmentToTracker* fails.
7. *TrackerToolMadeTransitionToTrackedStateEvent* : Generated when tracker starts tracking the tool.
8. *TrackerToolNotAvailableToBeTrackedEvent* : Generated when the tool is not being tracked by the tracker. The tool could be outside the tracking devices's field of measurement.
9. *ToolTrackingStartedEvent* : Generated when a call to *RequestReportTrackingStarted* method is invoked.
10. *ToolTrackingStoppedEvent* : Generated when a call to *RequestReportTrackingStopped* method is invoked.
11. *TrackerToolEvent* : Superclass of all events generated by Tracker tool class.
12. *TrackerToolErrorEvent* : Superclass of all error events generated by Tracker tool class.

9.4 Simulation and Testing

Tracking devices pose a unique challenge for the automated nightly regression testing that is done as part of the IGSTK development process. It is not possible to connect a physical tracking device to each of the computers on which the nightly tests are run, and even if it were possible, a method of moving the tools around during the testing would have to be devised.

The solution to this conundrum is as follows. Each tracking test is run once with a tracking device attached to the computer and with the tracked tools in motion. During this interactive test, the communication log with the tracking device is recorded and is then saved as a simulation file. This simulation file becomes part of the testing data, and the subsequent nightly tests replay the data rather than communicating with an actual tracking device.

The IGSTK class that replays the data is `igstk::SerialCommunicationSimulator`. This class is a subclass of the `igstk::SerialCommunication` that is actually used to talk to the device.

9.5 Hazardous Conditions

Since tracking systems are such important components of image-guided systems, we should take some time to look at 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.

9.5.1 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) if 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

9.5.2 Loss of Accuracy

The following hazards can lead to inaccurate 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 ask the user to 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 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

9.6 Example

The following two examples illustrate how to interface the IGSTK tracker component with MicronTracker and Polaris NDI tracking devices.

9.6.1 MicronTracker Example

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

This example illustrates how to use IGSTK's Tracker component to communicate with Micron Tracker and gather tool transform information. MicronTracker is a portable optical tracker that uses video image analysis techniques for localization and tracking of surgical tools.

To communicate with MicronTracker tracking device, include the MicronTracker header files
`igstk::MicronTracker` and `igstk::MicronTrackerTool`

```
#include "igstkMicronTracker.h"
#include "igstkMicronTrackerTool.h"
```

Define a callback class to listen to events invoked by the tracker class.

```
class MicronTrackerCommand : public itk::Command
{
public:
    typedef MicronTrackerCommand    Self;
    typedef itk::Command           Superclass;
```

```
typedef itk::SmartPointer<Self> Pointer;
itkNewMacro( Self );
protected:
    MicronTrackerTrackerCommand() {};

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

    void Execute(const itk::Object * object, const itk::EventObject & event)
    {
        // don't print "CompletedEvent", only print interesting events
if (!igstk::CompletedEvent().CheckEvent(&event) &&
    !itk::DeleteEvent().CheckEvent(&event) )
{
    std::cout << event.GetEventName() << std::endl;
}
    }
};
```

Instantiate tracker command callback object.

```
MicronTrackerTrackerCommand::Pointer
my_command = MicronTrackerTrackerCommand::New();
```

Next, set the necessary parameters for the tracker. The required parameters are directory that contains camera calibration files, micron tracker initialization file (.ini) and marker template directory.

```
std::string calibrationFilesDirectory = argv[1];
tracker->SetCameraCalibrationFilesDirectory(
    calibrationFilesDirectory );

std::string initializationFile = argv[2];
tracker->SetInitializationFile( initializationFile );

std::string markerTemplateDirectory = argv[3];
tracker->SetMarkerTemplatesDirectory( markerTemplateDirectory );
```

Next, request tracker communication be opened.

```
tracker->RequestOpen();
```

Next, add tracker tools to the tracker.

```

typedef igstk::MicronTrackerTool  TrackerToolType;

TrackerToolType::Pointer trackerTool = TrackerToolType::New();
trackerTool->SetLogger( logger );
std::string markerNameTT = "TTblock";
trackerTool->RequestSetMarkerName( markerNameTT );
trackerTool->RequestConfigure();
trackerTool->RequestAttachToTracker( tracker );
trackerTool->AddObserver( itk::AnyEvent(), my_command );
//Add observer to listen to transform events
ObserverType::Pointer coordSystemAObserver = ObserverType::New();
coordSystemAObserver->ObserveTransformEventsFrom( trackerTool );

TrackerToolType::Pointer trackerTool2 = TrackerToolType::New();
trackerTool2->SetLogger( logger );
std::string markerNamesPointer = "sPointer";
trackerTool2->RequestSetMarkerName( markerNamesPointer );
trackerTool2->RequestConfigure();
trackerTool2->RequestAttachToTracker( tracker );
trackerTool2->AddObserver( itk::AnyEvent(), my_command );
ObserverType::Pointer coordSystemAObserver2 = ObserverType::New();
coordSystemAObserver2->ObserveTransformEventsFrom( trackerTool2 );

```

Start tracking and gather the tracker tool transforms.

```

tracker->RequestStartTracking();

typedef igstk::Transform           TransformType;
typedef ::itk::Vector<double, 3>   VectorType;
typedef ::itk::Versor<double>      VersonType;

for(unsigned int i=0; i<100; i++)
{
    igstk::PulseGenerator::CheckTimeouts();

    TransformType          transform;
    VectorType             position;

    coordSystemAObserver->Clear();
    trackerTool->RequestGetTransformToParent();
    if (coordSystemAObserver->GotTransform())
    {
        transform = coordSystemAObserver->GetTransform();
        if ( transform.IsValidNow() )
        {
            position = transform.GetTranslation();
            std::cout << "Trackertool :"
                << trackerTool->GetTrackerToolIdentifier()

```

```
<< "\t\t Position = (" << position[0]
<< "," << position[1] << "," << position[2]
<< ")" << std::endl;
    }
}

coordSystemAObserver2->Clear();
trackerTool2->RequestGetTransformToParent();
if (coordSystemAObserver2->GotTransform())
{
    transform = coordSystemAObserver2->GetTransform();
    if ( transform.IsValidNow() )
    {
        position = transform.GetTranslation();
        std::cout << "Trackertool2 :"
                << trackerTool2->GetTrackerToolIdentifier()
                << "\t\t Position = (" << position[0]
                << "," << position[1] << "," << position[2]
                << ")" << std::endl;
    }
}
}
```

Stop tracking and close the communication with the tracker.

```
std::cout << "RequestStopTracking()" << std::endl;
tracker->RequestStopTracking();

std::cout << "RequestClose()" << std::endl;
tracker->RequestClose();
```

9.6.2 Polaris Tracker Example

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

This example illustrates IGSTK's interface to Polaris NDI tracker. Polaris trackers are optical measurement system that measure the 3D positions of either active or passive markers.

To communicate with Polaris tracking device, include the Polaris Tracker header files (`igstk::PolarisTracker`) and (`igstk::PolarisTrackerTool`).

```
#include "igstkPolarisTracker.h"
#include "igstkPolarisTrackerTool.h"
```

Instantiate serial communication object and set the communication parameters.

```

typedef igstk::SerialCommunication::PortNumberType PortNumberType;
unsigned int portNumberIntegerValue = atoi(argv[3]);
PortNumberType polarisPortNumber = PortNumberType(portNumberIntegerValue);
serialComm->SetPortNumber( polarisPortNumber );
serialComm->SetParity( igstk::SerialCommunication::NoParity );
serialComm->SetBaudRate( igstk::SerialCommunication::BaudRate115200 );
serialComm->SetDataBits( igstk::SerialCommunication::DataBits8 );
serialComm->SetStopBits( igstk::SerialCommunication::StopBits1 );
serialComm->SetHardwareHandshake( igstk::SerialCommunication::HandshakeOff );

serialComm->SetCaptureFileName( "RecordedStreamByPolarisTracker.txt" );
serialComm->SetCapture( true );

```

Wired and wireless tools can be handled by the Polaris Tracker class. For wired tracker tool type, invoke RequestSelectWirelessTrackerTool method.

```
trackerTool->RequestSelectWiredTrackerTool();
```

For wireless tracker tool type, invoke RequestSelectWirelessTrackerTool() method and set SROM file.

```

trackerTool2->RequestSelectWirelessTrackerTool();
//Set the SROM file
std::string romFile = argv[2];
std::cout << "SROM file: " << romFile << std::endl;
trackerTool2->RequestSetsROMFileName( romFile );

```

After configuring the tracker tool, make a request to attach the tracker tool to the tracker.

```
trackerTool2->RequestAttachToTracker( tracker );
```

Start tracking and observe tracker tool pose information.

```

tracker->RequestStartTracking();

typedef igstk::Transform           TransformType;
typedef ::itk::Vector<double, 3>    VectorType;
typedef ::itk::Versor<double>       VersonType;

```

```

for(unsigned int i=0; i<100; i++)
{
  igstk::PulseGenerator::CheckTimeouts();

  TransformType          transform;
  VectorType             position;

```

```
coordSystemAObserver->Clear();
trackerTool->RequestGetTransformToParent();
if (coordSystemAObserver->GotTransform())
{
    transform = coordSystemAObserver->GetTransform();
    if (transform.IsValidNow() )
    {
        position = transform.GetTranslation();
        std::cout << "Trackertool :"
            << trackerTool->GetTrackerToolIdentifier()
            << "\t\t Position = (" << position[0]
            << "," << position[1] << "," << position[2]
            << ")" << std::endl;
    }
}

coordSystemAObserver2->Clear();
trackerTool2->RequestGetTransformToParent();
if (coordSystemAObserver2->GotTransform())
{
    transform = coordSystemAObserver2->GetTransform();
    if (transform.IsValidNow() )
    {
        position = transform.GetTranslation();
        std::cout << "Trackertool2:"
            << trackerTool2->GetTrackerToolIdentifier()
            << "\t\t Position = (" << position[0]
            << "," << position[1] << "," << position[2]
            << ")" << std::endl;
    }
}
```

To end the tracking process, stop and close the tracker and close the serial communication channel.

```
std::cout << "RequestStopTracking()" << std::endl;
tracker->RequestStopTracking();

std::cout << "RequestClose()" << std::endl;
tracker->RequestClose();

std::cout << "CloseCommunication()" << std::endl;
serialComm->CloseCommunication();
```

9.7 Conclusion

Tracking is a fundamental part of an image-guided surgery procedure. The IGSTK Tracker component allows for communication with a tracking device, as well as for the replaying of a previous session for testing purposes.

One unique aspect of the Tracker component, as compared to other IGSTK components, is that it is multi-threaded. A tracker communication thread is created whenever the tracker transitions to its Tracking state, and this thread listens for tracking data from the device, logs the data, and places it in a buffer. The main tracker thread (i.e., the IGSTK application thread of execution) reads the data from this buffer.

Spatial Objects

Spatial objects define a common data structure for objects in IGSTK. The spatial object class hierarchy provides a consistent API for querying, manipulating, and interconnecting objects in physical space. The base spatial object class encapsulates an ITK spatial object; however, only functionalities that are essential for image-guided surgery applications are exposed to the user. A spatial object is a data structure describing the geometry of an object. Each spatial object contains an internal transformation that defines its location and orientation in space. In this chapter, we first describe how spatial objects 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 spatial objects from disk.

10.1 Spatial Object Aggregation Hierarchy

Spatial objects can be composed together to create higher order structures. The structure forms a directed acyclic graph (DAG), in which a child can only have one parent. Next we describe how to create a simple object tree.

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

This example describes how to group `igstk::SpatialObject`s to form an aggregate 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, `sphere1` and `sphere2` using the `igstk::EllipsoidObject` class. They are created using smart pointers, as follows:

```
typedef igstk::EllipsoidObject SpatialObjectType;
```

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

We create a transformation between `sphere1` and `sphere2` and set the translation vector to be *10mm* in each direction, with an error value of *0.001mm* and a validity time of *10ms*. Then we attach `sphere2` as a child of `sphere1` and we set the relative transform between the two objects via the `RequestSetTransformAndParent()` function. The commands appear as follows:

```
igstk::Transform transformSphere2ToSphere1;
transformSphere2ToSphere1.SetTranslation(10,0.001,10);
sphere2->RequestSetTransformAndParent( transformSphere2ToSphere1, sphere1 );
```

In order to retrieve a transformation between an object and its parent, an observer should be created using the `igstkObserverMacro`. The first parameter of the macro is the name of the observer, the second is the type of event to observe and the third parameter is the expected result type. Then we instantiate the observer using smart pointers.

```
igstkObserverMacro(TransformToParent,
                   ::igstk::CoordinateSystemTransformToEvent,
                   ::igstk::CoordinateSystemTransformToResult)
```

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:

1. `GotTransformToParent()` which returns true if the transform exists.
2. `GetTransformToParent()` which returns the actual pointer to the transform.

We create the observer of the transformation using smart pointers.

```
TransformToParentObserver::Pointer transformToParentObserver
    = TransformToParentObserver::New();
```

We add the observer to the object using the `AddObserver()` command. The first argument specifies the type of event to observe and the second argument is the observer instantiated previously.

```
sphere2->AddObserver( ::igstk::CoordinateSystemTransformToEvent(),
                      transformToParentObserver );
```

Then, we request the transform using the `RequestGetTransformToParent()` command:

```
sphere2->RequestGetTransformToParent();
if( !transformToParentObserver->GotTransformToParent() )
{
    std::cerr << "Sphere1 did not returned a Transform event" << std::endl;
    return EXIT_FAILURE;
}

igstk::Transform transform2 = transformToParentObserver
    ->GetTransformToParent().GetTransform();
```

Next, we introduce the `igstk::GroupObject`. The `igstk::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 definitions and smart pointers, as follows:

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

Since the `igstk::GroupObject` derives from `igstk::SpatialObject`, we can use the `RequestAddChild()` function to add object into the group. This function expects the relative transform between the two objects as the first argument. For instance, we group the previously created `sphere1` and the newly created `sphere3` together, as follows:

```
igstk::Transform transformSphere1ToGroup;
transformSphere1ToGroup.SetToIdentity(
    igstk::TimeStamp::GetLongestPossibleTime() );
group->RequestAddChild( transformSphere1ToGroup, sphere1 );
SpatialObjectType::Pointer sphere3 = SpatialObjectType::New();

igstk::Transform transformSphere3ToGroup;
transformSphere3ToGroup.SetToIdentity(
    igstk::TimeStamp::GetLongestPossibleTime() );
group->RequestAddChild( transformSphere3ToGroup, sphere3 );
```

We can request the number of objects in the group using the `GetNumberOfChildren()` function. This function returns only the number of first level children. In this case we have two children in the group, even if `sphere2` is a child of `sphere1`.

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

10.2 State Machine

Like all IGSTK components, spatial objects are governed by a state machine. The state diagram is shown in Figure 10.1.

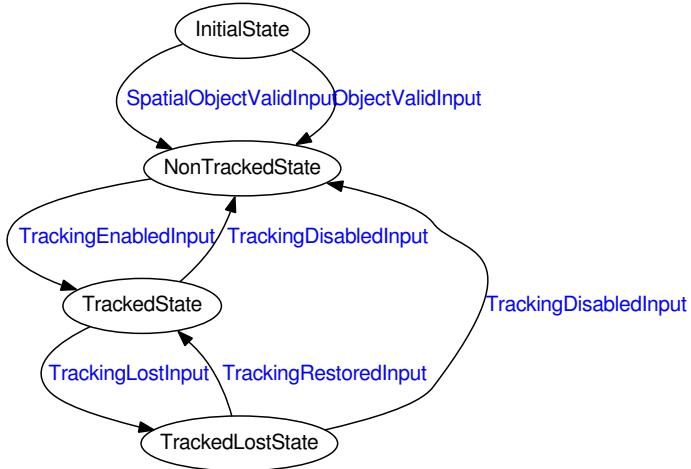


Figure 10.1: Spatial Object State Machine.

The state machine has the following states:

1. *InitialState*: Sets the initial state of the spatial object.
2. *NonTrackedState*: The spatial object is not being tracked via a tracker tool.
3. *TrackedState*: The spatial object is now being tracked.
4. *TrackedLostState*: Tracking information was not reported on the last pulse. This is distinct from NonTrackedState, as it indicates the expectation that tracking will be restored (TrackingRestoredInput).

10.3 Component Interface

The following methods are available in the public interface.

1. *RequestSetTransform (const Transform &)* : Set the transform corresponding to the object-to-world transformation of the spatial object.
2. *RequestAddObject (SpatialObject *)* : Set a new object.

3. *RequestGetTransform()* : Request the transform associated to the object-to-world transformation of the spatial object.
4. *GetObject(unsigned long)* : Get a child object given the id.
5. *RequestAttachToTrackerTool(const TrackerTool *)* : Request the protocol for attaching to a tracker tool. This is a one-time operation. Once a Spatial Object is attached to a tracker tool it is not expected to be detached nor to be re-attached to a second tracker tool.

Derived classes may provide additional `Get` methods specific to their geometry.

10.4 Common Objects

IGSTK provides a rich set of spatial object types. Figure 10.2 depicts the range of built-in spatial object types in IGSTK.

In this section we detail the different spatial object types present in IGSTK.

10.4.1 Axes Object

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 to provide a visual reference of the orientation of space in the context of the scene.

The `igstk::AxesObject` is useful when creating complex objects to make sure that the orientation within the scene is consistent. It is also very informative for debugging purposes.

First, we include the appropriate header file:

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

We then declare the object using smart pointers, as follows:

```
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);
```

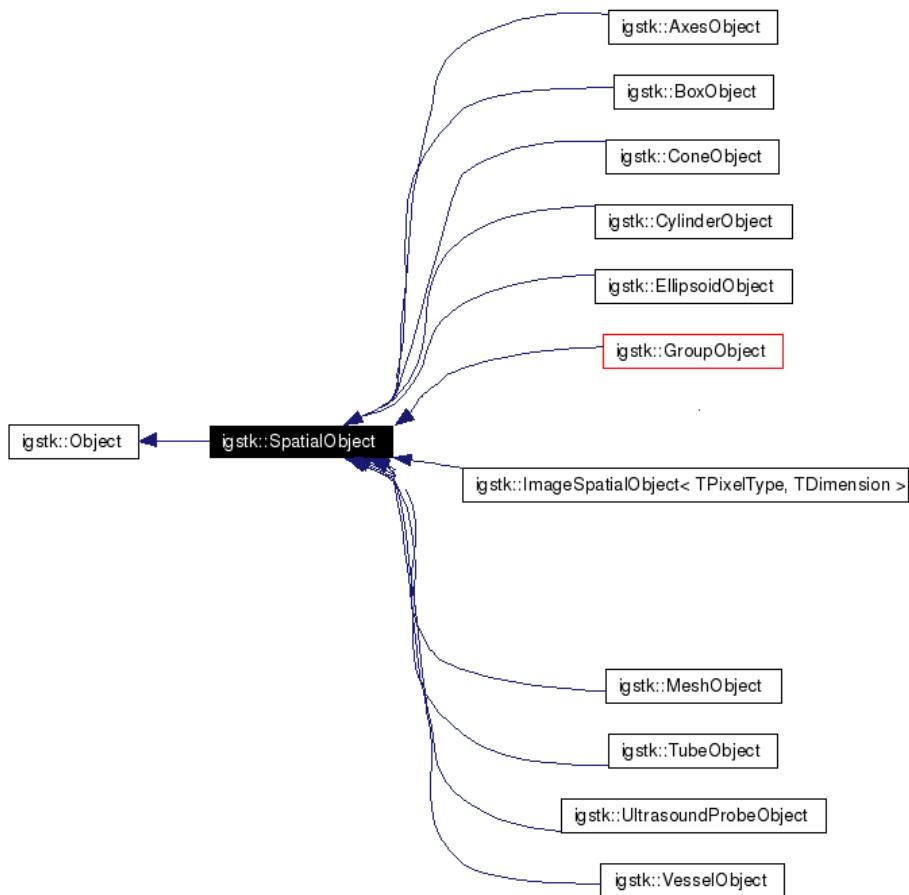


Figure 10.2: Spatial Object Types in IGSTK. Additional types may be constructed, though most other structures required may be composed using the techniques in Section 10.1.

In case we need to retrieve the length of the axes we can use the `GetSizeX()`,`GetSizeY()` and `GetSizeZ()` 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;
```

10.4.2 Box Object

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

The `igstk::BoxObject` represents a hexahedron in space. It is a useful building block to create a more complex object (see the 3D ultrasound probe in Chapter 11.6 for reference).

Let's 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 retrieve the size of the box, they can do so 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;
```

10.4.3 Cone Object

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, both expressed in mm. The radius represents the radius of the base of the cone. These two parameters can be set using `SetRadius()` and `SetHeight()`, as follows:

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

10.4.4 Cylinder Object

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, both expressed in mm. These two parameters can be set using `SetRadius()` and `SetHeight()`, as follows:

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

10.4.5 Ellipsoid Object

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 ellipsoid can be adjusted in each dimension using two different `SetRadius()` functions. The easiest way is to use the `SetRadius(double, double, double)` function, as follows:

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

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

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

10.4.6 Image Object

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 one class per modality for CT, MRI, and Ultrasound, as follows:

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

First, we declare an empty CT image using smart pointers:

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

Then, in some cases, it might be useful to know the intensity value of the image given a point in world coordinate (e.g. what is the intensity value at the tip of the needle). For a given point in physical space, we can ask if this 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 achieved using the `TransformPhysicalPointToIndex()` and `TransformPhysicalPointToContinuousIndex()` functions, as follows:

```

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. An image is considered empty if it has only pixels with zero intensity value.

```

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

```

10.4.7 Mesh Object

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 3D mesh structure. The mesh class provides an API to perform operations on points and cells. 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::MeshObjectType MeshObjectType;
MeshObjectType::Pointer mesh = MeshObjectType ::New();
```

A mesh is defined as a collection of 3D points (x,y,z) in space referenced by an identification number (ID). In order to add points to the mesh structure, we use the `AddPoint (unsigned int id, float x, float y, float z)` function. Let's add 4 points with consecutive IDs starting at zero, to 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);
```

The list of points can be retrieved using the `GetPoints()` function. The list of points returned consist of a list of `std::pair` with the first argument of the pair being the id of the point and the second argument storing the geometrical representation.

```
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 for adding cells to the mesh are defined as follows, where the vertices refer to the ID of the points previously defined.

```
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 one 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 `GetCells()`. This function returns a list of cells, as follows:

```
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++;
}
```

10.4.8 Tube Object

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 3D tubular structure in space. The tube is defined by a set of consecutive points representing its centerline. Each point has 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::TubeObjectType TubeObjectType;
TubeObjectType::Pointer tube = TubeObjectType ::New();
```

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

```
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. The command is as follows:

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

However, the second `GetPoints()` function is highly recommended. It is safer because it returns the internal list of points as a copy:

```
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;
```

10.4.9 Vascular Network and Vessel Objects

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

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

We first include the header files:

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

Next we declare `igstk::VascularNetworkObject`:

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

Then we create `igstk::VesselObject`:

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

Like the `igstk::TubeObject`, an `igstk::VesselObject` is defined as a collection of centerline points with an 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 vascular network. We use the `RequestAddVessel()` function. This function requires an `igstk::Transform` that relates the vessel to the vascular tree. In this example we set the identity transform with the longest possible expiration time.

```

igstk::Transform transform;
transform.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );

vasculature->RequestAddVessel( transform, vessel );

```

In some cases, we may want to get a vessel of interest from a VascularNetworkObject. To retrieve the vessel, we first need to create an observer.

```

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

```

Note that the declaration of the observer should be done outside of the class. As we have seen in the previous section this macro will create two functions depending on the name of the first argument:

1. GotVessel() which returns true if the vessel exists.
2. 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(
    igstk::VesselObjectModifiedEvent(),
    vesselObserver);

```

We then request a vessel given its position in the list using the RequestGetVessel(unsigned long position) function. We also check to see if the observer got the vessel, i.e if the vessel exists.

```

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

```

The vessel is retrieved using the GetVessel() function from the observer, as follows:

```

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

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

```

10.5 Reading Spatial Objects

IGSTK has the ability to read objects from files. In this section we demonstrate how to read a vascular network from a file. Reading other objects can be done in a similar manner.

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

`Examples/SpatialObjects/ReadVascularNetworkObject.cxx`.

This example describes how to use the `igstk::VascularNetworkReader` to read a `SpatialObject` vascular tree from a file. 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`. As we have seen previously, this macro expects three arguments: 1) the name of the observer (to be determined by the user), 2) the type of event to observe, and 3) the type of the object to be returned. The command is as follows:

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

First, we declare the `igstk::VascularNetwork` using standard type definitions and smart pointers.

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

We then plug a logger into the reader to get information about the reading process (see Chapter 13 for more information).

```
typedef igstk::Object::LoggerType 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 name of the file 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 plug the observer into the reader.

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

Then, we request the output vascular network.

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

If everything goes well, the observer should receive the vascular network as a pointer. We can confirm this by using the `GotVascularNetwork()` function of the observer.

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

Finally, we get the vascular network using the `GetVascularNetwork()` function.

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

Then, we display the object information to make sure everything is correct.

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

10.6 Conclusion

We have shown in this chapter, that the `igstk::SpatialObject` defines a base class for object geometry. By deriving from this base `igstk::SpatialObject` class, users can extend the current set of objects present in the toolkit. Moreover, complex objects can then be created, for instance the `igstk::UltrasoundProbeObject` can be found in Chapter 25.

In the next chapter, we introduce the `igstk::SpatialObjectRepresentation` class to display spatial objects.

Spatial Object Representation

11.1 Introduction

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

A spatial object representation 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` instance per `SpatialObject`. This allows one to tune the rendering parameters, such as the color of the object, while keeping the same common geometry.

This chapter describes spatial object representation structures, and provides examples to show how to connect spatial objects and views using spatial object representations.

11.2 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 the Fast Light GUI Toolkit (FLTK) to create a window, therefore we include the appropriate header files.

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

Like any applications in IGSTK we first initialize the `RealTimeClock` to ensure proper synchro-

nization between the different components.

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

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

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

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

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

Every `ObjectRepresentation` has a 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 set the `BoxSpatialObject` geometry to the `ObjectRepresentation`. Internally the `ObjectRepresentation` creates VTK actors from the actual geometry of the `igstk::SpatialObject`.

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

We then define the GUI window and the view.

```
F1_Window * form = new F1_Window(512,512,"Displaying my first object");

typedef igstk::View3D View3DType;

// Create an FLTK minimal GUI
typedef igstk::FLTKWidget      FLTKWidgetType;

View3DType::Pointer view3D = View3DType::New();

// instantiate FLTK widget
FLTKWidgetType * fltkWidget3D =
    new FLTKWidgetType( 6,6,500,500,"3D View");
fltkWidget3D->RequestSetView( view3D );

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. The refresh rate defines when the view should be rendered. Note that a faster refresh rate might be requested but might not be achieved, depending on the capabilities of the system.

```
view3D->SetRefreshRate( 0.1 );
```

A spatial relationship must be established between the View and the Object to be visualized. In this case we define it as an identity transform, which means that both of them refer to the same coordinate system.

```
igstk::Transform transform;
transform.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );

cube->RequestSetTransformAndParent( transform, view3D );
```

Finally we request the view to start rendering the scene.

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

We then refresh the display until the window is closed. Note that we use the internal timer from FLTK to make sure the view has time to be rendered. The PulseGenerator::CheckTimeouts() function is the heart of the system which tells the views to refresh itself.

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

At the end, we delete the fltkWidget3D and form since they are not using smart pointers.

```
delete fltkWidget3D;
delete form;
```

The output of this example is shown in Figure 11.1.

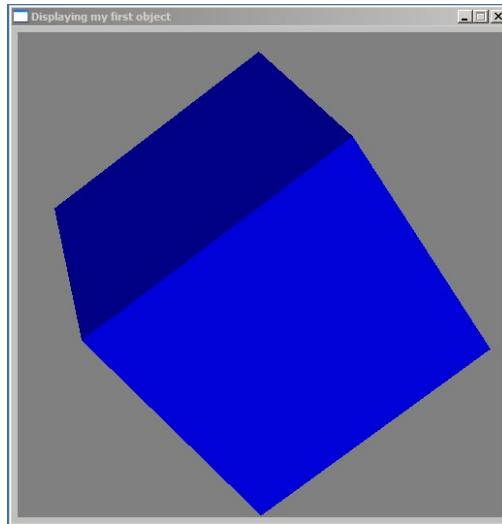


Figure 11.1: Object Representation Example.

11.3 State Machine

Like all IGSTK components, spatial object representations are governed by a state machine. The state diagram is shown in Figure 11.2.

The state machine has the following states:

1. *NullSpatialObjectState*: Initial state indicating not yet tied to a spatial object.
2. *ValidSpatialObjectAndInvisibleState*: State after being set to a spatial object.
3. *ValidSpatialObjectAndVisibleState*: After being set, the representation may be made visible.
4. *AttemptingUpdatePositionAndInvisibleState*: The state while attempting to update the object's position.
5. *AttemptingUpdatePositionAndVisibleState*: Same as previous except if the object was previously visible.
6. *ValidTimeStampState*: Used internally to check whether the time stamp of rendering is within the window of the timestamp of the transform. If it is, the representation is in this state.
7. *InvalidTimeStampState*: If it is not, the representation is set in this state, presumably leading to an invisible state on the next rendering.

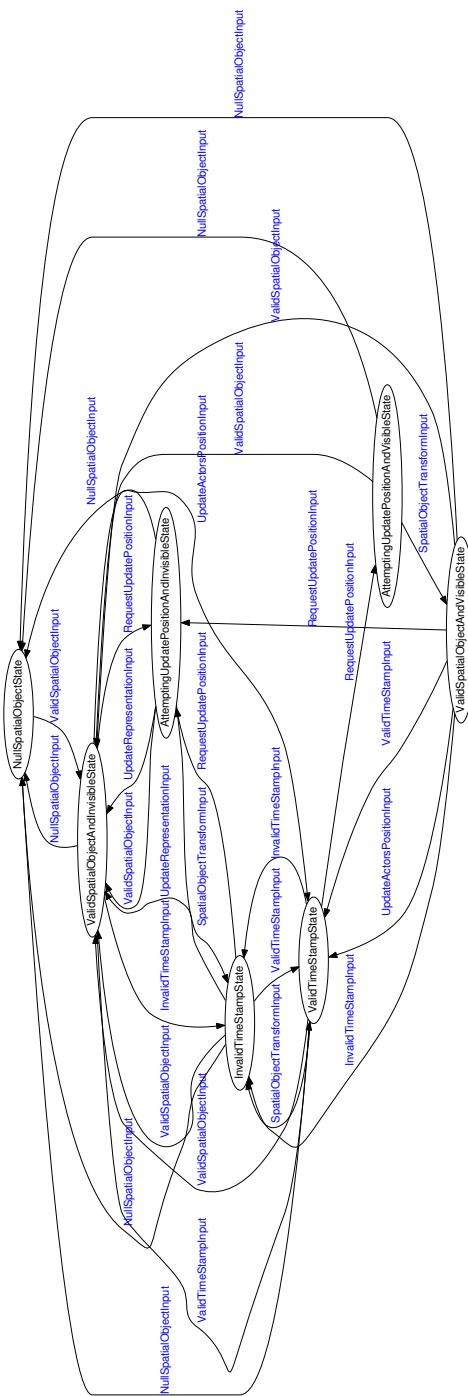


Figure 11.2: Spatial Object Representation State Machine.

11.4 Component Interface

The following methods are available in the public interface.

1. *RequestUpdatePosition (const TimeStamp &)* : Update the visual representation position.
2. *RequestUpdateRepresentation (const TimeStamp &)* : Update the visual representation with changes in the geometry.

Other Get methods are available for getting attribute information such as red, green, and blue color components. Other methods may be available in derived classes based on their available attributes.

11.5 Common Object Representations

Just as `SpatialObject` is a base class for various spatial object derived types, `ObjectRepresentation` serves as a base class and a number of derived classes for different representation types, as shown in Figure 11.3.

In this section we present the different spatial object representations available in the toolkit and we describe their internal workings.

11.5.1 Axes Object

The `igstk::AxesObjectRepresentation` uses a `vtkAxesActor` class 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 as shown in Figure 11.4.

11.5.2 Box Object

The `igstk::BoxObjectRepresentation` uses a `vtkCubeSource` object internally to display a 3D hexahedron as shown in Figure 11.4. The color and opacity of the box can be set.

11.5.3 Cone Object

The `igstk::ConeObjectRepresentation` uses a `vtkConeSource` object internally to display a 3D cone in space as shown in Figure 11.5. The color and opacity of the cone can be set.

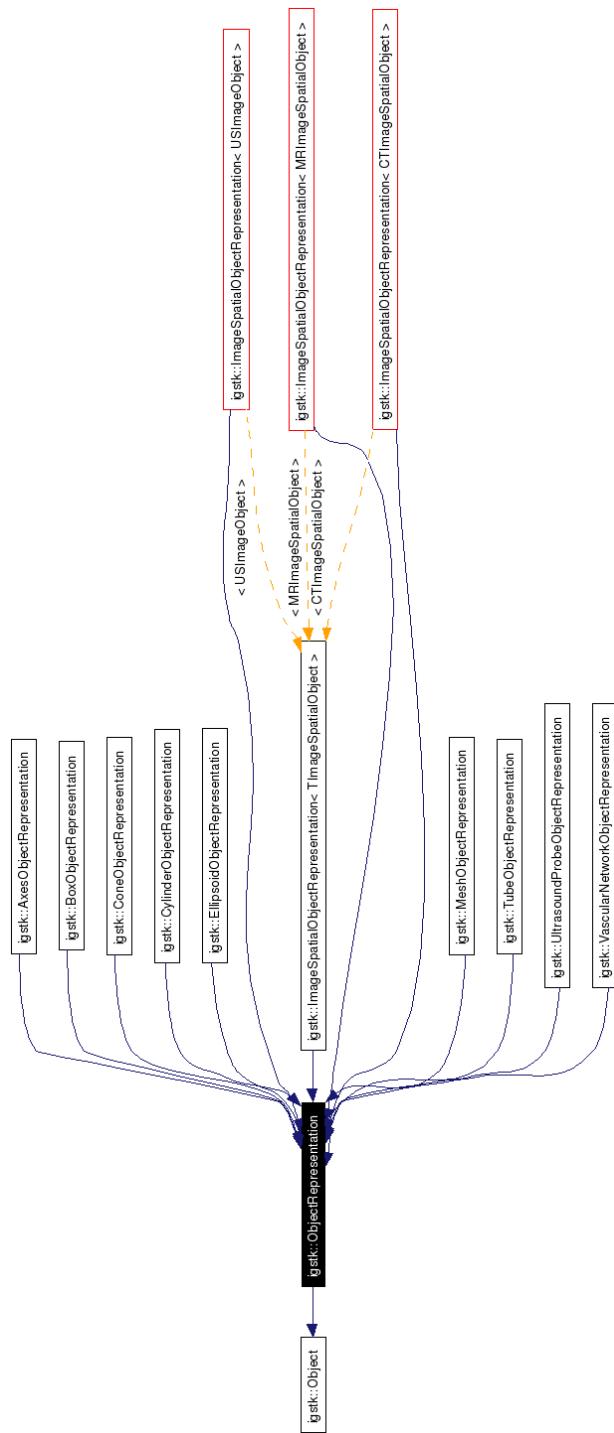


Figure 11.3: Spatial Object Representation Types in IGSTK.

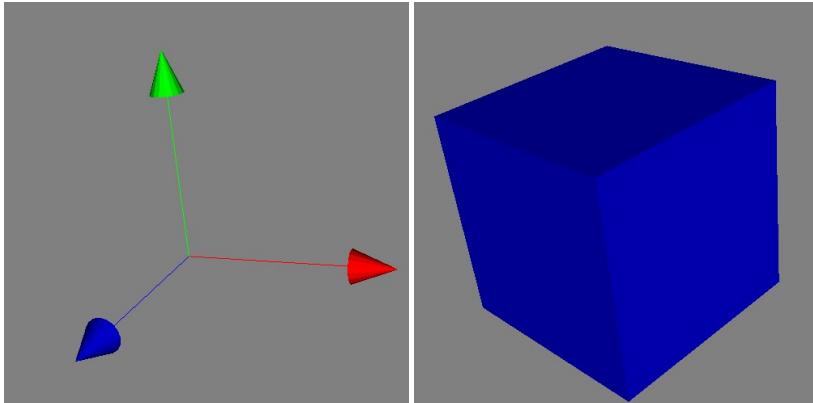


Figure 11.4: Axes and Box Object Representation.

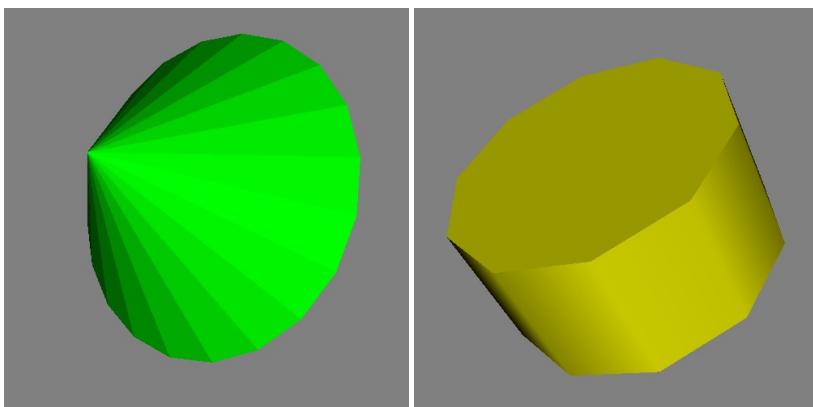


Figure 11.5: Cone and Cylinder Object Representation.

11.5.4 Cylinder Object

The `igstk::CylinderObjectRepresentation` uses a `vtkCylinderSource` object internally to display a 3D cylinder as shown in Figure 11.5.

11.5.5 Ellipsoid Object

The `igstk::EllipsoidObjectRepresentation` uses a `vtkSuperquadricSource` object internally to display a 3D ellipsoid as shown in Figure 11.6.

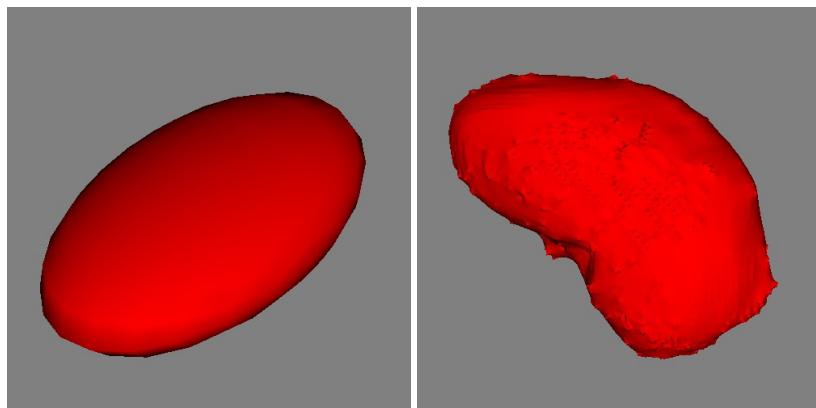


Figure 11.6: Ellipsoid and Mesh Object Representation.

11.5.6 Mesh Object

The `igstk::MeshObjectRepresentation` uses a `vtkUnstructuredGrid` object internally to display a 3D polydata in space as shown in Figure 11.6. The appropriate cells are created using VTK based on their geometry type.

11.5.7 Vascular Network Object

The `igstk::VascularNetworkObjectRepresentation` uses a complete VTK pipeline to display a 3-dimensional tube in space. First, a `vtkPolyLine` is used to describe the centerline of the tube, then it is converted to a `vtkCellArray` and plugged into a `vtkPolyData`. From there a cleaning stage is performed using the `vtkCleanPolyData` to make sure the tube is free from duplicate points. Finally we create the tube object using a `vtkTubeFilter`.

For a more realistic representation, we also add spheres using `vtkSphereSource` at both extremities of the tube.

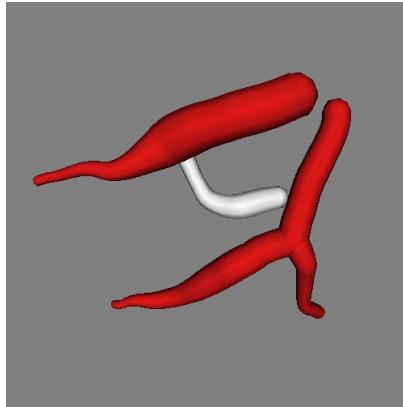


Figure 11.7: Vascular Network Object Representation.

11.6 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 `igstk::UltrasoundProbeObjectRepresentation` class provides a geometrical representation of an ultrasound probe as shown in Figure 11.8.

One can notice that the `UltrasoundProbeObject` class is fairly simple and only exposes the parameters the user is able to change. However the associated representation class involves a complex VTK pipeline. Among the VTK classes within the pipeline, the `vtkCylinder`, `vtkPlane`, `vtkImplicitBoolean` and the `vtkMarchingContourFilter` are the primary classes used. For more information, we recommend taking a look at the internal class implementation.

11.7 Sharing and Duplicating Object Representations

In some configuration where memory is limited, it might be useful to share representation and avoid creating duplicate rendering objects. In this section we show how to share object representations between views.

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

From the previous example, we add a second View to our window. We have `View3D1` and `View3D2` as two `View3Ds`.

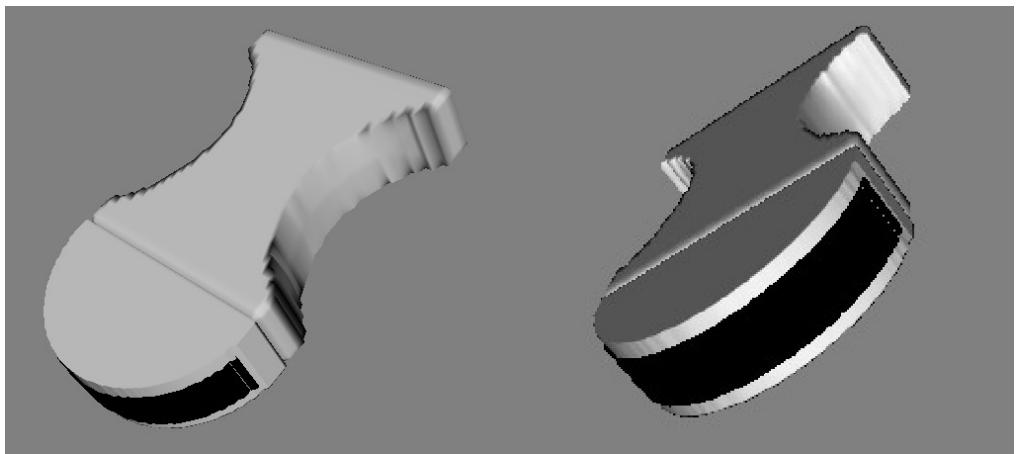


Figure 11.8: Ultrasound Probe Object Representation.

```
F1_Window * form = new F1_Window(600,300,"Sharing Object Representations");

typedef igstk::View3D           View3DType;
typedef igstk::FLTKWidget       FLTKWidgetType;

View3DType::Pointer view3D1 = View3DType::New();
View3DType::Pointer view3D2 = View3DType::New();

// instantiate FLTK widget
FLTKWidgetType * fltkWidget3D1 =
    new FLTKWidgetType( 10,10,280,280,"View 3D 1");
fltkWidget3D1->RequestSetView( view3D1 );

FLTKWidgetType * fltkWidget3D2 =
    new FLTKWidgetType( 310,10,280,280,"View 3D 2");
fltkWidget3D2->RequestSetView( view3D2 );

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 `igstk::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 );
```

A spatial relationship must be established between the Views and the Object to be visualized. In this case we define it as an identity transform, which means that both of them refer to the same coordinate system.

```

igstk::Transform transform;
transform.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );

view3D1->RequestSetTransformAndParent( transform, cube );
view3D2->RequestSetTransformAndParent( transform, cube );

```

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

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

Instead of being shared, the `ObjectRepresentation` can also be copied using the `Copy()` function which creates a deep copy of the current representation as shown below.

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

11.8 Conclusion

In this chapter, we have seen how spatial object representations can be used to render a spatial object on a display. IGSTK provides representation mappings for the various spatial object types defined in Chapter 10. These representations are treated as first-class components in IGSTK, following the same API and state machine design pattern, and provide an elegant decoupling of logical structure definition (spatial objects) to rendered projections of the structure. In the next chapter we complete the rendering infrastructure by describing how IGSTK manages displays using view components.

View

In an image-guided surgery application, clinicians depend on an accurate and informative graphical presentation of the surgical scene to better perform their task. IGSTK provides classes for this purpose in the view component. In addition, IGSTK provides widget classes that serve as a bridge between commonly used open source GUI libraries and the view classes.

12.1 View Component Design

View classes encapsulate VTK classes into a restrictive API subjected to control of a state machine. Viewers aggregate graphical descriptions of spatial objects (spatial object representations) into a single coherent scene. Synchronization between the scene generation and the rendering of the registered representations is achieved using `igstk::PulseGenerator`. `igstk::View2D` and `igstk::View3D` subclasses are implemented for 2D and 3D viewing capabilities.

12.1.1 Synchronization Between Scene Generation and Rendering

The view component uses a pulse generator to refresh the rendering at a predefined frame rate. The pulse generator generates clock tick events at regular interval. At each clock tick event, all of the `igstk::SpatialObjectRepresentation` classes that are registered with this View class get updated. For a detailed description of the synchronization, see Section 5.5.

12.1.2 GUI Interaction

Application developers build GUI-based image-guided applications using open source GUI libraries. To integrate IGSTK into such applications, IGSTK provides widget classes for Qt (`igstk::QWidget`) and FLTK (`igstk::FLTKWidget`) GUI libraries. These widgets display `igstk` view render window in the GUI library specific window class and translate GUI-specific interaction events into VTK interactions events that would be handled by the interactor.

12.2 State Machine

Figure 12.1 shows the State Machine of the `igstk::View` class.

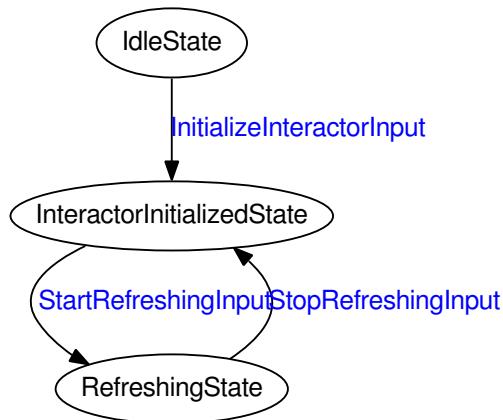


Figure 12.1: State Machine of the View class.

This class has the following states:

- *Idle* : Idle state
- *InteractorInitialized*: Interactor initialized state
- *Refreshing* : Refreshing state

12.3 Component Interface

The following methods are available in the public interface:

- `SetRefreshRate(double)` : Set the desired frequency for refreshing the view.
- `RequestAddObject(ObjectRepresentation *)` : Add an object representation to the list.
- `RequestAddAnnotation2D(Annotation2D *)` : Add corner annotation.
- `RequestRemoveObject(ObjectRepresentation *)` : Remove an object representation from the list.
- `RequestSaveScreenShot(std::string)` : Save a screen shot into a file in PNG format.
- `RequestDisableInteractions()` : Disable user interactions with the render window.

- *RequestEnableInteractions()* : Enable user interactions with the render window.
- *RequestResetCamera()* : Reset the camera to a known position and orientation.
- *RequestStart()* : Start the periodic refreshing of the view.
- *RequestStop()* : Stop the periodic refreshing of the view.
- *SetCameraPosition()* : Set camera position.
- *SetCameralFocalPoint()* : Set camera focal point.
- *SetCameraViewUp()* : Set camera view up vector.
- *SetCameraClippingRange()* : Set camera near and far clipping ranges.
- *SetCameraParallelProjection()* : Turn on or off parallel projection.
- *SetRendererBackgroundColor()* : Set background color.
- *SetCameraZoomFactor()* : Set camera scale factor.

12.4 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 render spatial object representations in FLTK window.

First, igstk 3D view, FLTK widget and other useful data types are defined:

```
typedef igstk::FLTKWidget           WindowWidgetType;
typedef igstk::View3D                View3DType;
typedef igstk::Object::LoggerType   LoggerType;
typedef itk::StdStreamLogOutput    LogOutputType;
```

For debugging purposes, the 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 this out, 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 the `igstk::EllipsoidObjectRepresentation` class. The representation class provides the mechanism to generate a graphical description of the spatial object for visualization in a VTK scene, as follows:

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

Next, the FLTK main window and IGSTK FLTK Widget objects are instantiated, and the FLTK form is completed by calling its `end()` method.

```
F1_Window * form = new F1_Window(301,301,"View Test");
WidgetType * widget = new WindowWidgetType(10,10,280,280,"3D View");
form->end();
```

The `View` object is instantiated and then is connected to the FLTK Widget by using the `RequestSetView()` method, and at that point the FLTK form can be displayed by calling its `show()` method.

```
View3DType::Pointer view3D = View3DType::New();
widget->RequestSetView( view3D );
form->show();
```

A geometrical transformation can then be applied to define the spatial relationship between the view and the ellipsoid object as follows:

```
const double validityTimeInMilliseconds =
    igstk::TimeStamp::GetLongestPossibleTime();

igstk::Transform transform;
igstk::Transform::VectorType translation;
translation[0] = 0;
translation[1] = 0;
translation[2] = 0;
igstk::Transform::VesorType 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->RequestSetTransformAndParent( transform, view3D );
```

The ellipsoid is added to the scene using the `RequestAddObject` method and then the scope of the camera is reset in order to make sure that the object is visible in the window.

```
view3D->RequestAddObject( ellipsoidRepresentation );  
  
view3D->SetCameraPosition( 0, 0, -1 );  
view3D->SetCameraViewUp( 0, -1, 0 );  
view3D->SetCameraFocalPoint( 0, 0, 0 );  
  
view3D->RequestResetCamera();
```

The View components are designed for refreshing their representation at regular intervals. The application developer must set the desired refresh rate in Hertz, which should trigger the internal generation of pulses that makes it possible for the View class to refresh itself as follows:

```
view3D->SetRefreshRate( 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 pulse generator timers in all the autonomous IGSTK classes are checked to see if they should trigger timer events. The same 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<100; i++)  
{  
    F1::wait( 0.01 );  
    igstk::PulseGenerator::CheckTimeouts();  
    F1::check();      // trigger FLTK redraws  
}
```

Once the event loop finishes, the method `RequestStop()` should be called to stop the refresh process of the View class as follows:

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

12.5 Conclusion

The view component is used to 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. Furthermore, IGSTK provides widget classes to provide support for GUI-based application development.

Part IV

Services

Logging

For critical software systems, a logging service enables post-analysis of the operational processes and potential recovery from failure. ITK¹ and IGSTK logging services help record messages to output streams so that analysis, verification, and debugging of systems and operational processes are done efficiently. Loggers can also be used to record any important data.

13.1 General Background

In image-guided surgery systems, logs will be used for enhancing the robustness of the system during development, analyzing the causes and points of failure, and verifying that the system works in an expected way. In IGSTK, it is important to guarantee that all operations will be logged before points of failure.

Therefore messages that are sent to the logger are continuously flushed to non-volatile storage to ensure that they will be available for review in case the system enters a non-recoverable condition.

13.2 Structure of the Logging Service

A priority level is assigned to each log message. A logger only prints messages from the level at or above the logger's designated level. With this feature, developers can filter out uninteresting messages. For example, messages for debugging can be printed only during development, and turned off when the application is deployed.

Each logger can print messages to multiple destinations (files, consoles, GUI windows, etc.). Messages from ITK and VTK message windows can be redirected to loggers so that every message generated using Logging components of ITK and IGSTK is included in the unified logging service.

¹Logging service was originally developed in IGSTK and later adopted by ITK.

Loggers can be run in a separate thread, printing messages sequentially without interleaving multiple messages. Multiple loggers can be created and used through a `LoggerManager`.

This section presents message prioritization strategies, important classes in the logger's output hierarchy, and thread-enabled classes.

13.2.1 Logger Priorities

Each logger object and message has a designated priority level. A logger object processes all messages at its own level or higher and ignores all messages at lower levels. The priority levels are as follows, displayed in descending order:

MUSTFLUSH Messages with this level must be flushed immediately.

FATAL Messages at this level contain information about fatal errors or exceptions. When a message of this type is sent, the application must abort.

CRITICAL Messages at this level contain information about a critical error. This level is displayed as **ERROR**. When a message of this type is sent, the current operation must abort.

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

INFO Messages with this level contain information without potential danger.

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

NOTSET Messages at this level are not filtered. This level is not recommended.

Again, a 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.

13.2.2 Logger Output Properties

This section describes various properties that control how and when logging output is created.

Flushing

Typically, data posted to output media is buffered to reduce the number of output operations. Output buffering is the technique of storing data in a reserved memory area in anticipation of writing out to disk, over a network interface, or out through some other communication channel. Buffering is especially effective when the relative cost of each output operation is expensive. For example, disk I/O is typically buffered until a buffer the size of a disk block is full. Flushing a buffer is the process of writing its contents to the output device. Typically, flushing is done when the buffer is full or the application requests it (such as when the application wants to

terminate normally). These situations are automatically handled by the operating system or system software.

An application that abnormally terminates cannot guarantee that its output buffers have been properly flushed. Likewise, a logger that does not request frequent flush operations will fall slightly out of sync with actual events recorded for the application. For image-guided surgery, we need to provide better guarantees that log records are written at specific important points in time. Urgent messages should be flushed, even sacrificing the efficiency of the output operation. IGSTK logger objects provide the following options to handle this requirement:

- Use the Flush() method manually.
- Set a minimal LevelForFlushing and let the logger automatically flush whenever messages with an equal or more serious level than the specified LevelForFlushing are met.
- Print a message with the MUSTFLUSH level so that the buffer is flushed.

Formatting

Loggers derived from LoggerBase can override the LoggerBase::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 follows:

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

For example, a logger prints messages in the default format as follows:

```
...
24455465909.031303  : MainLogger  (DEBUG)  in main()
24455465909.031994  : MainLogger  (DEBUG)  Hello world!
24455465909.032463  : MainLogger  (ERROR)  A file open failed.
24455465909.033783  : MainLogger  (DEBUG)  Hello world!
24455465909.034256  : MainLogger  (INFO)    Bye!
...
...
```

In the above example, the time stamp is being displayed in millisecond. We can also display it in a friendlier format, which is more readable. For instance, you can set the logger to log the time stamp in a human readable format by using these command:

```
logger->SetTimeStampFormat( itk::LoggerBase::HUMANREADABLE );
logger->SetHumanReadableFormat("%Y %b %d, %H:%M:%S");
```

The resulting log message will look like:

```
...
```

```
2009 Jan 27, 15:47:44 : MainLogger (DEBUG) in main()
2009 Jan 27, 15:59:09 : MainLogger (DEBUG) Hello world!
2009 Jan 27, 15:59:09 : MainLogger (DEBUG) A file open failed.
2009 Jan 27, 15:59:09 : MainLogger (DEBUG) Hello world!
2009 Jan 27, 15:59:17 : MainLogger (DEBUG) Bye!
...

```

Timestamp

A timestamp for log messages is provided by the `RealtimeClock` object. Each message contains its timestamp in seconds. The precision of the `TimeStamp` varies depending on the hardware platform and the operating system.

13.2.3 Logger Base Classes

The logger structure is encapsulated in a small number of classes and extensions from ITK. The class organization is shown in the Figure 13.1.

This section describes ancestor classes, while the next section describes specific functionalities provided by various derived classes.

LoggerBase

`LoggerBase` class is the base implementation of all 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.

Logger

The logger class is derived from `LoggerBase` as a placeholder, not by adding any functionality. `Logger` has the simplest form as a derived class of `LoggerBase`.

LoggerManager

A `LoggerManager` object manages multiple logger objects. A name is given to each logger so that it can be accessed from the `LoggerManager` by that name. Using the `LoggerManager` interface, a log message can be distributed to every logger managed by the `LoggerManager` via one method call, instead of having to call each logger individually.

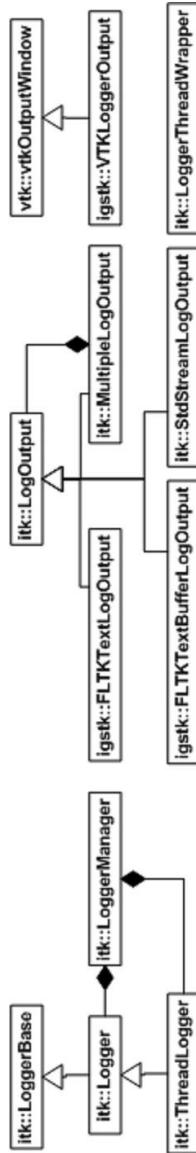


Figure 13.1: Class Hierarchy for ITK/IGSTK Logging Classes.

13.2.4 LogOutput

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

The `itk::LogOutput` class represents the destination of the logging and serves as a base class for other `LogOutput` classes. Classes derived from the `LogOutput` class allow the log data to be directed to specific locations (e.g. disk files) for recording. For example, the derived class `StdStreamLogOutput` will send the log data to a specific stream or file, and the derived class `MultipleLogOutput` will allow the log data to be sent to multiple locations. Several `LogOutput` implementations are available. These are described next.

First, we begin by including header files for using `LogOutput` objects. The `itk::StdStreamLogOutput` needs the `itkStdStreamLogOutput.h` header file. `itk::MultipleLogOutput`, `igstk::FLTKTextBufferLogOutput`, and `igstk::FLTKTextLogOutput` need proper header files to be included, as follows:

```
#include "itkStdStreamLogOutput.h"
#include "itkMultipleLogOutput.h"
#include "igstkFLTKTextBufferLogOutput.h"
#include "igstkFLTKTextLogOutput.h"
```

StdStreamLogOutput

The `itk::StdStreamLogOutput` encapsulates and allows the log data to be sent to the standard output stream, which can be a console output stream, an error output stream, or a file output stream.

Each `LogOutput` object must be created by the `New()` method and then set to have a reference to a stream object. The example code below shows how to create `itk::StdStreamLogOutput` objects and set streams for the console and a file output stream. The prepared `LogOutput` objects are added to a logger, see Chapter 13.2. The commands are as follows:

```
itk::StdStreamLogOutput::Pointer
    consoleLogOutput = itk::StdStreamLogOutput::New();
consoleLogOutput->SetStream( std::cout );

itk::StdStreamLogOutput::Pointer fileLogOutput =
    itk::StdStreamLogOutput::New();
ofstream fout("log.txt");
fileLogOutput->SetStream( fout );

// Creating a logger object
typedef igstk::Object::LoggerType           IGSTKLoggerType;
IGSTKLoggerType::Pointer logger = IGSTKLoggerType::New();
logger->SetName( "MainLogger" );
```

```
// Attaching LogOutput objects to a logger
logger->AddLogOutput( consoleLogOutput );
logger->AddLogOutput( fileLogOutput );

// Setting a priority level for a logger
logger->SetPriorityLevel( IGSTKLoggerType::DEBUG );

// Testing to print a debug message
logger->Debug("Debug message\n");
logger->Critical("Critical message\n");
logger->Fatal("Fatal message\n");

// Creating a LoggerManager object
itk::LoggerManager::Pointer manager = itk::LoggerManager::New();

// Creating a logger object through the LoggerManager object
typedef itk::Logger ITKLoggerType;

ITKLoggerType::Pointer logger2 =
    manager->CreateLogger( "org itk logTester logger",
                           itk::LoggerBase::DEBUG, itk::LoggerBase::CRITICAL );

// Adding a LogOutput object to the LoggerManager object
manager->AddLogOutput(consoleLogOutput);

// Testing to print a debug message
manager->Write(itk::LoggerBase::DEBUG, "This is a DEBUG message.\n");
```

FLTKTextBufferLogOutput

The `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.

Let's create an `igstk::FLTKTextBufferLogOutput` object,

```
igstk::FLTKTextBufferLogOutput::Pointer
    fltkBufferOutput = igstk::FLTKTextBufferLogOutput::New();
```

Then, the following codes create an FLTK window and an `Fl_Text_Display` widget, which is contained in the window:

```
Fl_Window *win = new Fl_Window(0,0,400,300,
                               "igstkFLTKTextBufferLogOutputExample");
Fl_Text_Display *textDisplay = new Fl_Text_Display(0,0,400,300,NULL);
```

An Fl_Text_Buffer object is created, then, the Fl_Text_Display widget is set to have a pointer to the text buffer using the buffer() method:

```
Fl_Text_Buffer *textBuffer = new Fl_Text_Buffer();
textDisplay->buffer(textBuffer);
```

The end() method denotes the end of the GUI code. The show() method makes the window visible:

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

Finally, we assign the FLTK text buffer object as a stream for the FLTKTextBufferLogOutput object and write some messages. The Write() method only writes a message on a text buffer object. It does not update the display. The Flush() method actually updates the display to show the contents of the buffer. Although the following example uses Write() and Flush() methods from the LogOutput object, it is recommended that a logger object be used instead.

```
fltkBufferOutput->SetStream(*textBuffer);
fltkBufferOutput->Write("This is a test message.\n");
fltkBufferOutput->Flush();
```

FLTKTextLogOutput

The igstk::FLTKTextLogOutput displays messages in a FLTK text window, which is in Fl_Text_Display type. This is meant to display log messages on a GUI window. The FLTKTextLogOutput does not need to use an explicit Flush() method because updating the display is automatically done by the FLTK text display object.

An igstk::FLTKTextLogOutput object is created; then, an Fl_Text_Display object is set as a stream of the FLTKTextLogOutput object. The Write() method prints some messages on the text display widget.

```
igstk::FLTKTextLogOutput::Pointer fltkTextOutput =
    igstk::FLTKTextLogOutput::New();
fltkTextOutput->SetStream(*textDisplay);
fltkTextOutput->Write("This is a test message.\n");
```

MultipleLogOutput

The itk::MultipleLogOutput method aggregates multiple LogOutput objects. It is used in the LoggerBase class so that logger classes can contain multiple LogOutput objects.

The following example shows how to create the itk::MultipleLogOutput object and add multiple LogOutput objects to the MultipleLogOutput object. Whenever an application sends

messages to the `MultipleLogOutput` object, all the messages are redirected to every `LogOutput` object in the `MultipleLogOutput` object. The commands are as follows:

```
itk::MultipleLogOutput::Pointer multipleLogOutput =
    itk::MultipleLogOutput::New();
multipleLogOutput->AddLogOutput(consoleLogOutput);
multipleLogOutput->AddLogOutput(fileLogOutput);
multipleLogOutput->Write("This is a test message.\n");
```

Extending LogOutput

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

13.2.5 Redirecting ITK and VTK Log Messages to Logger

Overriding `itk::OutputWindow`

The `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 any type of logger by using the `itk::LoggerOutput` class. `itkLoggerOutput.h` is the header file for using the `itk::LoggerOutput` class.

The following code fragment shows how to create an `itk::LoggerOutput` object and override `itk::OutputWindow` to redirect messages from ITK to a logger object. Once the setup using `OverrideITKWindow()` and `SetLogger()` methods is done, every message sent toward ITK is redirected to a logger object. The commands are as follows:

```
// Create an ITK LoggerOutput and then test it.
itk::LoggerOutput::Pointer itkOutput = itk::LoggerOutput::New();
itkOutput->OverrideITKWindow();
itkOutput->SetLogger(logger);

// Test messages for ITK OutputWindow
itk::OutputWindow::Pointer outputWindow = itk::OutputWindow::GetInstance();

outputWindow->DisplayText("This is from ITK\n");
outputWindow->DisplayDebugText("This is from ITK\n");
outputWindow->DisplayWarningText("This is from ITK\n");
outputWindow->DisplayErrorText("This is from ITK\n");
outputWindow->DisplayGenericOutputText("This is from ITK\n");
```

Overriding vtkOutputWindow

The `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. `igstkVTKLoggerOutput.h` should be included for using the `igstk::VTKLoggerOutput` class.

The structure of the example code is the same as the example for the `itk::LoggerOutput` class. An application creates an `igstk::VTKLoggerOutput` object and calls the `OverrideVTKWindow()` and `SetLogger()` methods. After that, all the messages toward `vtkOutputWindow` are redirected to a logger object. The commands are as follows:

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

// Test messages for VTK OutputWindow
vtkOutputWindow::GetInstance()->DisplayText(
    "This is from vtkOutputWindow\n");
vtkOutputWindow::GetInstance()->DisplayDebugText(
    "This is from vtkOutputWindow\n");
vtkOutputWindow::GetInstance()->DisplayWarningText(
    "This is from vtkOutputWindow\n");
vtkOutputWindow::GetInstance()->DisplayErrorText(
    "This is from vtkOutputWindow\n");
vtkOutputWindow::GetInstance()->DisplayGenericWarningText(
    "This is from vtkOutputWindow\n");
```

13.2.6 Multi-Threaded Logging

LoggerThreadWrapper

`itk::LoggerThreadWrapper` is a template class that wraps a logger class to enable logging in a separate thread. `LoggerThreadWrapper` inherits the logger class given as a template argument. Logging method calls made through a `LoggerThreadWrapper` object are queued and performed whenever the thread takes the available computational resource. `LoggerThreadWrapper` provides more flexibility than `ThreadLogger` by allowing various types of Loggers to be used; however, `LoggerThreadWrapper` may be problematic when used with compilers with weak C++ template support.

To use this class, an application should include the header file `itkLoggerThreadWrapper.h`. The example code fragment shows how to wrap a logger within the `itk::LoggerThreadWrapper` class:

```
typedef itk::LoggerThreadWrapper<itk::Logger> LoggerType;
LoggerType::Pointer wrappedLogger = LoggerType::New();
```

The wrapped logger is the same as other loggers except that it runs in a separate thread from the

thread that uses the logger object.

ThreadLogger

The `itk::ThreadLogger` provides the same functionality as `LoggerThreadWrapper` except that `ThreadLogger` is derived from the logger class. If different types of loggers are necessary instead of the simple logger class, these classes derive from the `ThreadLogger` class by creating a new class or by using `LoggerThreadWrapper`.

An application should include the header file called `itkThreadLogger.h` to use this class. The example code fragment shows how to create an `itk::ThreadLogger` object.

```
itk::ThreadLogger::Pointer threadLogger = itk::ThreadLogger::New();
```

13.3 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.

The `XMLLogger` class constructs log messages in XML format. Indentation is done automatically. Most XML viewers show the hierarchical structure of log messages and provide user interfaces to collapse and expand sub-elements. `XMLLogger` opens a new element when the first character of the log message is “<” and closes an element when the first character is “>”. Otherwise, a self-closing element is created when no angular bracket is used for the first character. The `BuildFormattedEntry()` method is redefined in the `XMLLogger` class for overriding default formatting. It creates a string containing a timestamp, the logger name, the priority level of a message, and the message content. Some of these components can be omitted if unnecessary. The logger name was omitted here to shorten the length of messages. The commands are as follows:

```
namespace igstk
{
    class XMLLogger: public igstk::Logger
    {
        public:
            typedef XMLLogger                               Self;
            typedef itk::SmartPointer<Self>               Pointer;
            typedef igstk::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 and 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( LoggerType::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, as follows:

```
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 follows:

```
<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>
```

13.4 Conclusion

IGSTK applications can use loggers to reveal the internal workings of an application for post-analysis, verification and debugging, writing important data for recording results, or perhaps in the future for storing the states of the hardware and software components for automatic recovery.

The logging component was designed to be extensible and customizable so that a variety of needs can be met. Therefore, developers can customize it for their specific purposes.

Image I/O

Image input/output (I/O) capabilities are essential services in an image-guided surgery application platform. Applications built using IGSTK typically have a need to read and write images, such as preoperative and intraoperative scans, for surgical planning and guidance. This chapter describes such image I/O functionalities provided by IGSTK.

14.1 DICOM Reader

At the beginning, a design decision was made that image formats not suitable for medical application would not be used in IGSTK. Hence, IGSTK only supports the DICOM image standard.

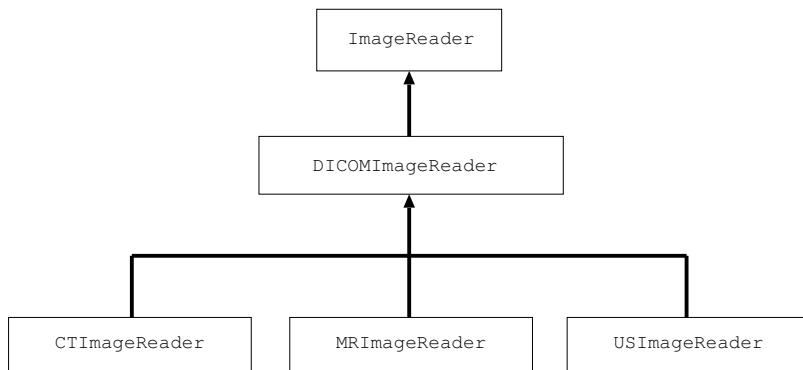


Figure 14.1: ImageReader Class Hierarchical Diagram.

The class `igstk::ImageReader` is the superclass of all the image reader classes in IGSTK. This class is templated by the `igstk::ImageSpatialObject`. Figure 14.1 shows the class hierarchy diagram of the image reader classes. As shown in the figure, the DICOM image reader class (`igstk::DICOMImageReader`) is a template class derived from `igstk::ImageReader`. This class encapsulates ITK DICOM image reader classes in a restrictive API controlled

by a state machine. The ITK DICOM image I/O classes internally use the GDCM library. GDCM is an open source DICOM library that was developed and maintained by the CREATIS (Research and Applications Center in Image and Signal Processing) Team at INSA-Lyon [2]. Image readers for CT, MRI, and Ultrasound modalities are then derived from the `igstk::DICOMImageReader` by instantiating with the appropriate spatial object type. For example, the MRI image reader class is derived from a `DICOMImageReader` class instantiated over a MRI image spatial object.

14.1.1 State Machine Design

Figure 14.2 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.
5. *ImageRead*: DICOM image is read.

14.1.2 Component Interface

The following methods are available in the public interface:

1. *RequestSetDirectory*: Sets the directory name containing the DICOM data.
2. *RequestReadImage*: Requests image reading.
3. *RequestGetPatientNameInformation*: Requests the reader to throw patient name information-loaded event.
4. *RequestGetModalityInformation*: Requests the reader to generate a modality information-loaded event.

14.1.3 Special Features

The DICOM image readers check the validity of the input DICOM data to avoid incorrect reading and reconstruction of 3D volumes. One issue is gantry tilt. The `itk::OrientedImage` class that is used internally by `igstk::DICOMImageReader` handles only DICOM data acquired in

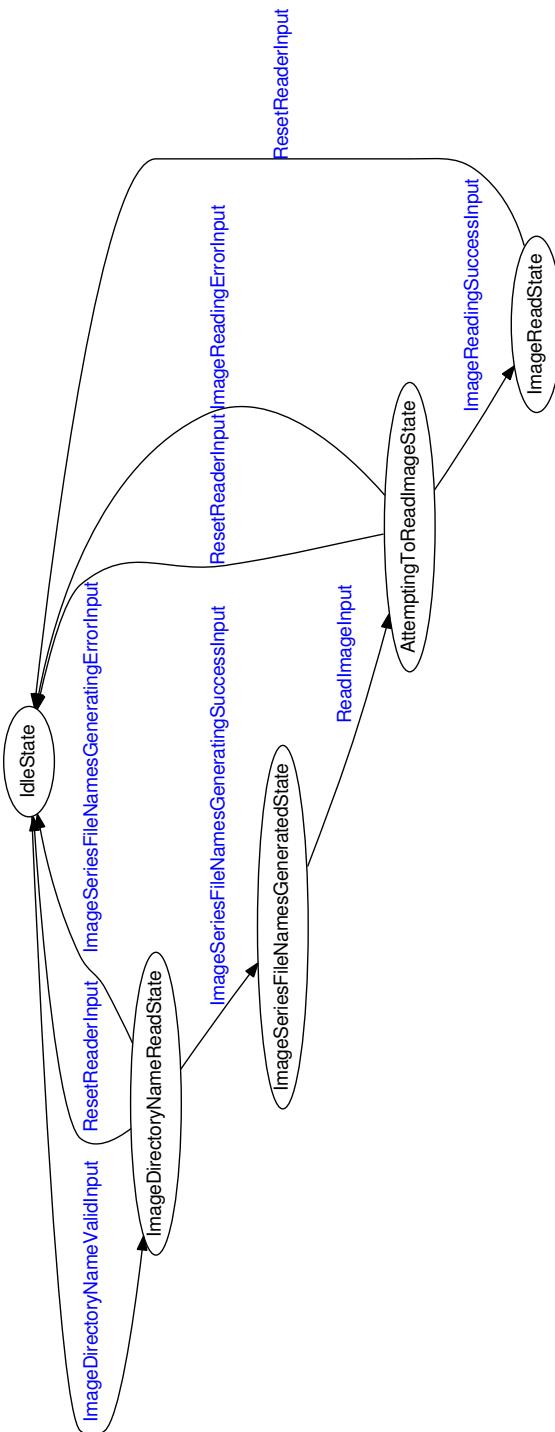


Figure 14.2: State Machine of the DICOMImageReader Class.

3D orthogonal space. In the preprocessing stage, the DICOM image reader checks the value of the gantry tilt. If the gantry tilt is greater than a threshold value, the reader throws an image invalid error event.

Similarly, the modality 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 one attempts to read CT image data using an MRI image reader, then an invalid image reading error event will be generated.

14.1.4 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 defined first. 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, a 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:
};
```

Similar callback classes need to be defined to observe patient name and image reading errors.

In this example, we would like to read a CT image. First, a CT image reader object is instantiated as follows:

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

A logger can be linked to the reader, as follows:

```
reader->SetLogger( logger );
```

The DICOM image directory is set as:

```
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();
```

A similar operation can be performed to access the patient name:

```
/* Add observer to listen to patient name info */
typedef DICOMImagePatientNameInformationCallback PatientCallbackType;

PatientCallbackType::Pointer dipnbc = PatientCallbackType::New();
reader->AddObserver( igstk::DICOMPatientNameEvent(), dipnbc );
reader->RequestGetPatientNameInformation();
```

14.2 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.

14.3 Conclusion

IGSTK provides classes to read DICOM datasets. The reader classes are ITK image I/O classes encapsulated in a restrictive API governed by a state machine. The reader classes perform validity checks on the DICOM data before loading it. A capability to capture and save screen shots to a file is also provided. This feature is implemented using a `vtkWindowToImage` filter.

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 a 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 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 the landmark registration error `igstk::igstkLandmarkRegistrationErrorPredictor`.

15.1 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 [14]. He developed a closed-form solution to the least-squares problem of computing transformation parameters between two coordinate systems.

15.1.1 State Machine Design

Figure 15.1 illustrates the state diagram for `igstk::Landmark3DRegistration`.

This class has the following states:

1. *Idle*: No landmark points added.

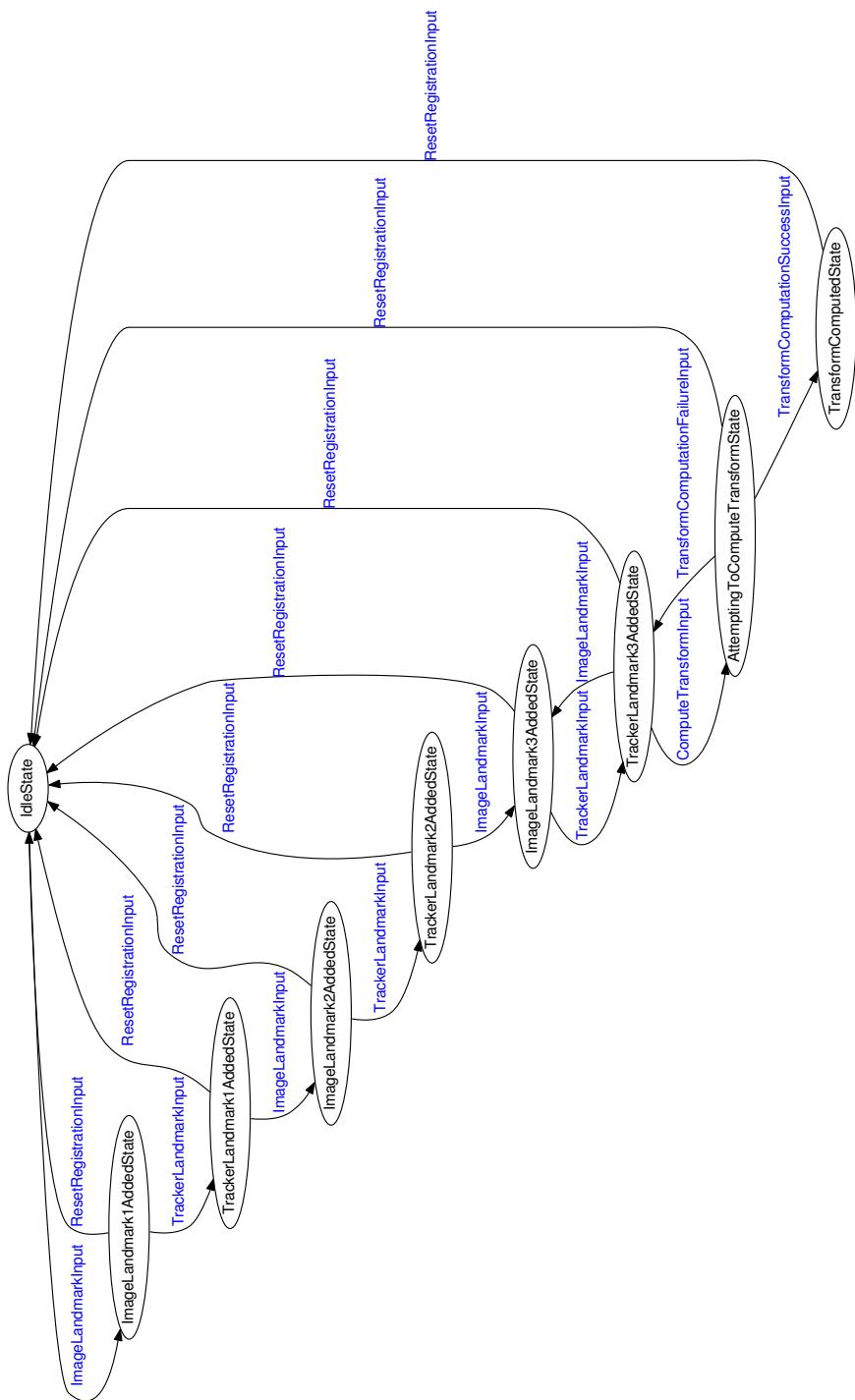


Figure 15.1: Landmark Registration State Machine Diagram.

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 *TrackerLandmark3Added* states as shown in the state diagram. To avoid human error in establishing the coordinates, 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).

15.1.2 Component Interface

The following methods are available in the public interface:

1. *RequestAddImageLandmarkPoint(LandmarkPointType)*: Adds image landmark point.
2. *RequestAddTrackerLandmarkPoint(LandmarkPointType)*: Adds tracker landmark point.
3. *RequestResetRegistration()*: Resets the state machine to idle state.
4. *RequestComputeTransform()*: Requests transform computation.
5. *RequestGetTransform()*: Requests transform parameters. The transform parameters are returned to the requesting application as a string-loaded event `igstk::TransformModifiedEvent`. The user needs to set up a callback to observe this transform event.
6. *ComputeRMSError()*: Computes root mean square of landmark registration.

All of the public interface methods are governed by the state machine, except `ComputeRMSError()`.

15.1.3 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"
#include "igstkCoordinateSystemTransformToResult.h"
```

To fully utilize the registration component, callbacks need to be set up to observe 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::CoordinateSystemTransformToEvent` event. This event is loaded with transform parameters that are computed by the registration component. The commands are as follows:

```
class Landmark3DRegistrationGetTransformCallback: public itk::Command
{
public:
    typedef Landmark3DRegistrationGetTransformCallback      Self;
    typedef itk::SmartPointer<Self>                         Pointer;
    typedef itk::Command                                     Superclass;
    itkNewMacro(Self);

    typedef igstk::CoordinateSystemTransformToEvent TransformEventType;

    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 TransformEventType * transformEvent =
            dynamic_cast < const TransformEventType* > ( &event );

        const igstk::CoordinateSystemTransformToResult transformCarrier =
            transformEvent->Get();

        m_Transform = transformCarrier.GetTransform();

        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;
};
```

For more information on IGSTK events, see Chapter 7. After the helper classes are defined, the main function implementation is started, as follows:

```
int main( int argc, char * argv[] )  
{
```

Next, all the necessary data types are defined:

```
typedef igstk::Object::LoggerType           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;
```

Then, the registration component is instantiated as follows:

```
Landmark3DRegistrationType::Pointer landmarkRegister =  
    Landmark3DRegistrationType::New();
```

Next, the landmark containers that hold the landmark image and tracker coordinates are instantiated:

```
LandmarkPointContainerType  imagePointContainer;  
LandmarkPointContainerType  trackerPointContainer;
```

Then, 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::InvalidRequestErrorEvent  InvalidRequestEvent;
```

```
landmarkRegister->AddObserver( InvalidRequestEvent(), lrcb );  
  
Landmark3DRegistrationErrorCallback::Pointer ecb =  
    Landmark3DRegistrationErrorCallback::New();  
typedef igstk::Landmark3DRegistration::TransformComputationFailureEvent  
    ComputationFailureEvent;  
landmarkRegister->AddObserver( ComputationFailureEvent(), ecb );
```

A logger can then 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( LoggerType::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 followed by the tracker coordinates. This design choice is consistent with the “safety by design” philosophy of IGSTK. The commands are as follows:

```
// 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 landmark coordinates are added, the transform computation is requested as follows:

```
landmarkRegister->RequestComputeTransform();
```

To access the transform parameters, a `GetTransform` callback is instantiated to observe the transform event, as follows:

```

Landmark3DRegistrationGetTransformCallback::Pointer lrtcb =
Landmark3DRegistrationGetTransformCallback::New();

landmarkRegister->AddObserver(
    igstk::CoordinateSystemTransformToEvent(), lrtcb );

```

To request that the registration component throw an event loaded with transform parameters, a `RequestGetTransform` function is invoked as follows:

```
landmarkRegister->RequestGetTransformFromTrackerToImage();
std::cout << "Transform " << transform << std::cout;
```

15.2 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 (RMS) error between corresponding fiducials after registration. However, it has been shown that the fiducial registration error is independent of fiducial configuration and is a poor predictor of registration

accuracy [27]. Consequently, West et al. [27] have derived a more robust error predictor that takes fiducial configuration into account, as shown in Equation 15.1. IGSTK has implemented this error prediction algorithm (`igstk::Landmark3DRegistrationErrorEstimator`) as follows:

$$TRE^2(t) = \frac{FRE^2}{N-2} \left(1 + \frac{1}{3} \sum_{k=1}^3 \frac{d_k^2}{f_k^2} \right) \quad (15.1)$$

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.

15.2.1 State Machine Design

Figure 15.2 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. *AttemptingToEstimateTargetRegistrationError*: Transition state to estimate target point registration error.
8. *TargetRegistrationErrorEstimated* : Target point registration error estimated.

15.2.2 Component Interface

The following methods are available in the public interface:

1. *RequestSetLandmarkContainer(LandmarkContainerType)*: Sets the landmark container.
2. *RequestSetTargetPoint(TargetPointType)*: Sets the target point.
3. *RequestSetLandmarkRegistrationError(ErrorType)*: Sets the landmark registration error.

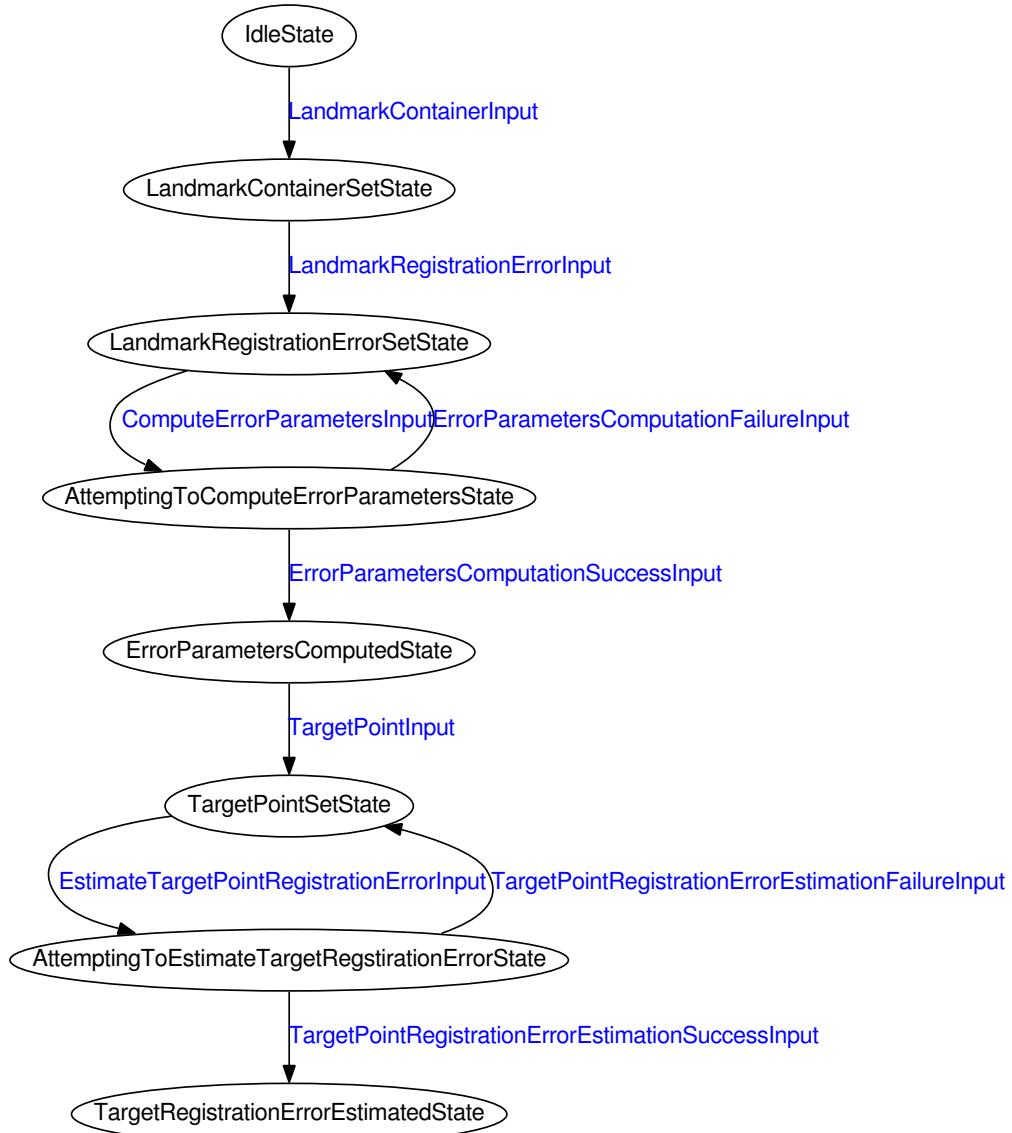


Figure 15.2: Landmark Registration Error Estimator State Machine Diagram.

4. *RequestComputeErrorParameters()*: Requests computation of error parameters needed to compute the target point registration error.
5. *RequestEstimateTargetPointRegistrationError()*: Requests estimation of target point registration.
6. *RequestGetTargetPointRegistrationErrorEstimate()* : Gets target point registration error. This method throws an event loaded with the target point registration error.

15.2.3 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. [27]. The target point registration error is dependent on the location of the target point, the registration error of the landmark points (RMS 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, as follows:

```
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 is basically the RMS error of the landmark registration. The commands are as follows:

```
Landmark3DRegistrationGetRMSErrorCallback::Pointer lRmscb =  
    Landmark3DRegistrationGetRMSErrorCallback::New();  
  
landmarkRegister->AddObserver( igstk::DoubleTypeEvent(), lRmscb );  
landmarkRegister->RequestGetRMSError();
```

```

landmarkRegistrationError = lRmscb->GetRMSError();

errorEstimator->RequestSetLandmarkRegistrationError(
    landmarkRegistrationError );

```

Next, the target point, which we will use for 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 << "LandmarkRegistrationErrorEstimator class failed to "
        << "throw a landmark registration error event" << std::endl;
    return EXIT_FAILURE;
}

targetRegistrationError = lrtcb->GetError();

```

15.3 Conclusion

Registration is an essential component in image-guided surgery applications. IGSTK provides a 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 an actual application. For this purpose, IGSTK provides a robust error prediction tool that also takes landmark configuration into account.

Calibration

The goal of an image-guided surgery application is to guide the physician by accurately displaying the spatial relationship between anatomical structures and manipulated tools. This is most often achieved by instrumenting tools and anatomy with markers that are tracked by a localization system. As the tools and anatomical structures are tracked indirectly, we need to adjust the measurements reported by the localizer so that they account for the transformation between the object's internal coordinate system and the tracked coordinate system. This adjustment is most often referred to as spatial calibration when the object of interest is an instrument, or registration when the object is an anatomical structure.

Most often calibration is performed in advance with the information stored for use at a later time. Currently, IGSTK supports the storage of several transformation types, and provides classes for reading and writing the data into an XML file. In addition, IGSTK provides classes for performing the standard pivot calibration [22]. While this calibration approach is primarily used for localizing the tip of a pointer tool in the tracked coordinate system, it has also been utilized for localizing the center of the femoral head in orthopedic interventions [19].

16.1 Precomputed Transformations, Reading and Writing Files

To enable the use of precomputed transformation data IGSTK uses XML files with the following tags and attributes:

1. `precomputed_transform` - Root element of the XML file.
2. `description` - Human readable description of the transformation.
3. `computation_date` - Human readable date and time when the transformation was estimated.
4. `transformation` - The actual transformation data, whose format varies according to transformation.

5. estimation error - Transformation attribute denoting the error associated with the transformation.

IGSTK supports three types of transformations, rigid, affine, and perspective. The rigid transformation is parameterized as $[\mathbf{q}, \mathbf{t}]$, where $\mathbf{q} = [q_x, q_y, q_z, q_w]$ is a unit quaternion and \mathbf{t} is a 3×1 translation vector. The affine transformation is parameterized by $[A, \mathbf{t}]$, where A is a 3×3 matrix and \mathbf{t} is a 3×1 vector. The perspective transformation is parameterized by $[K, R, \mathbf{t}]$, where K is a 3×3 upper triangular matrix, describing intrinsic camera parameters, and R, \mathbf{t} are a 3×3 rotation matrix and a 3×1 translation vector, representing the extrinsic camera parameters.

An example of a rigid transformation XML file:

```
<precomputed_transform>

<description>
    MR/CT transformation for patient X
</description>

<computation_date>
    Thursday July 4 12:00:00 1776
</computation_date>

<transformation estimation_error = "0.5">
    0 0 0 1 0 0 0
</transformation>

</precomputed_transform>
```

Examples of all three transformation formats can be found in the IGSTK source directory under *Testing/Data/Input/TransformData*.

To load the transformation data from an XML file we use the `TransformFileReader` class. The specific transformation type is determined by the XML reader that is set using the `RequestSetReader()` method. Available XML readers include `RigidTransformXMLFileReader`, `AffineTransformXMLFileReader`, and `PerspectiveTransformXMLFileReader`. We now show how these classes work together to load a precomputed transformation:

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

This example illustrates how to use xml readers to load precomputed transformations. These are either tool calibrations or results of pre-operative registration (e.g. MR/CT). The program receives an xml file containing a precomputed transformation, reads it, presents the user with the general information about the transformation (when computed and estimation error). The user is then asked if they wish to use the transformation or not. If the answer is yes, the program prints the transformation and exits, otherwise it just exits.

To use the xml readers and the precomputed transform data type we need to include the following files:

```
#include "igstkPrecomputedTransformData.h"
#include "igstkTransformFileReader.h"
#include "igstkPerspectiveTransformXMLFileReader.h"
#include "igstkAffineTransformXMLFileReader.h"
#include "igstkRigidTransformXMLFileReader.h"
```

Instantiate the transformation file reader which is parameterized using the xml reader corresponding to the relevant transformation type.

```
igstk::TransformFileReader::Pointer transformFileReader =
    igstk::TransformFileReader::New();
```

We are interested in a rigid transformation, so instantiate that xml reader.

```
xmlFileReader = igstk::RigidTransformXMLFileReader::New();
```

Set the reader and the name of the file we want to read.

```
transformFileReader->RequestSetReader( xmlFileReader );
transformFileReader->RequestSetFileName( argv[2] );
```

Try to read. The success or failure of this attempt are observed using the standard IGSTK observer-command approach.

```
transformFileReader->RequestRead();
```

Finally, a corresponding set of classes used to write transformation XML files is also available (`TransformFileWriter`, `RigidTransformXMLFileWriter`,...). These work in a similar manner to the readers.

16.2 Pivot Calibration

We are interested in the location of a tool's tip relative to a tracked coordinate system. By rotating the tool while its tip location remains fixed, we compute both the tip location relative to the tracked coordinate system and the fixed point's coordinates in the tracker's coordinate system.

Every transformation $[R_i, \mathbf{t}_i]$ acquired by the tracker during the rotation yields three equations in six unknowns, $R_i \mathbf{t}_{\text{tip}} + \mathbf{t}_i = \mathbf{t}_{\text{fixed}}$, with the unknown 3x1 vectors \mathbf{t}_{tip} and $\mathbf{t}_{\text{fixed}}$. Figure 16.1 shows the coordinate systems and transformations.

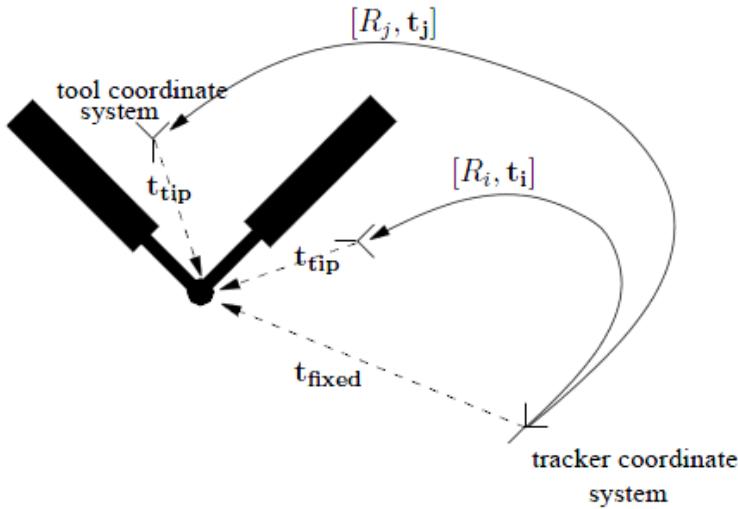


Figure 16.1: Pivot Calibration Coordinate Systems and Transformations. Solid arrows denote known transformations, dashed arrows unknown transformations.

The solution is obtained in a least squares manner by acquiring multiple transformation while rotating the tool around its tip resulting in the following overdetermined equation system:

$$\begin{bmatrix} R_0 & -I \\ \vdots & -I \\ R_n & -I \end{bmatrix} \begin{bmatrix} t_{tip} \\ t_{fixed} \end{bmatrix} = \begin{bmatrix} -t_0 \\ \vdots \\ -t_n \end{bmatrix}$$

This equation system is solved using the pseudoinverse [25].

Another method of obtaining our unknown translations, not implemented in IGSTK, is to first fit a sphere to the set of points t_i , the center of this sphere is t_{fixed} . The remaining unknowns are estimated as $t_{tip} = R_i^T t_{fixed} - R_i^T t_i$.

The implementation of the pivot calibration is divided into three classes:

1. `PivotCalibrationAlgorithm` - This is the implementation of the method described above.
2. `PivotCalibration` - This class is responsible for acquiring a prescribed number of transformations for a given tool. It uses the algorithm class to compute the unknowns.
3. `PivotCalibrationFLTKWidget` - A class that provides an FLTK wrapper around the pivot calibration class, providing visual feedback with regard to data collection (progress bar) and results of the calibration.

It should be noted that all three classes work with generic tracker tools and can thus be used for performing pivot calibration for all IGSTK supported tracking systems. An example program showing the use of the FLTK widget for pivot calibration with the Polaris tracking system (Northern Digital Inc., Waterloo Ontario, Canada) can be found in *Examples/PivotCalibrationFLTKWidget*. The application UI after calibration was performed is shown in Figure 16.2. The application allows the user to select a tool, specify the number of transformations to acquire, and save the resulting calibration in an XML file. Data gathering starts after a user specified delay.

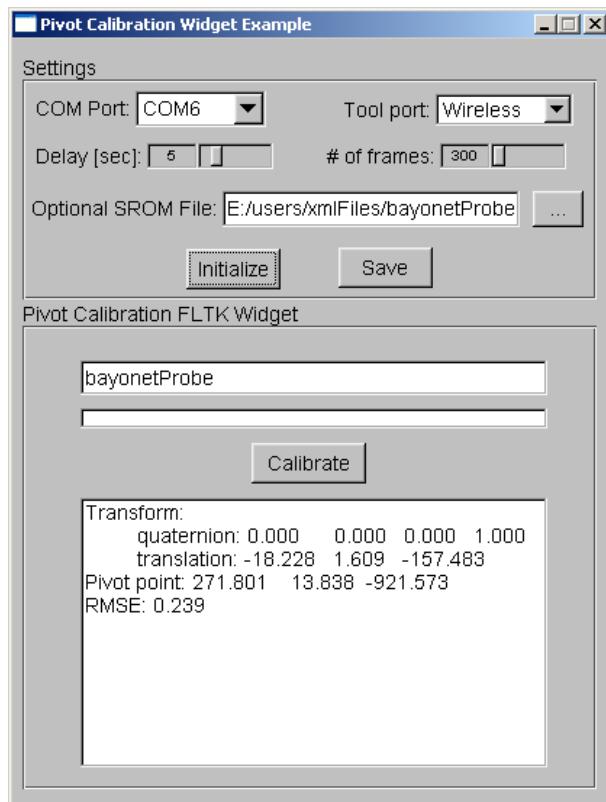


Figure 16.2: FLTK Pivot Calibration Example Application. Screen shot acquired after performing a successful calibration. Bottom part of the application consists of the PivotCalibrationFLTKWidget which can be embedded in any FLTK UI.

Reslicing

17.1 General Background

Three-dimensional image data are used extensively in image-guided navigation systems. Often times, clinicians have a need to visualize the input image volume at different orientations and viewpoints to be able to perform their procedures efficiently and effectively. We will refer to this operation of visualizing a two dimensional "cut through" of a three dimensional image volume as "reslicing" in this book. The reslicing plane can be arbitrarily oriented and positioned in the 3D image space.

IGSTK provides an implementation for reslicing as described here. The reslicing component contains two types of classes: reslice plane spatial object (`igstk::ReslicerPlaneSpatialObject`) and reslice object representation classes (`igstk::ImageResliceObjectRepresentation` and `igstk::MeshResliceObjectRepresentation`). The reslicing plane spatial object defines the orientation and position of the reslicing plane. A graphical representation of the resliced 2D image is generated using the reslice object representation classes. The representation classes basically describe how the object should be displayed on screen. Currently, IGSTK provides representation classes for image and mesh data types.

17.2 Design

17.2.1 Class Hierarchy

Class hierarchy diagrams for the reslicing classes are shown in Figures 17.1–17.3.

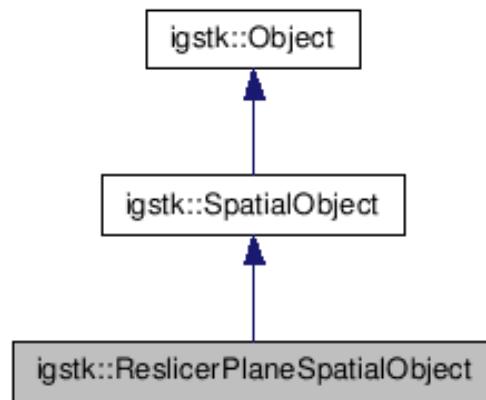


Figure 17.1: ReslicerPlaneSpatialObject Class Hierarchy.

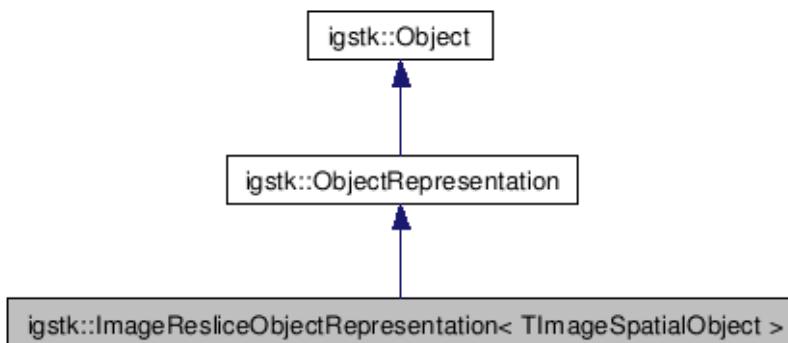


Figure 17.2: ImageResliceObjectRepresentation Class Hierarchy.

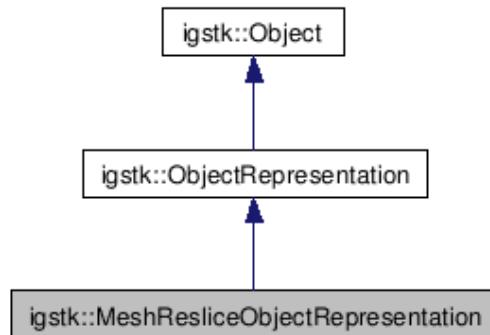


Figure 17.3: MeshResliceObjectRepresentation Class Hierarchy.

17.2.2 Manual and Automatic Reslicing

The `igstk::ReslicerPlaneSpatialObject` provides manual and automatic ways of reslicing. In the manual mode, class users specify spatial position of the reslicing plane (x,y,z) explicitly using the provided method in the API (`RequestSetCursorPosition()`). Position coordinates are usually specified using a mouse pointer or slider inputs in a GUI-based application using the reslicing operation. The orientation of the reslicing plane can be set to axial, coronal, or sagittal.

In the automatic mode, the reslicing plane position and orientation are retrieved from the tracking tool transforms. The position of the reslicing plane is automatically updated with the change in the position of the tracking tool. In typical applications, both manual and automatic operations are made available to the clinicians. Clinicians will select the best option depending on the procedure.

17.2.3 Reslicing Modes

The position and orientation of the reslicing plane can be set in different ways. To be able to handle different scenarios, three types of reslicing modes are available in `ReslicerPlaneSpatialObject` : Orthogonal, OffOrthogonal and Oblique.

In Orthogonal mode (`Reslicing::Orthogonal`), the tool tip spatial object provides the reslicing plane position and orientation information is extracted from the bounding box extracted from the input image or mesh data. In this mode, the three mutually orthogonal orientations are named: Axial, Sagittal and Coronal.

In OffOrthogonal mode, the orientation of the reslicing plane is computed using the tracker tool spatial object and the bounding box specified. In this mode, the three mutually orthogonal orientations are named : OffAxial, OffSagittal and OffCoronal.

Lastly, in Oblique mode, both orientation and position information of the reslicing plane are obtained from the tracker tool. For this mode, the three orientations are: `PlaneOrientationWithZAxesNormal`, `PlaneOrientationWithYAxesNormal`, and `PlaneOrientationWithXAxesNormal`.

The `igstk::ReslicerPlaneSpatialObject` class provides methods in the public API to set reslicing mode type and plane orientation.

17.2.4 Image and Mesh Data Reslicing

As discussed above, the IGSTK reslicing component provides support for image and mesh data reslicing. In both cases, `igstk::ReslicerPlaneSpatialObject` is used to provide the reslicing plane. For image data reslicing, texture interpolation technique is used to extract the resliced 2D image. The VTK class `vtkImageReslice` is used internally for this purpose. Three different types of texture interpolation can be used in the class. Ordered by increasing complexity and appearance quality, these are: `VTK_NEAREST_RESLICE`, `VTK_LINEAR_RESLICE`, `VTK_CUBIC_RESLICE`. The application designer should choose the best approach based on

available computing power and overall performance.

For mesh input data, the reslicing plane is used to reslice the mesh data. The output of mesh reslicing is a contour resulting from the intersection between the reslicing plane and the mesh data. The resliced mesh data is a textured polygon. Note that depending on the reslicing plane position and orientation, the resulting polygon may have more than four sides. Furthermore, the resulting contour may be open or have multiple parts.

17.2.5 State Machine Diagrams

`igstk::ReslicerPlaneSpatialObject` State Machine

Figure 17.4 illustrates the state diagram for `igstk::ReslicerPlaneSpatialObject`.

`igstk::ImageResliceObjectRepresentation` State Machine

Figure 17.5 illustrates the state diagram for `igstk::ImageResliceObjectRepresentation`.

`igstk::MeshResliceObjectRepresentation` State Machine

Figure 17.6 illustrates the state diagram for `igstk::MeshResliceObjectRepresentation`.

17.2.6 Component Interface

`igstk::ReslicerPlaneSpatialObject`

The following methods are available in the public interface:

1. `RequestSetReslicingMode()`: request the internal state machine to attempt to set the reslicing mode.
2. `RequestSetOrientationType()`: request the internal state machine to attempt to set the orientation type.
3. `RequestSetBoundingBoxProviderSpatialObject()` : request the state machine to set the bounding box provider spatial object (an image or a mesh).
4. `RequestSetToolSpatialObject()`: request the internal state machine to attempt to set the tool spatial object. A tool spatial object is optional for Orthogonal modes. However, it is required for the OffOrthogonal and Oblique reslicing modes.
5. `RequestSetCursorPosition()`: request the internal state machine to attempt to set reslicing cursor position.

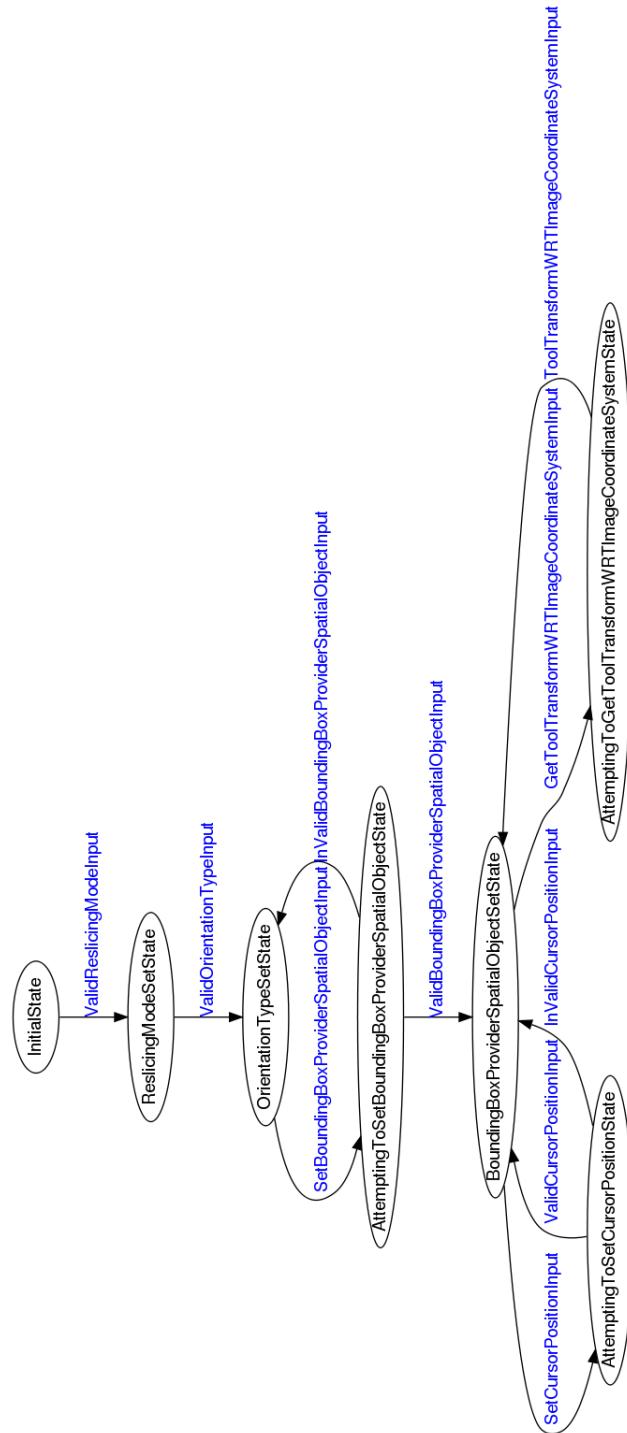


Figure 17.4: ReslicerPlaneSpatialObject State Machine Diagram.

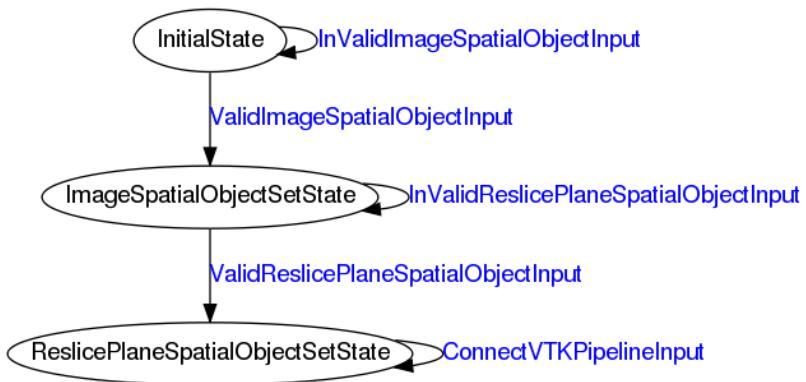


Figure 17.5: ImageResliceObjectReperesentation State Machine Diagram.

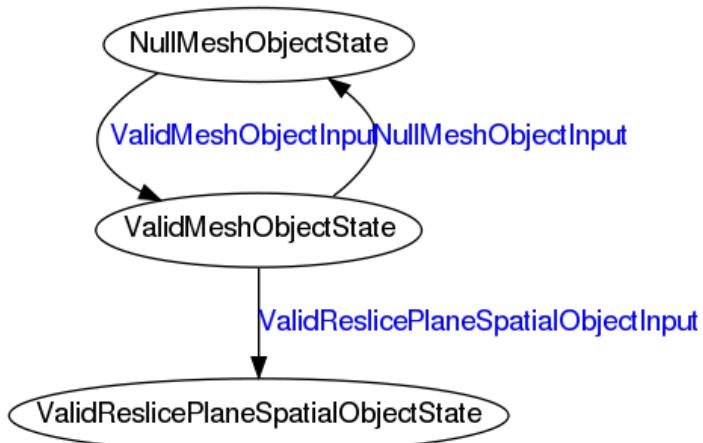


Figure 17.6: MeshResliceObjectReperesentation State Machine Diagram.

6. RequestGetReslicingPlaneParameters(): request to get reslicing plane parameters.
7. RequestComputeReslicingPlane(): request compute reslicing plane.

igstk::ImageResliceObjectRepresentation

The following methods are available in the public interface:

1. RequestSetReslicePlaneSpatialObject(): request set the reslicing plane spatial object.
2. RequestsetImageSpatialObject(): request set the input image spatial object that would be resliced.
3. SetWindowLevel(): set window level of the resliced image representation.
4. SetFrameColor(): set frame color of the representation class.

igstk::MeshResliceObjectRepresentation

1. RequestSetReslicePlaneSpatialObject(): request set the reslicing plane spatial object.
2. RequestSetMeshObject(): request set the input meshspatial object that would be resliced.
3. SetVisibility(): Turn on/off visibility of the representation.
4. Set/GetLineWidth(): Set/Get width of a line property.

17.3 Examples

The source code for this section can be found in the file
Examples/Reslicing/Reslicing1.cxx.

This example illustrates how to perform orthogonal reslicing of CT image data using a tool spatial object.

Add the necessary reslicing class header files.

```
#include "igstkReslicerPlaneSpatialObject.h"
#include "igstkImageResliceObjectRepresentation.h"
```

Instantiate image spatial object to store the CT image data.

```
ImageSpatialObjectType::Pointer imageSpatialObject = ImageSpatialObjectType::New();
imageSpatialObject = ctImageObserver->GetCTImage();
```

Instantiate image reslice object representation object and set rendering properties such as window level.

```
typedef igstk::ImageResliceObjectRepresentation< ImageSpatialObjectType >
    ImageResliceRepresentationType;

ImageResliceRepresentationType::Pointer imageResliceRepresentation =
    ImageResliceRepresentationType::New();

imageResliceRepresentation->SetWindowLevel( 1559, -244 );
```

Set the input image spatial object to the reslice object representation object.

```
imageResliceRepresentation->RequestSetImageSpatialObject( imageSpatialObject );
```

Build the tool spatial object that will be used to reslice the input image. CylinderObject is used for this purpose.

```
typedef igstk::CylinderObject                               ToolSpatialObjectType;
ToolSpatialObjectType::Pointer toolSpatialObject = ToolSpatialObjectType::New();
toolSpatialObject->SetRadius( 0.1 );
toolSpatialObject->SetHeight( 2.0 );
```

Instantiate a reslicer plane spatial object.

```
typedef igstk::ReslicerPlaneSpatialObject                  ReslicerPlaneType;
ReslicerPlaneType::Pointer reslicerPlaneSpatialObject = ReslicerPlaneType::New();
```

Set the reslicing mode type using RequestSetReslicingMode method. For example, Orthogonal reslicing mode is selected as follows.

```
reslicerPlaneSpatialObject->RequestSetReslicingMode( ReslicerPlaneType::Orthogonal );
```

Set the plane orientation using RequestSetOrientationType method.

```
reslicerPlaneSpatialObject->RequestSetOrientationType( ReslicerPlaneType::Axial );
```

Reslicer plane spatial object extent needs to be defined. The extent is defined using input image spatial object or other spatial object. To specify the spatial object that is needed to define the bounding box, use RequestSetBoundingBoxProviderSpatialObject method.

```
reslicerPlaneSpatialObject->RequestSetBoundingBoxProviderSpatialObject(
    imageSpatialObject );
```

Set the reslicer plane spatial object to the image representation.

```
imageResliceRepresentation->RequestSetReslicePlaneSpatialObject(  
    reslicerPlaneSpatialObject );
```

Then, set the tool spatial object that will be used to reslice the input image.

```
reslicerPlaneSpatialObject->RequestSetToolSpatialObject(  
    toolSpatialObject );
```

Then, add the reslice image representation to the view similar to the other image representation objects. In this 2D view, the resliced image will be displayed.

```
view2D->RequestAddObject( imageResliceRepresentation );
```

17.4 Conclusion

Effective visual feedback is essential for the success of interventional procedures. IGSTK reslicing component plays a key role in this regard. Clinicians have a need for viewing their surgical instruments at different orientations in the surgical field. For this, IGSTK reslicing classes provide the necessary implementations to be able to choose different reslicing modes and plane orientations.

Videolmager Component

“Learn to see, and then you’ll know that there is no end to the new worlds of our vision.”

—Carlos Castaneda

Image-guided navigation systems utilize pre-operative and intra-operative images to guide the physician during an intervention. A prerequisite is that the images be transferred to the guidance system. Pre-operative images are commonly stored and transferred using the DICOM protocol. Intra-operative images, most often, are acquired with apparatus that does not support the DICOM protocol. As a result, transferring these images from the apparatus to the navigation system requires the use of a VideoImager component that translates the images into a format that can be processed by the navigation system. Among others, these images include video acquired with endoscopic systems, ultrasound (US) systems, and X-ray fluoroscopy. Once transferred to the navigation system they are used in a variety of tasks including registration, aligning the pre-operative images to the patient location, 3D US using a 2D system and tracking, augmented reality, and more.

The IGSTK VideoImager component provides an abstract model for concrete video devices and acts as a connection between the specific video acquisition hardware and other IGSTK components. The abstract model is independent of the video acquisition hardware with sub classes providing the hardware specific implementations.

18.1 The Role of the Videolmager Component

The IGSTK VideoImager component communicates with specific video producing devices, such as US or endoscopy systems and provides other IGSTK components with a video stream. The corresponding spatial object `igstk::VideoFrameSpatialObject` and representation `igstk::VideoFrameRepresentation` are responsible for rendering the video stream into the IGSTK scene.

18.2 Structure of the Videolmager Component

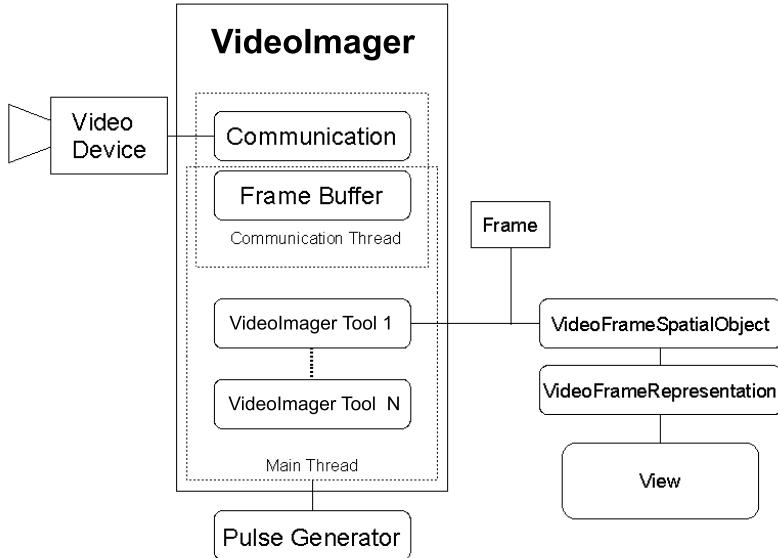


Figure 18.1: The IGSTK Videolmager Component.

The `igstk::VideoImager` describes a generic interface for acquiring video from concrete image acquisition systems such as US, endoscopy, bronchoscopy, etc. One `igstk::VideoImager` object manages one or more `igstk::VideoImagerTool` objects, where the `igstk::VideoImager` object is responsible for communication and the `igstk::VideoImagerTool` is responsible for specific settings dependent upon the connected video device. Using one video device the `igstk::VideoImager` and `igstk::VideoImagerTool` represent one device, otherwise the `igstk::VideoImager` represents a multiple video acquisition device (i.e. frame grabber or video hub) and the `igstk::VideoImagerTool` is one of multiple video streams connected to the device.

The VideoImager communicates continuously in a separate thread with the acquisition hardware and stores each frame temporarily in a tool frame buffer. The tool frame buffer contains for each VideoImagerTool, which is identifiable through a unique name, a separate frame. Every frame is stored in a `igstk::Frame` object. These `igstk::Frame` objects are not used immediately as they are received, rather they are buffered in the tool frame buffer until the main thread wants to update the VideoImagerTools i.e. until the PulseGenerator generates a pulse to update the VideoImager. Whenever the PulseGenerator sends a pulse event to the Pulse-Generator-Observer the "UpdateStatus" callback method is invoked for updating the status of the VideoImagerTools. The VideoImager updates the connected VideoImagerTools according to a specific frequency. The function `RequestSetFrequency()` sets the frequency for the `igstk::PulseGenerator` which triggers the main update process.

To render the video frame, a `igstk::VideoFrameSpatialObject` instance has to be connected to a `igstk::VideoImagerTool` instance. The `igstk::VideoFrameSpatialObject` instance is connected again to a `igstk::VideoFrameRepresentation` instance. Finally, a `igstk::View` instance can render the representation object in a GUI window. The `igstk::VideoFrameSpatialObject` and `igstk::VideoImagerTool` communicate using `igstk::Frame` objects (see Figure 18.1). Device specific implementations are derived from these generic classes. An example that requires minimal hardware is the `igstk::WebcamWinVideoImager` and `igstk::WebcamWinVideoImagerTool` classes that acquire images from a webcam.

18.2.1 Threading

The `igstk::VideoImager` object implements a dedicated communication thread to capture video from specific devices. This communication thread is linked continuously with the video device and stores captured frames from possibly multiple video devices in a tool frame buffer. Figure 18.1 shows the two threads involved inside dotted rectangles. Since the tool frame buffer contains data that is shared between the communication thread (within which the `VideoImager` places data into the buffer) and the main IGSTK execution thread (within which the `VideoImager` extracts data from the buffer), the `VideoImagerBuffer` has a mutual exclusion lock to ensure that the main thread does not attempt to read a data record that the communication thread has only partially written. One should consider that the communication thread is only active in the Imaging state.

18.2.2 Buffering

As stated in Section 18.2.1, an extra thread communicates with the video device and continuously acquires frames from one or more `VideoImagerTools`. The `igstk::VideoImager` class stores every frame in a tool frame buffer. Every time the `PulseGenerator` triggers an update the current frame from each video device is stored into the appropriate ring buffer of the `igstk::VideoImagerTool`. The ring buffer immediately increments its index pointing to the current place, where the next frame will be stored. Every time the function `igstk::VideoImagerTool::GetTemporalCalibratedFrame()` is called the `igstk::VideoImagerTool` will return the frame at the current index position minus a user specified offset. This offset can be set with `igstk::VideoImagerTool::SetDelay(unsigned int)`. This is a useful feature for temporally synchronizing multiple video sources. In cases where there are different lag times transferring video from the various imaging apparatus to the navigation computer, one needs to temporally synchronize the video streams to provide a consistent display. In our implementation the temporal consistency is limited to a discrete delay specified by an integral offset, even though the offset is actually continuous and may require the use of a sub frame offset.

18.2.3 Frames and Timestamps

Every time the communication thread grabs a frame from the video stream it stores the current frame and a timestamp in `igstk::Frame` object, which contains:

- a image taken at a certain time
- a timestamp that gives the time at which the frame was captured
- an expiration time after which the image will be considered invalid

Since each frame is associated with a timestamp the `igstk::VideoFrameRepresentation` can decide if the frame is valid at render time. Depending on that the video stream is visible or not. The validity time of a frame depends on the device frequency and is calculated as follows:

Validity time for frame = $1000/\text{frequency}$

e.g.: $1000/25 \text{ Hz} = 40 \text{ ms}$ That means that every 40 ms the video stream should provide the VideoImager with a new frame. To prevent flickering, 10 ms overlap time is added so that a `igstk::Frame` is valid for 50 ms after it was generated. After that time it expires. Meanwhile another frame should arrive. If not, the representation of the frame turns to invisible.

18.3 Class Hierarchy

The VideoImager component consists of two main super classes `igstk::VideoImager` and `igstk::VideoImagerTool`. Hardware specific classes are derived from these classes. Figure 18.2 and Figure 18.3 show the class hierarchy diagram for the `igstk::VideoImager` and `igstk::VideoImagerTool`.

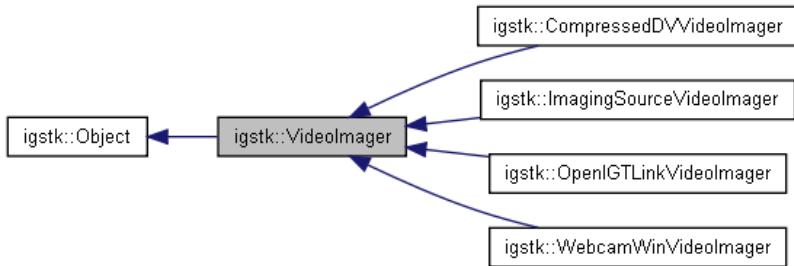


Figure 18.2: Videolmager Class Hierarchy.

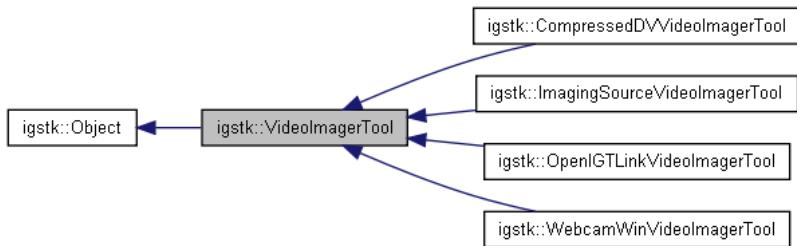


Figure 18.3: VideoImagerTool Class Hierarchy.

18.3.1 VideoImager

State Machine

Figure 18.4 illustrates the state machine diagram of the `igstk::VideoImager`.

The major states are as follows:

1. *Idle* : Initial state. The VideoImager will return to this state after the `RequestClose()` method is called.
2. *CommunicationEstablished* : Communication is established. This state is entered if a call to the VideoImager's `RequestOpen()` was successful. The VideoImager can be returned to this state from any state except the *Idle* state by calling the `RequestReset()` method.
3. *VideoImagerToolAttached*: VideoImager tool is attached to the VideoImager.
4. *Imaging* : VideoImager component captures frames from a video stream. This state is entered if a call to `RequestStartImaging()` was successful.

Interface Methods

The public interface to the VideoImager component consists of the following methods:

1. `RequestOpen()` : Initiate communication with the video device.
2. `RequestClose()` : Close communication with the video device.
3. `RequestReset()` : Attempt to reset the video device to its initial state.
4. `RequestStartImaging()` : Put the device into its imaging state, from which it will grab video frames from each tool. In this state, each VideoImagerTool buffer will be continuously updated with new frames from one or more imaging devices.
5. `RequestStopImaging()`: Stop VideoImager frame acquisition.

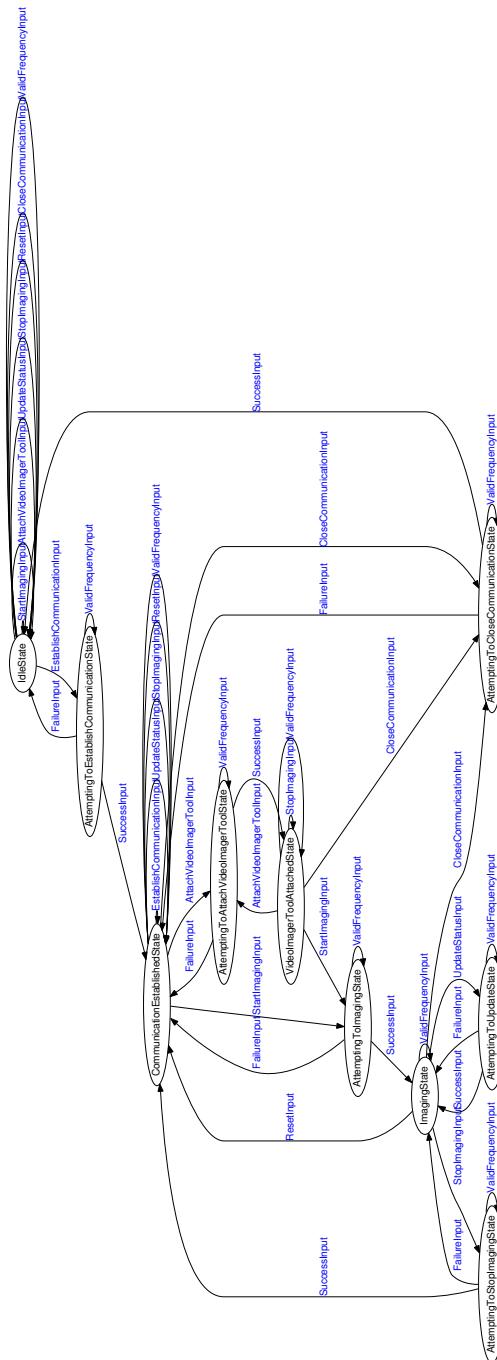


Figure 18.4: State Machine Diagram of the Videolmager Class.

6. *RequestSetFrequency()*: Set the frequency at which the frames will be acquired from the imaging device.

Events

The following events are generated by the VideoImager object.

1. *VideoImagerOpenEvent* : Generated when a call to *RequestOpen()* was successful.
2. *VideoImagerOpenErrorEvent* : Generated when a call to *RequestOpen()* failed to initiate the video device.
3. *VideoImagerCloseEvent* : Generated when a call to *RequestClose()* was successful.
4. *VideoImagerCloseErrorEvent* : Generated when a call to *RequestClose()* failed to close the video device.
5. *VideoImagerInitializeEvent* : Generated when a call to *RequestInitialize()* was successful.
6. *VideoImagerInitializeErrorEvent* : Generated when a call to *RequestInitialize()* failed.
7. *VideoImagerStartImagingEvent* : Generated when a call to *RequestStartImaging()* was successful.
8. *VideoImagerStartImagingErrorEvent* : Generated when a call to *RequestStartImaging()* failed.
9. *VideoImagerStopImagingEvent* : Generated when a call to *RequestStopImaging()* was successful.
10. *VideoImagerStopImagingErrorEvent* : Generated when a call to *RequestStopImaging()* failed.
11. *VideoImagerUpdateStatusEvent* : Generated when a call to *RequestUpdateStatus()* was successful.
12. *VideoImagerUpdateStatusErrorEvent* : Generated when a call to *RequestUpdateStatus()* failed.
13. *VideoImagerEvent* : Superclass of all events generated by the VideoImager for successful request completions.
14. *VideoImagerErrorEvent* : Superclass of all events that are generated by the VideoImager as a result of a failed request or other error.

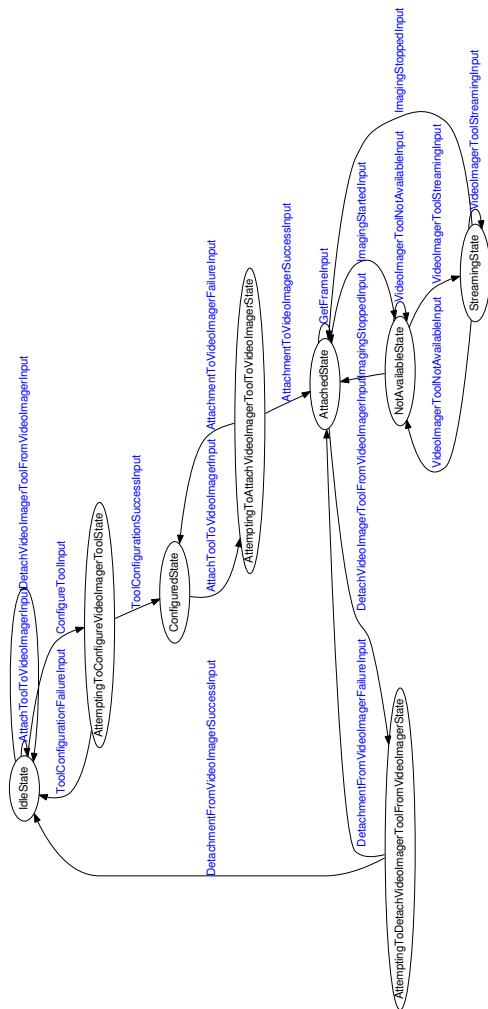


Figure 18.5: State Machine Diagram of the VideolmagerTool Class.

18.3.2 Videolmager Tool

State Machine

Figure 18.5 illustrates the state machine diagram of the `igstk::VideoImagerTool`.

The major states are as follows:

1. *Idle* : Initial state.
2. *Configured*: VideoImagerTool ready for attachment. After all the required parameters are set for the VideoImagerTool, the VideoImagerTool makes the transition to this state.
3. *Attached*: VideoImagerTool attached to the VideoImager
4. *NotAvailable*: VideoImagerTool is not available to send frames.
5. *Streaming*: VideoImagerTool available to send frames.

Interface Methods

1. *RequestConfigure()*: Request the VideoImagerTool to be configured.
2. *RequestAttachToVideoImager()*: Request the VideoImagerTool to be attached to the VideoImager.
3. *RequestDetachFromVideoImager()*: Request the VideoImagerTool to be disconnected from the VideoImager.
4. *GetVideoImagerToolIdentifier()*: Get the unique VideoImagerTool identifier.
5. *RequestGetFrame()*: Get the last stored frame from the ring buffer.
6. *GetFrameFromBuffer(const unsigned int index)*: Get the frame stored at index in ring buffer.
7. *GetTemporalCalibratedFrame()*: Get the temporal calibrated frame.
8. *SetFrameDimensions(itk::vector<unsigned int,3>)*: Set frame width, height and number of channels e.g. for PAL standard with RGB the dimensions are 720 * 576 * 3 and for PAL standard with grayscale the dimensions are 720 * 576 * 1).
9. *GetFrameDimensions(itk::vector<unsigned int,3>)*: Get frame width, height and number of channels.
10. *SetPixelDepth(unsigned int)*: Set amount of bits per color value (e.g. for grayscale images pixel depth represent the amount of bits for one gray value and for true color images it is the amount of bits for one channel).
11. *unsigned int GetPixelDepth()*: Get pixel depth.

Events

The following events are generated by the VideoImagerTool object.

1. *VideoImagerToolConfigurationEvent* : Generated when a call to *RequestConfigure()* method is successful.
2. *VideoImagerToolConfigurationErrorEvent*: Generated when a call to *RequestConfigure()* method fails.
3. *InvalidRequestToAttachVideoImagerToolErrorEvent* : Generated when a call to *RequestAttachToVideoImager()* is made when the VideoImagerTool is not yet configured.
4. *InvalidRequestToDeleteVideoImagerToolErrorEvent* : Generated when a call to *RequestDetachFromVideoImager()* is made when the VideoImagerTool is not attached to the VideoImager.
5. *VideoImagerToolAttachmentToVideoImagerEvent* : Generated when a call to *RequestAttachToVideoImager()* is successful.
6. *VideoImagerToolAttachmentToVideoImagerErrorEvent* : Generated when a call to *RequestAttachToVideoImager()* fails.
7. *VideoImagerToolMadeTransitionToStreamingStateEvent* : Generated when the tool is ready to stream video.
8. *VideoImagerToolNotAvailableEvent* : Generated when the tool is not ready to stream video.
9. *ToolImagingStartedEvent* : Generated when a call to *RequestReportImagingStarted()* method is invoked.
10. *ToolImagingStoppedEvent* : Generated when a call to *RequestReportImagingStopped()* method is invoked.
11. *VideoImagerToolDetachmentFromVideoImagerEvent* : Generated when a call to *RequestDetachFromVideoImager()* method is successful.
12. *VideoImagerToolDetachmentFromVideoImagerErrorEvent* : Generated when a call to *RequestDetachFromVideoImager()* method fails.
13. *FrameModifiedEvent* : Generated when a call to *RequestGetFrame()* method is invoked. The event contains also the current frame.
14. *VideoImagerToolEvent* : Superclass of all events generated by VideoImager tool class.
15. *VideoImagerToolErrorEvent* : Superclass of all error events generated by VideoImager-Tool class.

18.4 Example

The following is an example of using the video (VideoImager/VideoImagerTool) framework in IGSTK with a standard webcam as the video source. Note that this example requires that the OpenCV (<http://sourceforge.net/projects/opencvlibrary/>) library be installed (compiled using CMake), and the IGSTK_USE_OpenCV CMake flag set to ON.

18.4.1 Videolmager Component Example

This small example combines all important parts of the VideoImager component. Figure 18.6 and Figure 18.7 show screenshots of the example application.

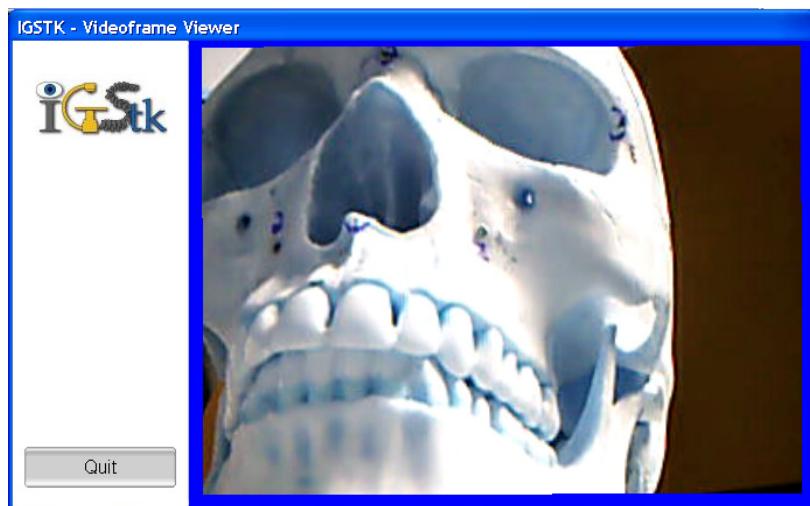


Figure 18.6: Example Viewer Screen Shot.

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

Define refresh rates for View and VideoImager.

```
#define VIEW_3D_REFRESH_RATE 25
#define VIDEOIMAGER_DEFAULT_REFRESH_RATE 25
```

The GUI is implemented using FLTK. For this purpose an instance of `igstk::FLTKWidget` and `igstk::View3D` are created and connected via the method:

```
m_VideoWidget->RequestSetView( m_VideoView );
```

This is done in `VideoFrameGrabberAndViewerWebcamWinView.cxx`. Afterwards we set up and start `igstk::View3D`.

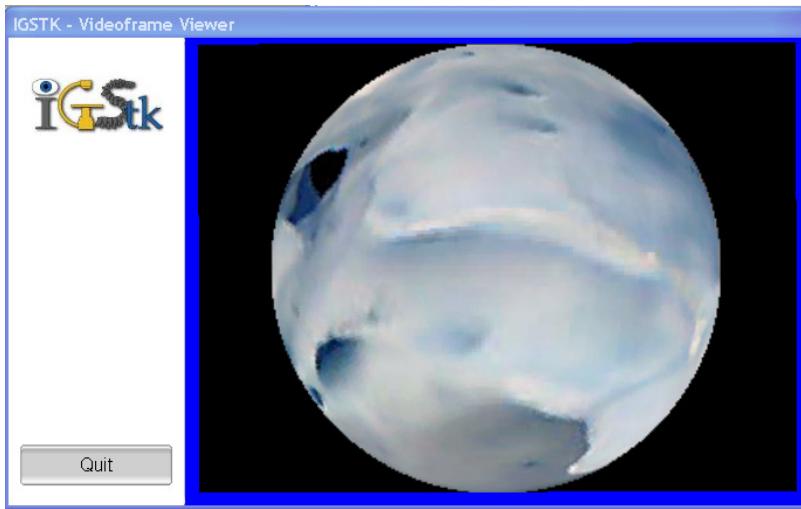


Figure 18.7: Example Viewer Screen Shot 2. View is similar to an endoscopic view.

```
// Set background color to the View
m_ViewerGroup->m_VideoView->SetBackgroundColor(0,0,1);
// Set parallel projection to the camera
m_ViewerGroup->m_VideoView->SetCameraParallelProjection(true);
// Set refresh rate
m_ViewerGroup->m_VideoView->SetRefreshRate( VIEW_3D_REFRESH_RATE );
// Start the View
m_ViewerGroup->m_VideoView->RequestStart();
// Enable user interactions with the window
m_ViewerGroup->m_VideoWidget->RequestEnableInteractions();
```

The geometrical description of the videoframe in the scene is managed by the videoframe SpatialObject. For this purpose we need `igstk::VideoFrameSpatialObject`. Since `igstk::VideoFrameSpatialObject` is a template class we define in the first parameter the pixeltype and in the second parameter the channel number. Here we set "unsigned char" for pixeltype and 3 channels for RGB. We then declare and instantiate a `igstk::VideoFrameSpatialObject` as follows:

```
typedef igstk::VideoFrameSpatialObject<unsigned char, 3 >
    VideoFrameSpatialObjectType;
VideoFrameSpatialObjectType::Pointer m_VideoFrame;
```

We set device specific parameters in `igstk::VideoFrameSpatialObject`.

```
m_VideoFrame = VideoFrameSpatialObjectType::New();
m_VideoFrame->SetWidth(320);
```

```
m_VideoFrame->SetHeight(240);
m_VideoFrame->SetPixelSizeX(1);
m_VideoFrame->SetPixelSizeY(1);
m_VideoFrame->SetNumberOfScalarComponents(3);
m_VideoFrame->Initialize();
```

The following scene graph is implemented here: igstk::View3D - igstk::VideoFrameSpatialObject - igstk::AxesObject with identity transform for each dependency.

```
m_VideoFrame->RequestSetTransformAndParent( identity, m_WorldReference );
m_ViewerGroup->m_VideoView->RequestDetachFromParent();
m_ViewerGroup->m_VideoView->RequestSetTransformAndParent( identity, m_VideoFrame );
```

The visual Representation of SpatialObjects in the visualization window is created using SpatialObject Representation classes. The corresponding Representation class for igstk::VideoFrameSpatialObject is igstk::VideoFrameRepresentation, which is also templated and needs the igstk::VideoFrameSpatialObject as parameter. We declare and instantiate igstk::VideoFrameRepresentation as follows:

```
typedef igstk::VideoFrameRepresentation<VideoFrameSpatialObjectType>
    VideoFrameRepresentationType;

VideoFrameRepresentationType::Pointer m_VideoFrameRepresentationForVideoView;

m_VideoFrameRepresentationForVideoView =
    VideoFrameRepresentationType::New();
```

Add the videoframe Spatialobject to the videoframe Representation.

```
m_VideoFrameRepresentationForVideoView->
    RequestSetVideoFrameSpatialObject( m_VideoFrame );
```

Next, the Representation is added to the View as follows:

```
m_ViewerGroup->m_VideoView->RequestAddObject(
    m_VideoFrameRepresentationForVideoView );
```

We create here an Observer for the igstk::VideoImager. This Observer will catch all failure events generated by igstk::VideoImager. The ErrorObserver is implemented in VideoFrameGrabberAndViewerWebcamWin.h.

```
this->m_ErrorObserver = ErrorObserver::New();
```

The following code instantiates a new VideoImager object for conventional webcams. The `igstk::WebcamWinVideoImager` derive from `igstk::VideoImager` and implement device specific communication. For device specific implementations see `igstk::WebcamWinVideoImager.h` and `igstk::WebcamWinVideoImager.cxx`.

```
igstk::WebcamWinVideoImager::Pointer videoImager =
    igstk::WebcamWinVideoImager::New();
```

Depending on the connected device, set the refresh rate here:

```
videoImager->RequestSetFrequency( VIDEOIMAGER_DEFAULT_REFRESH_RATE );
```

Before any request calls to the VideoImager we add an observer to the VideoImager class to catch possible error events.

```
unsigned long observerID = m_VideoImager->AddObserver( IGSTKErrorEvent(),
    this->m_ErrorObserver );
```

Now, we try to open the communication with the device and retrieve errors that might occur.

```
m_VideoImager->RequestOpen();

if( this->m_ErrorObserver->ErrorOccured() )
{
    this->m_ErrorObserver->GetErrorMessage( this->m_ErrorMessage );
    this->m_ErrorObserver->ClearError();
    m_VideoImager->RemoveObserver(observerID);
    cout << this->m_ErrorMessage << endl;
}
```

Next we create an `igstk::WebcamWinVideoImagerTool` and set frame dimensions, pixel depth, and an unique name for identification. Note these parameters must be the same as the parameters for the `igstk::VideoFrameSpatialObject`. After setup, the VideoImager tool can be configured.

```
VideoImagerTool::Pointer videoImagerTool;
WebcamWinVideoImagerTool::Pointer videoImagerToolWebcam =
    WebcamWinVideoImagerTool::New();

unsigned int dims[3];
dims[0] = 320;
dims[1] = 240;
dims[2] = 3;
videoImagerToolWebcam->SetFrameDimensions(dims);
videoImagerToolWebcam->SetPixelDepth(8);
videoImagerToolWebcam->RequestSetVideoImagerToolName("Camera");
videoImagerToolWebcam->RequestConfigure();
```

Here we connect the VideoImagerTool to the VideoImager:

```
videoImagerTool->RequestAttachToVideoImager( m_VideoImager );
```

After that the VideoImager tool can set to the videoframe Spatialobject as follows:

```
m_VideoFrame->SetVideoImagerTool(videoImagerTool);
```

Here we request the system to start the VideoImager. In case of success the communication thread starts retrieving frames continuously from the device and the main application thread fills the ringbuffer in the VideoImager tool according to the pulse generator frequency.

```
m_VideoImager->RequestStartImaging();
```

Finally, before exiting the application, the VideoImager is properly closed and other clean up procedures are executed, as follows:

```
m_ViewerGroup->m_VideoView->RequestRemoveObject(
    m_VideoFrameRepresentationForVideoView );
m_ViewerGroup->m_VideoView->RequestResetCamera();

this->m_VideoImager->RequestStopImaging();

this->m_VideoImager->RequestClose();
```

18.5 Conclusion

The video acquisition framework (VideoImager, VideoImagerTool) facilitates the integration of video sources into IGSTK applications. This is a key component in image-guided interventions, as real time imaging provides the physician with an updated view of the anatomical structures which is not reflected by the pre-operative images used for navigation. The video framework also benefits from the use of the "safety by design" philosophy of IGSTK.

Currently available device specific implementations are:

Class	BSD	LGPL	GPL	Libraries	OS
igstkWebcamWinVideoImager/Tool	X			openCV	Windows
igstkOpenIGTLinkVideoImager/Tool	X			OpenIGTLink	Windows Linux
igstkCompressedDVVideoImager/Tool		X		libraw1394 libdv libiec61883	Linux
igstkImagingSourceVideoImager/Tool			X	unicap	Linux

Table 18.1: Videolmager Implementations, Supported Operating Systems, and Software Licenses.

OpenIGTLink

This chapter introduces the OpenIGTLink protocol, an open, extensible peer-to-peer message passing mechanism for data transfer in image-guided therapy (IGT). The protocol uses TCP/IP connections to transfer various data types including rigid and linear transformations, images, and device status. The protocol emerged through a collaboration of academic, clinical, and industrial partners in developing an integrated robotic system for MRI-guided prostate interventions [7], and has been generalized by the IGT community. The benefit of using such a community developed protocol from your IGSTK applications is that it maximizes inter-operability of applications with other IGS software and devices. An open source implementation of the protocol is available from the project's web site at:

<http://www.na-mic.org/Wiki/index.php/OpenIGTLink>

19.1 Applications and Use Cases

There are a number of clinical applications that require network communication among devices and software. The followings are example use case scenarios of IGSTK accompanied with OpenIGTLink:

IGSTK as a tracking device interface: By connecting existing navigation software with IGSTK using the OpenIGTLink protocol, the users can access a wide variety of tracking devices supported by IGSTK from their navigation software. Since IGSTK and the OpenIGTLink protocol work as an abstraction layer between tracking device and navigation software, the users need to write very little code to adapt their software to particular tracking devices. Use of OpenIGTLink-based network communication also allows flexible configuration of the system, such as multicast of tracking data.

Ultrasound navigation using OpenIGTLink: The OpenIGTLink protocol can be used to transfer real-time images from an ultrasound scanner device to an IGSTK application. The merit of using the OpenIGTLink protocol is to ensure the freedom of developers to

choose the platform to run their IGSTK application, even if the image acquisition hardware or vendor-provided application program interfaces is platform-dependent. Also, the developers can separate proprietary code for image acquisition from the IGSTK application so that they can distribute the application to the public without having any license issues.

Integration of IGSTK with a commercial navigation system: IGSTK applications can be integrated with a commercial navigation system, if the vendor provides an OpenIGTLINK interface for their product. Such integration allows the users to share clinical data between proprietary software for clinical routines and research software to prototype a new surgical navigation system or scientific data analysis. In terms of patient safety, proprietary software approved by the FDA or other regulatory agencies is preferable, but this usually limits access to image and other types of data from clinical cases for research purposes. The OpenIGTLINK protocol interface allows the researchers to access clinical data from their research software, promoting clinical research. System integration using a standardized protocol like OpenIGTLINK also solves issues of license compatibility between commercial and research software as well as restriction of computer platform for the research.

Integration of IGSTK with a surgical robot: The OpenIGTLINK protocol allows the developers to integrate their IGSTK application with a surgical robot. The OpenIGTLINK protocol offers a set of message types required in communications between the navigation system and robotic device controller, such as current position and device status, and if necessary, the developers can add new message types specific to their own devices.

19.2 The OpenIGTLINK Protocol

OpenIGTLINK was designed for use in the Application Layer on the TCP/IP stack. OpenIGTLINK itself does not include mechanisms to establish and manage a session. A message, the minimum data unit of this protocol, contains all information necessary for interpretation by the receiver. The message begins with a 58-byte header section, which is common to all types of data, followed by a body section (Figure 19.1). The format of the body section varies by data type, specified in the header section. Since any compatible receiver can interpret the header section, which contains the size and data type of the body, every receiver can gracefully handle any message, even those with unknown data type. Therefore, this two-section structure allows developers to define their own data type while maintaining compatibility with other software that cannot interpret their user-defined data types. This simple message mechanism eases the development of OpenIGTLINK interfaces and improves compatibility. Details of standard data types are described in the following sections.

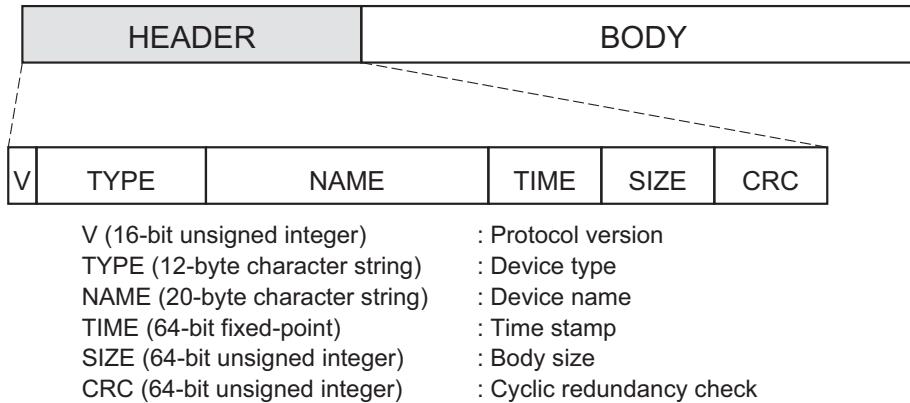


Figure 19.1: OpenIGTLink Message Structure. The message consists of general header and data body sections. The header contains information that is required for the receiver program to interpret the data body section. The structure of the data body varies depending on the data type, which is specified in the device type field in the header.

19.2.1 General Header Section

The header contains generic information about the message, including the type of the data, a field called Device Name, a time stamp, and the size of the data section. The Device Name identifies the source of the data, and is useful in sending multi-channel data through a single connection. For example, optical or electromagnetic tracking devices may track multiple objects and report the position of each object in a separate message. Thus the general header section consists of six fields:

Version number (2 bytes): The version number of the OpenIGTLink protocol used. (All data structures described in this paper were defined as Version 1.)

Data type name (12 bytes): The OpenIGTLink protocol defines five default types: IMAGE, POSITION, TRANSFORM, STATUS, and CAPABILITY, the details of which are described in the next section. In OpenIGTLink, usually the sender pushes data to the receiver, but the sender can also request the receiver to send data back. These requests are issued as messages with null-body section and with special data type names: GET_IMAGE, GET_POSITION, GET_TRANS, GET_STATUS, and GET_CAPABILITY. Developers can also define application-specific formats for their data type and associate it with a type name specified here in the message header. This practice, however, is not encouraged because it acts against portability. The following default data types are provided in the OpenIGTLink.

Device name (20 bytes): Name of the source device. This field can be used to differentiate channel of the data in multi-channel tracking.

Time stamp (8 bytes): This field informs the receiver of the time when the data was generated.
(Developers may opt to not use this field by filing it with 0.)

Body size (8 bytes): The size of the message body attached to this header.

CRC (8 bytes): The 64-bit cycle redundancy check for the body. This is used to verify the integrity of data and detect system faults. This feature may be helpful when the protocol is integrated in devices requiring Food and Drug Administration (FDA) or other regulatory approval for clinical use.

19.2.2 Data Body Section

The body structure varies by the data type being sent in the message, explained as follows.

POSITION / GET_POSITION The POSITION data type is used to transfer position and orientation information. The data are a combination of 3-dimensional vector for the position and quaternion for the orientation. The quaternion used in the OpenIGTLLink protocol, if normalized to unit magnitude, is equivalent to the versor introduced in previous chapters. Although equivalent position and orientation can be described with the TRANSFORM data type, the POSITION data type has the advantage of smaller data size, where the size of POSITION message is 19 % less than that of TRANSFORM message. It is therefore more suitable for pushing high frame-rate data of tracking devices.

TRANSFORM / GET_TRANSFORM The TRANSFORM data type is used to transfer a homogeneous linear transformation in 4-by-4 matrix form defined by:

$$M = \begin{pmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} t & s & n & p \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (19.1)$$

where the upper-left 3-by-3 matrix represents rotation, and the upper-right 3-by-1 column vector represents transformation (center position of the image). The matrix is equivalent to the conjunction of normal column vectors for directions of voxel arrays in i, j, and k indices, which are denoted as \vec{t} , \vec{s} , and \vec{n} in Equation 19.1. TRANSFORM data type in the OpenIGTLLink protocol contains only the top 3 rows of the 4-by-4 matrix. Note, that if a device is sending only translation and rotation, then TRANSFORM is equivalent to POSITION. But TRANSFORM can also be used to transfer affine transformations or simple scaling.

IMAGE / GET_IMAGE The IMAGE format in OpenIGTLLink supports 2D or 3D images with metric information including image matrix size, voxel size, coordinate system type, position, and orientation. Data type can be either scalar or vector, and numerical values can be 8-, 16-, 32-bit integer, or 32- or 64-bit floating point. The format also supports partial image transfer, in which a region of the image is transferred instead of the whole image. This mechanism is suitable for real-time applications, in which images are updated region-by-region. The sub-volume must be box shaped and defined by 6 parameters consisting of the indices for the corner voxel of the sub-volume and matrix size of the sub-volume. The indices begin at 0, similar to

the conventional zero-based array indices in the C/C++ programming languages. The position, orientation, and pixel size are represented in the top 3 rows of a 4-by-4 matrix, which has been shown earlier in equation (1). The position and orientation vectors can be described in either a left-posterior-superior (LPS) or right-anterior-superior (RAS) coordinate system, depending on the coordinate system specified also in this data format. The LPS is a right handed coordinate system used in the DICOM standard, while the RAS is used in open-source surgical navigation software such as the 3D Slicer [4].

STATUS / GET_STATUS The STATUS data type is used to notify the receiver about the current status of the sender. The data consist of status code in a 16-bit unsigned integer, sub code in a 64-bit integer, error name in a 20-byte-length character string, and a status message. The length of the status message is determined by the size information in the general header. The status code is defined as a part of the OpenIGTLink specification listed in the Table 1. The sub code is device specific and is defined by developers. In addition, developers can build their own error name/code into the status message and additional optional description in the following data field. GET_STATUS is null data to request status information from the receiver. The receiver returns a STATUS message as a response.

CAPABILITY / GET_CAPABILITY The CAPABILITY data type lists the names of message types that the receiver can interpret. Although the OpenIGTLink protocol guarantees that any receiver can at least skip messages with unknown type and continue to interpret the following messages, it is a good idea to get the capability information at system startup to ensure application-level compatibility of various devices. In a CAPABILITY message type, each message type name comes with format version number. If the receiver can interpret multiple versions for a certain message type, they should be listed as independent types. GET_CAPABILITY is a null data request for the list of the names of message types that the receiver can interpret.

User-defined data types OpenIGTLink allows developers to define their own message types. As long as the general header has correct information about the size of the data body, it retains compatibility with any software compliant with OpenIGTLink, because the receiver can skip data that it cannot interpret. The OpenIGTLink specification recommends that the developer add an * (asterisk) at the beginning of the type name in the general header, to avoid future conflicts with standard types and also to make it more obvious to distinguish between standard and user-defined message types.

19.3 The OpenIGTLink Library

The OpenIGTLink community is providing the OpenIGTLink Library as a reference implementation of the OpenIGTLink interface. The library is a free open-source software (FOSS) distributed under a BSD-style open-source license placing no restrictions on use. The library consists of three components, as seen in Figure 19.2.

- A C-based library defining structures and utility functions to serialize data into OpenIGTLink Message. This library is useful in developing embedded systems or software for platforms where a modern C++ compiler is not available.

- A set of high-level C++ classes wrapping the C-based library that provide a safer and more convenient way to implement OpenIGTLLink messaging function into software. Developers can define their own message types by inheriting the base message class defined in the library.
- A set of multi-platform C++ classes to handle sockets and threads. The multi-platform socket and thread classes are currently compatible with 32-bit Windows, Linux, Mac OS X, and Solaris platforms.

In the following section we show how to use the OpenIGTLLink library to export tracking data obtained by an IGSTK application. This is useful for third party programs that do not want to incorporate IGSTK code directly into their code base (e.g. 3D Slicer).

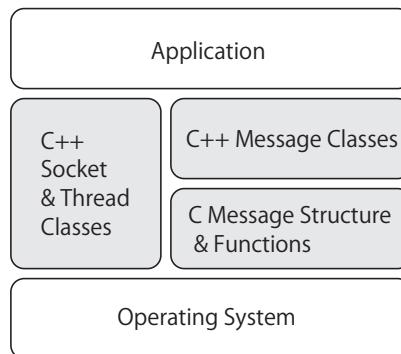


Figure 19.2: The OpenIGTLLink Library Architecture. In the lowest level, the messages are defined as C structures with several supporting functions for message serialization. On top of the C structures and function, C++ message classes are built to provide easier, safer, and more extensible way to access to OpenIGTLLink messages. Besides message serialization support, multi-platform C++ socket and thread classes are provided to support writing platform-independent application codes.

19.4 Example: Exporting Tracking Data using the OpenIGTLLink

This example enables the user to broadcast tracking data using IGSTK. All the user needs to do is compile the program(s) and create an xml file that is appropriate for the desired setup. By default a command line program is generated (OIGTLLinkTrackerBroadcasting). If you have set the Cmakelists flag USE_FFLTK to ON a second program with a UI is also generated (OIGTLLink-TrackerBroadcastingUI). Both programs receive as input an xml file that contains the relevant setup: communication settings, which tools to track, whether one of the tools is a dynamic reference, and tool calibration information. Note that for the command line version you will need to run the program and specify the xml file as an argument. The GUI version receives the xml file via a text field (part of the UI).

The programs support all tracking systems supported by IGSTK with minimal effort on the user side. That is, the user need only edit an xml file describing the current configuration.

What data is sent:

1. If a tool is designated as a dynamic reference frame its transformations will not be sent.
2. If a calibration file is specified for the tool this transformation is compounded with that reported by the tracker. For example, when using a tracked pointer with calibration information denoting the transformation to the tool's tip, the tip's location and orientation are the reported transformation.
3. If one of the tools is designated as a dynamic reference frame then all reported transformations are relative to that tool. If the reference frame or the tool cannot be tracked no data is sent (wildly waving the tool in front of the Vicra system will not help if the reference is in the desk drawer).

An example configuration using Northern Digital Inc.'s Vicra tracker:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<tracking_system type = "polaris vicra">
    <!--acquire data from the tracker at this rate-->
    <refresh_rate> 20 </refresh_rate>
    <!--serial communication settings-->
    <serial_communication>
        <com_port>5</com_port>
        <baud_rate>115200</baud_rate>
        <data_bits>8</data_bits>
        <parity>N</parity>
        <stop_bits>1</stop_bits>
        <hand_shake>0</hand_shake>
    </serial_communication>

    <!--use this tool as a dynamic reference frame, the tool's-->
    <!--transformations will not be sent -->
    <tool usage="reference">
        <name> reference frame </name>
        <srom_file>E:/users/xmlFiles/drft.rom</srom_file>
    </tool>

    <!--send this tools transformations to the given addresses-->
    <!--this will be the probe's tip location+orientation -->
    <!--relative to the reference tool -->
    <tool>
        <name> bayonet probe </name>
        <srom_file>E:/users/xmlFiles/bayonetProbe.rom</srom_file>
```

```

<calibration_file>E:/users/xmlFiles/bayonetCalibration.xml</calibration_file>
<send_to> 127.0.0.1:1001 </send_to>
<send_to> 127.0.0.1:1002 </send_to>
</tool>

</tracking_system>
```

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

Examples/OIGTLLinkTrackerBroadcasting/OpenIGTLLinkTrackingBroadcaster.h.

This example illustrates how to export tracking data through OpenIGTLLink connection. The example program supports multi-cast data transfer, and is the client part of the OpenIGTLLink client-server approach.

To use the OpenIGTLLink POSITION message, and client socket we need the following includes:

```
#include "igt1OSUtil.h"
#include "igt1PositionMessage.h"
#include "igt1ClientSocket.h"
```

ToolUpdatedObserver class observes the TrackerToolTransformUpdateEvent for a specific tool. It checks that the event is for the relevant tool and then gets the tool's transform to its parent and sends it via socket. In the ToolUpdatedObserver constructor we instantiate a igt1::PositionMessage, which is updated with the transformation data.

```
this->m_PositionMessage = igt1::PositionMessage::New();
```

In *ToolUpdatedObserver::Initialize()*, we set the device name of the message by invoking the *SetDeviceName()* method.

```
this->m_PositionMessage->SetDeviceName( toolName.c_str() );
```

We create a list of destinations to support multicast data transfer. Then we establish connections for each destination on the list.

```

std::vector< std::pair<std::string, unsigned int> >::iterator
destinationIt;
for ( destinationIt = destinations.begin();
      destinationIt != destinations.end(); ++destinationIt)
{
    igt1::ClientSocket::Pointer socket = igt1::ClientSocket::New();
    int r = socket->ConnectToServer( destinationIt->first.c_str() ,
                                    (int)destinationIt->second );
    if (r != 0)
    {
```

```
for (it = this->m_Sockets.begin(); it != this->m_Sockets.end(); ++it)
{
    (*it)->CloseSocket();
    (*it)->Delete();
}
this->m_Sockets.clear();
std::ostringstream msg;
msg<<"Failed to connect to " << destinationIt->first
     << " port " << destinationIt->second;
throw ExceptionWithMessage( msg.str() );
}
this->m_Sockets.push_back(socket);
}
```

In *ToolUpdatedObserver::Execute()*, we define the event handler to receive a transform, fill the OpenIGTLink message, and send it out.

```
igstk::Transform transform =
    this->m_TransformObserver->GetTransform();
igstk::Transform::VectorType t = transform.GetTranslation();
igstk::Transform::VersorType r = transform.GetRotation();
this->m_PositionMessage->SetPosition(t[0], t[1], t[2]);
this->m_PositionMessage->SetQuaternion(r.GetX(), r.GetY(),
                                         r.GetZ(), r.GetW());
this->m_PositionMessage->Pack();

std::vector< igtl::ClientSocket::Pointer >::iterator it;
for (it = this->m_Sockets.begin(); it != this->m_Sockets.end();
     ++it)
{
    (*it)->Send(this->m_PositionMessage->GetPackPointer(),
                  this->m_PositionMessage->GetPackSize());
}
```


State Machine Validation

“Quality has to be caused, not controlled.”

—Philip Crosby

State machine validation to IGSTK means:

- State machines are structurally sound; they adhere to rules governing how states, transitions, and inputs may be combined.
- State machines exhibit predictable behavior. On a given input from a valid state, an expected transition is taken to a new (or possibly the same) state.
- A state machine is always in a valid state after each operation.
- Specific states and transitions in state machines should be visited based on the visible behaviors of an IGSTK application.
- Assisting programmers in the tasks of designing, constructing, and understanding state machine governed interactions.

The goal of state machine validation is to create tools that validate the safe design, implementation, and execution of state machines in IGSTK components and applications¹. This chapter presents a collection of tools that together provide a way for developers to validate state machine design and implementation within the context of IGSTK’s agile process.

20.1 Motivation

IGSTK employs an agile methodology, which favors continuous builds and testing over extensive design activities. For example, consider the classic approach when basing an architecture

¹We acknowledge this is more a form of architecture *implementation* validation than architecture *design* validation, which is the typical interpretation of the phrase architecture validation in research circles.

on state machines. Typically, one would create a model of the state machine in a tool or formal language. Then the model would be verified through analytical (provable) means or extensive simulation. The model would then be used to generate code through a correctness-preserving automated or manual process.

IGSTK does not follow this approach. IGSTK provides its own internal state machine execution semantics. State machines are coded by hand, without the aid of visual modeling tools, formal languages, or OTS (off-the-shelf) runtime support. The ability to completely control execution semantics, and not introduce a 3rd party dependence is viewed as a safety benefit to IGSTK. It means IGSTK is not tied to OTS technologies, and does not have to map IGSTK execution semantics onto tools whose semantics generally were intended for embedded systems with asynchronous and multithreaded runtime semantics.

Agile methods and a lack of a formal specification of the architecture are an unusual combination for *quasi* real-time safety-critical software, although it is not unprecedented within the medical device community [24]. A particular concern is that although IGSTK’s continuous testing process emphasizes black box (functional) testing at the component level, it does not validate state machine structure (static analysis) or expected runtime properties (dynamic analysis). An IGSTK component that exhibits proper functional behavior with respect to unit tests may have an undiscovered defect in the construction of its state machine. Further, there is currently no means by which the global state of the system, taken as the composition of states of its instantiated components, is validated for consistency over time.

To address these issues, we created tools to perform architecture validation post-construction. We extract a model of component state machines from the source code, and then validate this model with respect to architecture characteristics relevant to IGSTK. The current areas of emphasis are on static (structural) analysis and global state consistency through dynamic analysis (runtime constraint checks). These techniques are being integrated into IGSTK’s continuous build and testing process.

20.2 Background

There is a significant body of theoretical work in the area of concurrent systems and state machines. Model checkers are tools that have been applied to validation problems in these areas. Model checkers provide complex functionality for validating state machine models, but their use is not suited for IGSTK because:

- They consider validation criteria that are important for real-time concurrent applications, but not important for IGSTK. IGSTK applies state machines for *safety-by-design* not for managing complexity of a concurrent application.
- Missing or poorly constructed documentation and the need to learn specialized languages lead to a large learning curve.
- They are often too powerful, having more features and complexity than IGSTK requires.

- They are often not distributed under an open source license.
- They are often general-purpose tools, whereas some specificity is necessary in the surgical domain with regards to safety.

Tools considered included Hugo/RT [16], UPPAAL [1], and SPIN [12]. Hugo/RT is a model translator, which takes an XMI² as input and generates the necessary files to be given as input to UPPAAL or SPIN. UPPAAL and SPIN are model checkers which are used to verify if a particular state can be reached or if it is possible at any instance of time for the system to go into an inconsistent state. Initially we tried to incorporate Hugo/RT as its acceptance of XMI gave us hope that we could use the same exported information in UML design tools. However, incompatibilities in XMI version support and the specific representation of statecharts made these attempts unsuccessful.

We then successfully implemented first-generation validation tools that were integrated with Macgee and Kramer's Labeled System Transition Analyzer (LTSA [20]). This tool's license is not compatible with IGSTK, and the underlying representation of state machines, while simple, is not extensible nor based on any standard.

Fundamentally, IGSTK's agile approach is at odds with model-driven specification and validation. These tools do not enable one to move quickly and build a dependence on a third-party notation or tool to provide the safety IGSTK applications require. Instead of forcing the community to change its process, we looked for ways to leverage current understanding of validation techniques and integrate them into the agile process. The result is a post-implementation validation suite of tools that assists with both rapid design and continuous verification activities in an agile process.

20.3 Approach

The IGSTK Validation Toolset is a suite of tools that provide the following collective features:

1. *State Machine Export* - IGSTK state machines are hand-coded; the only way to work with them external to the code is either to reverse engineer or explicitly export the state machines. While reverse engineering would not be altogether too difficult, state machine export has been supported from the earliest versions of the toolkit.
2. *Global State Validation* - IGSTK state machines are encapsulated to individual components (classes). Global state validation allows a developer to write rules for test cases that check that multiple component instances cannot be in inconsistent states at any given point in time.
3. *Simulation* - The primary tool allows one to "run" a sequence of events through a state machine instance. The nature of the event stream and the ability to instrument the simulation process provide significant value to validation.

²OMG's XML Model Interchange, based on the UML Meta-Object Facility (MOF). Specification available at <http://www.omg.org/technology/documents/formal/xmi.htm>.

4. *Visualization* - As mentioned above, no design tool exists in which to model IGSTK state machines and then generate code. The validation toolset includes a visualization tool that takes exported static representation of state machines and renders them in a Java application. Further, the tool also interfaces to the simulation subsystem to show a state machine animation.
5. *Continuous Testing Integration* - To make the validation toolset useful within the agile context, the tests that can be applied through simulation may have their results reported to a CDash dashboard.

Each of these features are discussed in more detail in the remainder of this section.

20.3.1 State Machine Export

IGSTK's agile approach to unit testing is a classic black-box approach - instrument the code to execute specific methods and/or method sequences and evaluate the observable results against known answers. White-box testing is not typically performed, though one can output the values of internal variables at any given time and use this as a basis for a unit test.

This approach does not tell the developer if a state machine is in fact operating according to its conceptual design. Keep in mind, there is not a tangible *design* here, simply a conceptual design in the developer's head or scribbled out on a piece of paper (or perhaps the Wiki). What needs to be validated is 1) does this design make sense, 2) does it conform to IGSTK guidelines for state machine development, and 3) does the implementation result in state machine behavior that matches the developer's expectations? Again, since the state machine design is only manifested in source code, there is no model or picture that one may inspect to perform this validation.

The first step then, in validating a state machine is to export it to some other format and take a look. This is a form of visualization that has been supported in IGSTK since the beginning of the toolkit via various export methods on the state machine class. At first, IGSTK developers exported state machines to .dot format (GraphViz) and posted the images on the developer Wiki for community review. Later an export method to LTSA was added.

Static image exports are useful for seeing the abstraction, but obviously one cannot interact with it. Not so obvious however, is that one cannot really analyze the state machine as well. An IGSTK state machine is a straight deterministic finite automata (DFA) which does not incorporate abstractions in Harel-like [11] (UML) statecharts. That is, there are no guard conditions on transitions, no hierarchies, and no pseudostates (other than the initial state). Therefore analysis of the input sequences accepted by an IGSTK state machine is straightforward, except there is no formal representation of them in the first place!

To facilitate a more robust analysis, a method was introduced that exports IGSTK state machines from code to the W3C StateChart XML (SCXML) format. This format was chosen for several reasons. First, previous attempts to use XMI were too cumbersome. Second, SCXML is XML-based, making it human-readable and easily manipulable. Third, SCXML is straightforward and IGSTK state machines were readily mapped to it (compared to XMI, where the UML

MOF³ layer made it too cumbersome). Fourth, an open source project from the Apache Foundation, Apache Commons SCXML, provided a rich library for creating and executing SCXML-represented state machines. Finally, we were surprised to find that few standard representations for state machines existed.

20.3.2 Global State Validation

The concept of having each component in a safe state may be extended to global system state by validating that the state of two distinct components are consistent with each other. For example, an assertion such as the IGSTK Spatial Object Representation for a scalpel may not be rendered while the Tracker device is malfunctioning is tied to two component states, not one. Global validation is the ability to check that two or more IGSTK components are in compatible states. Consider how a request-response sequence is done in IGSTK, as shown in Figure 20.1 below.

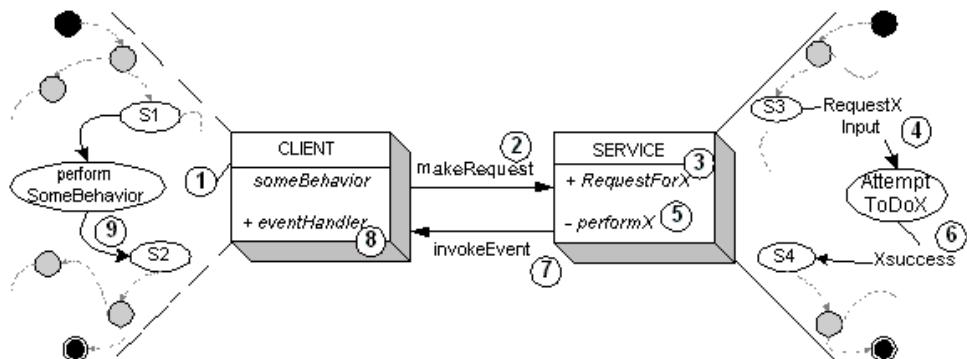


Figure 20.1: IGSTK Component Request-Event Exchange.

In the figure, Client is performing some behavior while in a well-known state (1). Client makes a requests (2) via a public method invocation on Service. Service accepts the request (3) and translates it to an input to its state machine (4). Based on the transition, a private method invocation on component Service also can occur (5). Upon successful completion of the computation (6), Service generates an event (7) and dispatches the event to Client (8). Client translates the event into an input to its state machine (9), thereby avoiding error-prone conditional logic on return values.

Cascading request and event sequences between two or more components carries an implicit expectation that one component is in a given state while a second dependent component is in a compatible state. This is not explicitly coded anywhere as the state machines are encapsulated within a given component, but if it were not true then call sequences within IGSTK would fail. It is perfectly acceptable (and safe) for this type of implicit dependency at the coding level, but a developer should be able to validate compatible states between components.

³UML's Meta-Object Facility.

20.3.3 State Machine Simulation

In a simulation-based approach to validation, input sequences are generated that represent *tests*, while collections of related tests that achieve some validation goal are called *test cases*. The number and nature of test cases provides the power behind the simulation approach. For example, a set of input sequences may be constructed that represent situations where an IGSTK component fails in mid-operation. As an another example, input sequences may be derived from a previous live execution of an IGSTK application (this has been achieved and is known as *replay*, which is described below).

Simulation is the desired approach because there is not a feasible way to unit test individual state machines for IGSTK components. IGSTK components push inputs and send events to other components (which are transduced to state machine inputs, see Chapter 7) within their implementations. Extracting state machine representations and executing them in a simulator allows the developer (or tester) to focus exclusively on the design of the state machine.

The ability to send arbitrary input sequences to the state machine means one can define a test case for any validation goal. For example, a developer might want to check that state "Initialized" is visited before state "Executing" in any input sequence, or that an error-correcting subprocess is entered and exited before resuming normal operation of a component. Input sequences, represented as XML *send-event* files, may be manually or automatically constructed to check for these cases. This capability, combined with the validation rules described above, provide the majority of the validation capability of the toolset.

20.3.4 Visualization and Animation

State machines are an abstraction that are familiar to computer scientists, particularly in a visual form through *state transition diagrams* or *UML (Harel-style [11]) statecharts*. However, as an agile-oriented project, IGSTK foregoes traditional design processes in favor of a rapid feature development and continuous test approach. Therefore, the only true "design" tool in IGSTK is the developer Wiki, and the only upfront modeling or visualization of abstract concepts occurs on the Wiki.

This is unusual in a safety-critical framework, and in one that uses state machines specifically. The prevailing approach in the community is to use a formal methods tool that typically includes a visualization component - for example LTSA as mentioned above or commercial tools such as Telelogic's Rhapsody or IBM/Rational's RoseRT. IGSTK is no different, and can benefit from "seeing" these core state machine abstractions.

Further, it is useful to see if the state machine behaves in an expected manner when faced with varying input combinations. While one can formally analyze a state machine for various properties (reachability of a state, acceptable input sequences, deadlocks, etc.), it is useful for a developer to "see" if the state machine behaves at runtime in the way expected. That is, it is not enough to see the structure of the state machine, it is also necessary to see it *live*, processing input sequences and following transitions. Only in this way can the developer know if the state machine is governing the state machine in a safe yet productive manner.

The validation toolset provides a way to visualize static state machine structures, and view an animation of that structure by processing a sequence of inputs and showing the transitions taken.

20.3.5 Continuous Testing Integration

IGSTK's development community is growing, drawing interest from around the world. Like many distributed efforts on open source projects, the IGSTK team relies on agile methods and tools to develop the architecture. IGSTK incorporates a number of lightweight best practices, including constant communication, sandboxing, and unit testing. Some concessions are made to more invasive practices due to the domain, including code reviews, continuous requirements reviews, and requirements management for traceability. To implement its process, IGSTK relies heavily on automating practices using tools. IGSTK uses a cross-platform make tool (CMake), a nightly build process (CDash), a web-enabled defect tracker (PHPBT), CVS scripts, and a documentation generator (Doxygen). IGSTK uses tools for everything except design.

IGSTKs non-model driven architecture approach and reliance on downstream tools (particularly the CMake/CDash enabled nightly build process) creates unique constraints on validation efforts. Reorienting toward a full architecture specification (with review), model verification, and code generation with underlying runtime support is not an option. The validation toolset must be able to post-validate state machine structure and execution and report its results as part of the agile process supported by these tools. Our challenge is to create a toolset that not only addresses the unique approach to state machine design and implementation in IGSTK, but to integrate these validation tools into an agile development process that relies heavily on automated tool support.

20.4 Installing and Running the Validation Toolset

20.4.1 Getting and Building the Validation Software

The IGSTK Validation Tools are written mostly in Java to more easily provide cross-platform support and use existing facilities like the Apache Commons SCXML library. The validation toolset can be compiled using either CMake or Apache Ant, and Ant targets also exist to execute various tool components.

To obtain and build the IGSTK Validation Toolset, retrieve it from the IGSTK Contribution project:

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

When prompted for password, reply "igstk". After login, you can check out the Contribution module using this command:

```
$ cvs checkout Contribution
```

Compile the project using this command:

```
$ ant compile
```

The tools are covered by the same open source license as IGSTK proper.

20.4.2 Running the Validation Software

The current validation tools platform is a set of programs and scripts that provide a platform for architecture validation for IGSTK. These tools include:

1. Methods on the IGSTK state machine class to export state machine structures to SCXML and LTSA formats.
2. Stand-alone programs and integrated rules to perform *structural* (static) analysis.
3. A set of Java programs to produce send-events files that realize state machine (node, edge, and heuristic) coverage test cases.
4. A program to parse IGSTK logfiles for state machine-related log messages and translate them to send-events files.
5. A state machine simulation engine based on the Apache Commons SCXML library. This engine takes an SCXML representation of an IGSTK state machine and a send-events file and performs a mock execution of the state machine based on the events.
6. Drools rule files that declaratively specify staticm dynamic, and ad hoc custom constraints on state machine execution. These rules represent the conditions of individual tests that may be checked for during simulation.
7. A stand-alone program named *SMVIZ* that displays IGSTK state machines, animates state machines by processing send-events files, and allows an end user to interact with state machines by sending one event at a time to an instance.
8. Ant scripts to build and run Java code.
9. CMake files that can build the Java code and submit simulation test results to a specified CDash dashboard.

Before executing the tools, you should make sure you have SCXML files corresponding to each of the IGSTK component types for your local build of IGSTK. The IGSTK validation tools CVS module does include an `xmlFiles` subdirectory that itself has four subdirectories. `scxmlFiles` has a dump of IGSTK 4.0 SCXML files, `logFiles` has example IGSTK application logfiles, `processLogFile` has example events files generated from running the `processLogFile` tool on the logfiles, and `sendEventFiles` has various example events files for use in the simulator. All of these files are included with the CVS module for experimentation purposes. Applying

the validation tools means running them against data files generated by your local build. After generating your simulation data sets, edit the properties/vtconfig.xml configuration file to setup what state machine(s) to execute via the simulator and define what event sequence to send to each state machine. Then set the `IGSTK_VT_HOME` to the root of the validation tools installation.

The easiest way to execute the various validation tools is to use ant targets already setup to run them. Targets prefixed with `run` execute a tool. Run targets include `run-smviz`, `run-simulator`, `run-processLogFile`, `run-coverage`, and `run-junit`.

20.4.3 Using the Tools

The flow diagram in Figure 20.2 below depicts how to use these tools together.

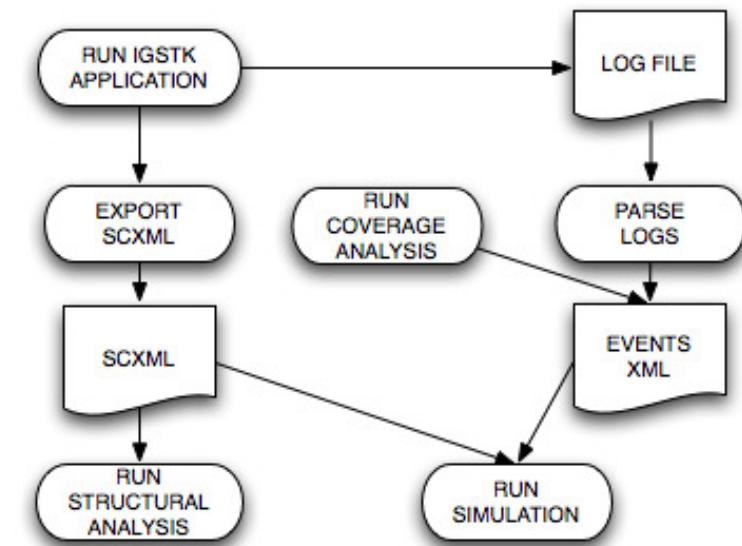


Figure 20.2: IGSTK Validation Tools Flow.

In this figure, ovals represent actions a developer or tester takes, while the document icons are data output by those actions. The validation process starts, appropriately enough, by running an IGSTK application. The application can produce two types of data: SCXML documents or log files. SCXML documents are producable by having the application invoke the `StateMachine::ExportDescriptionToSCXML` method on each state machine instance it wishes to create an SCXML file for. The IGSTK unit test driver includes an export unit test that will create all SCXML documents for all IGSTK component types. IGSTK log files are produced by the logger, described in Chapter 13. The `StateMachine` class is instrumented to log a message each time the machine fires a transition.

With the SCXML document one can run stand-alone structural analysis tools as shown on the left side of the figure. Together with an events file (the `send-events` file), one can run a sim-

ulation. Events files can be generated by parsing the logfiles, identifying log messages for transitions, and capturing these as events. In this way *replay* can occur. Event files may also be generated by a *generator*. An example generator shown in Figure 20.2 uses the coverage tools (*Run Coverage Analysis*), to generate one or more events file.

Once the SCXML and corresponding events files exist, they can be fed into the simulator. Once in the simulator, they can be used as the foundation for a test case, a visualization/animation (SMVIZ), replay, or user experimentation (interact with the state machine on a step-by-step basis) - or all of the above at the same time. Test case output is submitted to a CDash dashboard in the usual CTest way.

The following subsections further explain each of the steps in this process.

Exporting SCXML

Statechart XML (SCXML) is a draft standard⁴ of the W3C. This easy-to-understand XML format is exported by the `StateMachine::ExportDescriptionToSCXML` method, and the State Machine Export unit test writes the SCXML representations of each IGSTK component class' state machines to the `IGSTKHOME/Testing/Temporary/StateMachineDiagrams` directory under an IGSTK build. You may use this test driver or write a custom program to export state machines.

The SCXML representation for the IgstkDICOMImageSurrogateReader shown in Chapter 6:

```
<?xml version="1.0" encoding="us-ascii"?>
<scxml version="1.0" initialstate="IdleState">
  <state id="IdleState">
    <transition event="imageDirectoryNameValidInput">
      <target next="ImageDirectoryNameReadState"/>
    </transition>
  </state>
  <state id="ImageDirectoryNameReadState">
    <transition event="imageSeriesFileNamesGeneratingSuccessInput">
      <target next="ImageSeriesFileNamesGeneratedState"/>
    </transition>
    <transition event="resetReaderInput">
      <target next="IdleState"/>
    </transition>
    <transition event="imageSeriesFileNamesGeneratingErrorInput">
      <target next="IdleState"/>
    </transition>
  </state>
  <state id="AttemptingToReadImageState">
    <transition event="imageReadingSuccessInput">
```

⁴The latest working version of the standard, dated May 2008, is available from <http://www.w3.org/TR/scxml/>.

```
<target next="ImageReadState"/>
</transition>
<transition event="resetReaderInput">
    <target next="IdleState"/>
</transition>
<transition event="imageReadingErrorInput">
    <target next="IdleState"/>
</transition>
</state>
<state id="ImageSeriesFileNamesGeneratedState">
    <transition event="readImageInput">
        <target next="AttemptingToReadImageState"/>
    </transition>
</state>
<state id="ImageReadState">
    <transition event="resetReaderInput">
        <target next="IdleState"/>
    </transition>
</state>
</scxml>
```

Structural Analysis

The structural validation toolset checks for structural anomalies in all IGSTK state machines. Structural anomalies we addressed are 1) checking that IGSTK state machines are fully connected, 2) verifying that all IGSTK state machines are deterministic, and 3) ensure all inputs defined for a given state machine are in fact used by the state machine.

The first tool is particularly useful in IGSTK. An IGSTK design best practice is to ensure a state machine defined a transition for each and every possible input to the machine. This is part of the safety-by-design philosophy – if a developer does not define such a transition (or a null transition), then it is possible that the developer never considered that situation, which is considered unsafe. A code review audit in 2008 found that many state machines were not following this best practice. The structural analysis tool now finds these instances as part of the automated testing process and posts the results to the dashboard.

Coverage Tools

Another part of IGSTK’s safety-by-design philosophy was to remove as many “if/then” constructs within method bodies, particularly those that are based on the return value of some other method invocation. Instead, conditional logic is programmed into the state machine architecture. But this introduces the problem of how to verify the programmed flows in the state machines. Traditional testing approaches use the concept of *code coverage* to construct test cases that ensure the code is sufficiently exercised to detect defects. But, since the conditional

logic is not embedded in the state machine, the concept of coverage also must be ported to the state machine.

Our approach was to implement a test-case generator based on *state*, *transition*, and *path* coverage. These concepts are analogous to traditional node, edge, and path code coverage algorithms. *State coverage* is a technique used to check whether each state in the state machine can be reached from the initial state or not. This is a relatively weak criterion in IGSTK as states are not as interesting as the set of allowable transitions from a state. *Transition coverage* requires generating test cases where every transition in the state machine is considered at least once. The validation tools include a flag for testing all transitions versus testing just those that transition to another state (which is usually all that is required). Transition coverage is more interesting as each transition to another state represents a method invocation allowed by the current state of the component.

The most interesting case is *path coverage*, which attempts to construct input sequences representing all possible paths through the machine. Like traditional path coverage testing, this algorithm is computationally intractable, and so we pursued a heuristic path generation process based on classic structured testing methods⁵.

Basis path testing is a form of white box testing. Watson and McCabe [26] describes structured testing as a methodology used to determine path coverage criteria of control flow graphs. The number of basis paths is the number of linearly independent paths through a strongly connected graph and is equal to the cyclomatic complexity. According to the structured testing if a basis set of paths through a module are tested then additional paths can be realized as linear combinations of the paths that have already been tested. The validation tool uses this structured testing methodology to identify a basis set of paths through each IGSTK state machine, ensuring that all basis paths are executed at least once.

Additionally, the path coverage algorithm may take into account *domain-independent* or *domain-dependent* heuristics. These heuristics are used to direct the algorithm to generate input sequences that satisfy one or more constraints. Domain-independent heuristics are general heuristics that are applicable to all the state machines. Common domain-independent heuristics include *do not traverse a loop more than two times*, or *avoid states with ERROR as the suffix of the state label*. The purpose of the first heuristic would be to limit the amount of computation with the expectation that no cumulative behavior traversing a loop is of interest to the test case. The second heuristic might be used for a test case that only expects to verify normal operation (happy day) cases. In both cases, the nature of the heuristic rule does not depend on the component that uses this state machine.

Domain-dependent heuristics are more interesting in that the developer/tester may configure the path generating algorithm to create input sequences specific to a type of component. Example domain-dependent heuristics include *for the Tracker component, updateStatusInput will be iterated 20 times by default because while updating the value is going to be different each time*, and *for the LandmarkRegistration component, TrackerLandmarkInput will be considered at least twice to consider situations where more than the minimum of three landmarks are used*.

⁵Structured is used here in the classic sense [26] and is not to be confused with the above discussion on structural testing of state machines.

Heuristic definitions are placed in an XML file in the External/heuristics directory under the validation tools root. Several examples are included with the validation tools download.

Parsing Logfiles

The validation toolset includes a utility to parse IGSTK application logfiles and produce a *send-events* file. This utility is written in Java and executable via the main Ant script's run-processLogFile target. This program extracts logfile lines generated by the DEBUG level of the state machine class logger. For example, here is a logfile snippet showing DEBUG messages for a View3D instance:

```
24514344244.69265:(DEBUG) igstkView::RequestStart() called...
24514344244.69569:(DEBUG) State transition is being made : View3D
    PointerID 02D10200 InteractorInitializedState(93) with
        StartRefreshingInput(85) ---> RefreshingState(94).
24514344244.70348:(DEBUG) igstkView::StartProcessing() called...
24514344244.70676:(DEBUG) igstkView::RequestResetCamera() called...
24514344244.70999:(DEBUG) State transition is being made : View3D
    PointerID 02D10200 RefreshingState(94) with
        ResetCameraInput(84) ---> RefreshingState(94).
24514344244.71764:(DEBUG) igstkView::ResetCameraProcessing() called...
24514344244.77372:(DEBUG) igstkView::RequestSetRenderWindowSize(...)
    called...
24514344244.77785:(DEBUG) State transition is being made : View3D
    PointerID 02D10200 RefreshingState(94) with
        ValidRenderWindowSizeInput(89) ---> RefreshingState(94).
```

These log messages have timestamps, pointer addresses (used as object identifiers within the log parsing routine), and state transition messages. The processLogFile tool will convert this snippet into an XML send-events file:

```
<igstk>
  <metadata>
   LogFile Name: "xmlFiles/log.txt"
    Component Name: "View3D"
    Component Instance: "02D10200"
    Time: Sun Nov 23 10:25:39 2008
  </metadata>
  <events>
    <send event="StartRefreshingInput"/>
    <send event="ResetCameraInput"/>
    <send event="ValidRenderWindowSizeInput"/>
  </events>
</igstk>
```

Running Simulations

The main simulator engine can be thought of as a state machine interpreter. It loads one or more state machine instances and processes events targeted for those instances. The simulator is generic in that a simulation may be run as a test case, a visualization exercise, for experimentation, or any other purpose a developer desires. Its use for validation is simply the main motivating use case, in the future we may look to the simulator as more of a design-time tool.

To run the simulator, you need to first configure the environment. Edit the vtconfig.xml file in the properties directory to indicate what state machines and what event files should be loaded. Set your `IGSTKVT_HOME` variable to the root of the IGSTK validation tools installation. Then execute the `run-simulator` Ant target.

Rules as Test Conditions

By default, the simulator checks for the presence of an `igstk.drl` file. This file contains declarative constraints (expressed as Drools⁶ rules) that define test condition logic. The default rules will probably not be useful for your purposes, considering moving this file out of the way and creating a different set of rules to codify your test conditions.

Similar in concept to the path coverage algorithms, there are two types of rules: those that are independent of a given state machine (called *invariant rules* and those that are specific to a given component state machine. The most common invariant rules are those that encode structural checks as described above. More commonly, developers or testers define rules dependent on one or more state machines, and setup observers of rule execution that report when a rule has been violated. This is analogous to the practice of a runtime assertion, except that the constraints are maintained distinct from the code in a text-based rules file for easier maintainability.

The rules framework provides a way to validate global state through declarative constraints expressed among two or more components within a single rule. For example, here is a sample rule that checks if a Tracker object is in an `AttemptingTo` state while the corresponding Spatial Object is in a `read` state:

```
rule "Rule 2"
when
    (and stateMachine : IgstkvtStateMachineExecutor(currentState ==
                "AttemptingToUpdateState", stateMachineId == "1")
        stateMachine1 : IgstkvtStateMachineExecutor( currentState ==
                "ObjectFileNameReadState", stateMachineId == "2"))
then
    stateMachine.notifyListeners("DROOLS: The StateMachine " +
        stateMachine.getStateMachineName() + " is in state " +
        stateMachine.getCurrentState() + " and the StateMachine " +
```

⁶Drools is a popular open source business rules framework that was recently incorporated into the JBoss platform and renamed JBoss Rules. See <http://www.jboss.org/drools>.

```
stateMachine1.getStateMachineName() + " is in state " + " " +
stateMachine1.getCurrentState() + " which is against rules!!!" );
```

Developers have the ability to set these constraints and have the simulator check them on each state machine transition. The results may then be posted to an IGSTK dashboard. ASU maintains an experimental IGSTK dashboard showing the output of the validation tool. Rule output is shown in the IGSTK-VT-TEST area at the bottom of the dashboard.

Visualizing, Animating, Interacting with SMVIZ

Visualizing state machine structure and dynamic behavior provides a powerful way for developers to verify implemented state machines match developer intentions. The first-generation visualization capability provided limited visualization functionality through the use of the Labeled System Transition Analyzer (LTSA [20]). More recently, a from-scratch rewrite of the visualization platform was done to add needed IGSTK features and make a tool available under the IGSTK commercial-friendly license (LTSA's license was not compatible).

A screenshot of the new visualization tool is shown below in Figure 20.3.

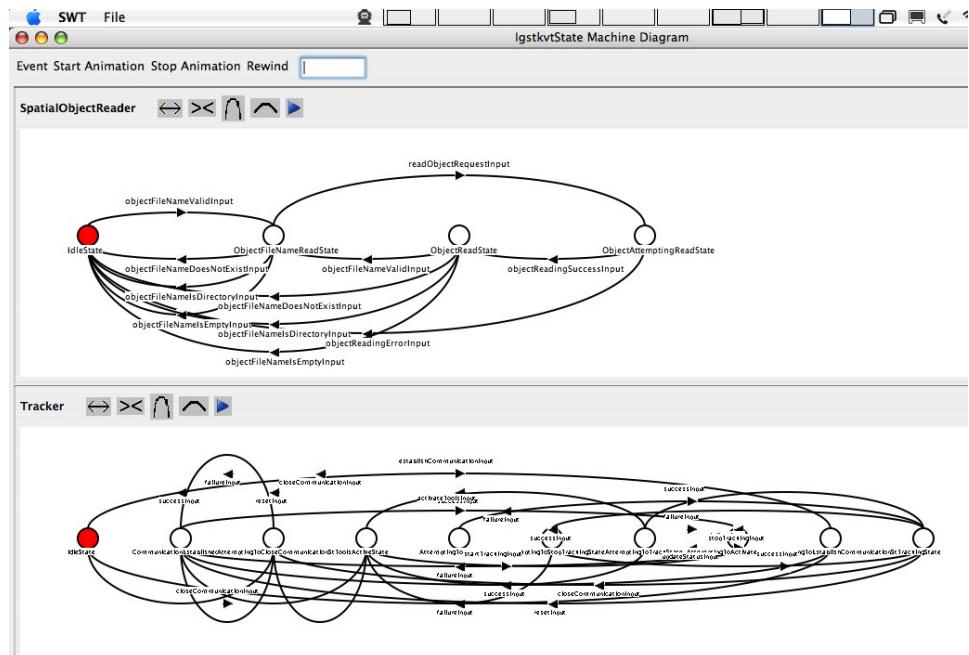


Figure 20.3: State Machine Visualization Tool.

This new tool, *SMVIZ*, is written in Java using the Standard Widget Toolkit from the Eclipse Foundation (SWT). It draws state machines from SCXML representations and animates their

execution from the validation toolset simulator (acting as an observer). It also allows the user to interact an event at a time with a state machine. The animation capability is native and is easier to use than LTSA's XML Scenebeans approach. SMVIZ also shows multiple state machines at a time.

20.5 Conclusion

The IGSTK Validation toolset is an optional add-on to IGSTK. However, it is governed by the same open source license as IGSTK. The toolset validates the safety-by-design approach by providing an automated way to check that a developer's implementation matches the design expectations not explicitly stated in documentation. The toolset also opens the possibility of upfront design and prototyping of complex IGSTK state machines, potentially reducing defects.

20.6 Acknowledgements

Many graduate students at Arizona State University over the years have contributed to various parts of the validation toolset. These students include Beulah David, Mwaffwaq Otoom, Shylaja Kokoori, Benjamin Muffih, Rakesh Kukkamalla, Janakiram Dandibhotla, and Santosh Rajendran.

Part V

Example Applications

Tracking Working Volume Viewer

21.1 Introduction

The size of a tracking system's work volume places constraints on positioning the system's hardware with regard to the desired working space. When using an electromagnetic tracking system the position of the field generator is constrained, and when using an optical system the position of the camera rig is constrained. Most often, we attempt to maximize the overlap between the desired work volume and the tracking system's work volume. As the tracker's work volume is not visible, positioning is often a trial and error process for an inexperienced user.

A simple approach to achieving a good overlap between the expected work space and the tracking work space is to use a graphical user interface to guide the user. The example program we describe here provides this functionality.

One or more sensors are placed in the expected working volume and their position inside the tracked space is displayed. The field generator or camera rig is moved so that all the sensors appear inside the measurement volume. The program's user interface is shown in Figure 21.1. Note that this is an FLTK application which requires setting the Cmakelists `IGSTK_USE_FLTK` flag to ON.

The program requires that the user provide a mesh model of the working volume and an optional mesh model of the field generator or camera rig. Sample mesh files for several tracking systems can be found in the `IGSTK` source directory under `Testing/Data/Input/TrackerWorkingVolumeMeshes`.

The program demonstrates the use of spatial objects, data re-slicing and working with a tracking system. The spatial objects include `igstk::AxesObject`, `igstk::EllipsoidObject`, and `igstk::MeshObject`. They are displayed using their corresponding representations `igstk::AxesObjectRepresentation`, `igstk::EllipsoidObjectRepresentation`, and `igstk::MeshObjectRepresentation`. To obtain the outer contour of the mesh representing the tracker's work volume we use the `igstk::ReslicerPlaneSpatialObject` class to define the dynamic re-slice plane equation based on the tracking data, with the `igstk::MeshResliceRepresentation` computing the contour from the specified mesh and

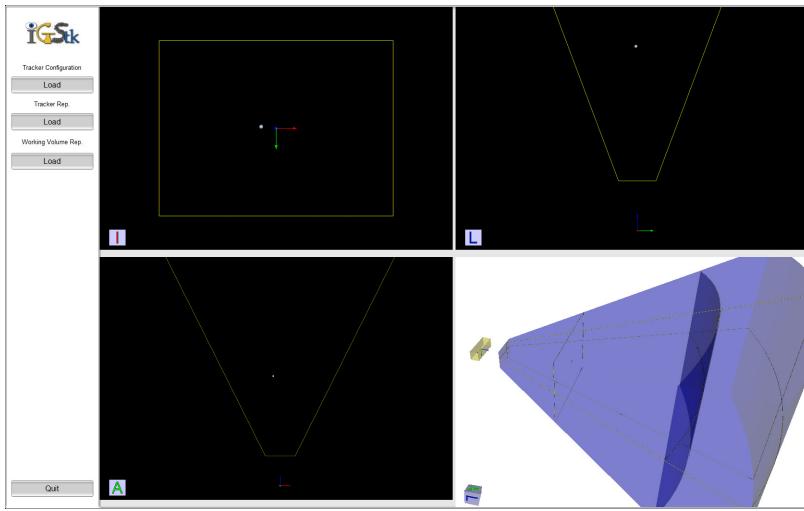


Figure 21.1: Tracking Working Volume Viewer GUI. GUI displaying the working volume of the Micron-Tracker system, reslicing through the volume and a 3D view. Tools are displayed as spheres.

plane, and displaying it.

To easily use all tracking systems supported by IGSTK the program also requires an XML file describing the desired tracker configuration. We next describe the set of classes that enable the development of tracker-independent programs and the supported XML file formats.

21.2 Tracker-independent Application Development

In this section we describe a number of utility classes that enable developers to implement programs which are independent of specific tracking devices. By using these classes a program will transparently support all IGSTK supported tracking devices with minimal effort on the part of the developer.

These utility classes include:

1. `TrackerController` - Replaces the direct use of the core IGSTK tracker classes. Parameterized using a configuration object specifying tracker and tool settings.
2. `TrackerConfiguration` - Abstract superclass for the various tracking devices supported by IGSTK. An adapter between GUI/file reader classes and the `TrackerController` class. Serves as a container for the parameters defining the desired tracker and tool setup. Developer uses subclasses corresponding to actual tracking devices (e.g. `MicronTrackerConfiguration`).

3. `TrackerConfigurationFileReader` - A class that is used to read XML configuration files describing the desired tracker and tool setup. Parametrized using an xml reader corresponding to a specific tracker type (e.g. `PolarisVicraConfigurationXMLFileReader`).

TrackerController functionality:

1. Transparently initialize any tracker supported in IGSTK (`RequestInitialize()`). The configuration given to the initializer method describes the desired tracker and tool configuration, including the option to designate one of the tools as a dynamic reference frame.
2. Start, stop, and shut down tracking (`RequestStartTracking()`, `RequestStopTracking()`, `RequestShutdown()`).
3. Get a list of all non-reference tools, access a specific non-reference tool, access the reference tool if using one (`RequestGetNonReferenceToolList()`, `RequestGetTool()`, `RequestGetReferenceTool()`).
4. Add a spatial object as a scene graph child or parent of the tracking device (`RequestSetParentSpatialObject()`, `RequestAddChildSpatialObject()`).

TrackerConfiguration functionality:

1. Set the tracking device's refresh rate (`RequestSetFrequency()`).
2. Add a tool to the configuration (`RequestAddTool()`).
3. Add a tool as a dynamic reference frame (`RequestAddReference()`).

Additional tracker specific configuration settings are found in the corresponding subclasses.

TrackerConfigurationFileReader functionality:

1. Set the name of the file we want to read (`RequestSetFileName()`).
2. Set the specific xml reader corresponding to a specific tracking system (`RequestSetReader()`).
3. Try to read the file (`RequestRead()`).
4. Get the configuration data (`RequestGetData()`).

Sample xml configuration files for all tracking devices can be found in the IGSTK build directory under Data/TrackerConfiguration.

We now show how these classes work together to load a configuration file and initialize the tracking device.

21.3 Example

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

`Examples/TrackerConfiguration/TrackerConfigurationExample.cxx`.

This example illustrates how to use xml readers and the TrackerController class to initialize a specific tracker configuration described in an xml configuration file. To use the tracker controller we need to include the following files:

```
#include "igstkTrackerConfiguration.h"
#include "igstkTrackerController.h"
```

To enable reading of all supported trackers we need to include the corresponding xml file readers:

```
#include "igstkTrackerConfigurationFileReader.h"
#include "igstkPolarisVicraConfigurationXMLFileReader.h"
#include "igstkPolarisSpectraConfigurationXMLFileReader.h"
#include "igstkPolarisHybridConfigurationXMLFileReader.h"
#include "igstkAuroraConfigurationXMLFileReader.h"
#include "igstkMicronConfigurationXMLFileReader.h"
#include "igstkAscensionConfigurationXMLFileReader.h"
```

Instantiate all xml readers:

```
const unsigned int NUM_TRACKER_TYPES = 6;
igstk::TrackerConfigurationXMLFileReaderBase::Pointer
    trackerConfigurationXMLReaders[NUM_TRACKER_TYPES];
trackerConfigurationXMLReaders[0] =
    igstk::PolarisVicraConfigurationXMLFileReader::New();
trackerConfigurationXMLReaders[1] =
    igstk::PolarisSpectraConfigurationXMLFileReader::New();
trackerConfigurationXMLReaders[2] =
    igstk::PolarisHybridConfigurationXMLFileReader::New();
trackerConfigurationXMLReaders[3] =
    igstk::AuroraConfigurationXMLFileReader::New();
trackerConfigurationXMLReaders[4] =
    igstk::MicronConfigurationXMLFileReader::New();
trackerConfigurationXMLReaders[5] =
    igstk::AscensionConfigurationXMLFileReader::New();
```

We then instantiate the configuration file reader which is parameterized using the xml readers.

```
igstk::TrackerConfigurationFileReader::Pointer trackerConfigReader =
    igstk::TrackerConfigurationFileReader::New();
```

Set the file name we want to read.

```
trackerConfigReader->RequestSetFileName( argv[1] );
```

Try to read the given file by using each of the xml readers till one of them succeeds.

```
for( unsigned int i=0;
      trackerConfiguration == NULL && i<NUM_TRACKER_TYPES;
      i++ )
{
//setting the xml reader always succeeds so I don't
//observe the success event
trackerConfigReader->RequestSetReader( trackerCofigurationXMLReaders[i] );

//try to read using the current xml reader
trackerConfigReader->RequestRead();

//xml file doesn't match current reader
if( rfso->GotUnexpectedTrackerType() )
{
    rfso->Reset();
} //xml file matches current reader, but there
//is a problem with the xml data
else if( rfso->GotFailure() && !rfso->GotUnexpectedTrackerType() )
{
    std::cerr<<"Failed reading configuration file (";
    std::cerr<<rfso->GetFailureMessage()<<").\n";
    return EXIT_FAILURE;
} //we read the xml data and everything is fine
else if( rfso->GotSuccess() )
{
    //get the configuration data from the reader
    trackerConfigReader->AddObserver(
        igstk::TrackerConfigurationFileReader::TrackerConfigurationDataEvent(),
        tco );
    trackerConfigReader->RequestGetData();

    if( tco->GotTrackerConfiguration() )
    {
        trackerConfiguration = tco->GetTrackerConfiguration();
    }
}
```

Instantiate TrackerController.

```
igstk::TrackerController::Pointer trackerController =
    igstk::TrackerController::New();
```

Initialize TrackerController, initializes the specific tracker.

```
trackerController->RequestInitialize( trackerConfiguration );
```

Needle Biopsy 2.0

This application is an enhanced version of the Needle Biopsy application described in Chapter 24. The new features include: support for more trackers (Aurora, Polaris, and Micron), simplified user interface, usage of the new coordinate system component, and support for a simple treatment plan read/write. The main purpose of this chapter is to describe how to use the application and present discussion on scene graph generation for image-guided surgery applications. The discussion is meant to supplement to the examples shown in Chapter 8.

Figure 22.1 shows the user interface of the new needle biopsy application written in FLTK. A drop down menu is provided to select choice of tracking system. This application can connect to multiple trackers simultaneously and one can choose different types of tracking tools. Although the application provides support for different tracking systems, co-registration between the different systems is not implemented.

22.1 Running the Application

To build this application, first make sure the `IGSTK_USE_FLTK` and `IGSTK_BUILD_EXAMPLES` options are turned on in CMake configuration step. If you want to use Micron tracker with this application, make sure you turn on `IGSTK_USE_MicronTracker` option and specify the MTC include headers directory and library. If you follow this process and successfully built the IGSTK, you should find the application executable in your binary directory.

Other requirements to run the application include:

1. A Polaris, Aurora, or Micron Tracker connected to your system, with necessary drivers installed and additional files for setting up the tracker (such as SROMs, markers, and camera calibration file)
2. An image volume with fiducial markers in it. Image should be DICOM format with CT or MR as modality, and the directory should contain only one series. The fiducials will be used for landmark registration.

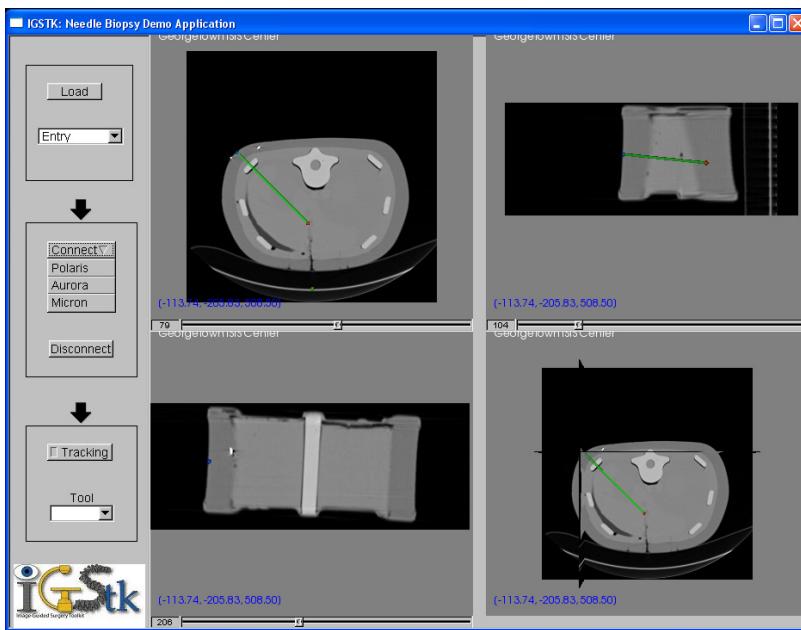


Figure 22.1: User Interface for the New Needle Biopsy Program.

3. A calibrated pointer tool. You can set the tooltip offset in the tracker configuration window.
4. A reference tool is optional.

How to run the application:

1. Run the executable.
2. Load the image series from disk.
3. The application will automatically look for the corresponding treatment plan for this image series (DirName_TreatmentPlan.igstk). If the plan file doesn't exist, it will create a default plan of 5 points (entry, target, and 3 fiducials). An example plan file is shown below. For more details, please refer to classes `igstkTreatmentPlan` and `igstkTreatmentPlanIO` in the example source code directory.
4. You can change the values for these points by selecting them from the drop-down choice box and then click on the image. Annotation should show the updated value in red. Meanwhile the values will also be saved to the disk, next time you load the same image, it will read these values back.
5. You can then choose to connect to tracker, after setting all the parameters, you can click confirm, if success, the registration window will appear.

6. You need to use the tracker tool to touch the fiducial on the object, in the same sequence as in the planning part, and then click on set tracker fiducial button. It will also jump to the next fiducial for the user to set.
7. After setting all the fiducials, you can click on Registration as this will do the landmark registration between image space and tracker space. If it returns successfully, a new tracker and tracker tool will be added to the list box. The view windows should also start tracking the tool location.
8. If you have multiple trackers, or tracker tools connected, you can change the active tool from the choice box, which will switch the tool that the View window is following.

Example treatment plan file format:

```
# Entry point  
0.820425 -143.635 -186  
# Target point  
54.268 -108.513 -191  
# Fiducial points  
98.4887 -152.976 -181  
-1.89214 -148.996 -191  
-59.2006 -190.563 -191
```

22.2 Building the Scene Graph

Buidling a scene graph is a critical part of the application development process. The coordinate systems of the objects involved in the surgical procedure have to be correctly wired. Below is a a step-by-step discussion of architectural considerations.

1. Choose a world coordinate system as the initial point of reference, you can use any class that has a coordinate system API, such as SpatialObject, Tracker, TrackerTool, and View. In this case we made up a virtual "WorldReference" using an AxesSpatialObject.

```
m_WorldReference = igstk::AxesObject::New();
```



Figure 22.2: The Initial Scene Graph.

2. Now we need to add objects to the scene by connecting them to the WorldReference. At this stage we are just going to link all objects to the world reference system using the identity transform. Notice that we use the igstk::TimeStamp::GetLongestPossibleTime() as the valid period of the transform. This transform is essentially a constant transform. *Don't forget to connect the View. Only when the View is connected to the scene graph will it be able to render the scene.*

```
igstk::Transform transform;
transform.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );
ViewerGroup->m_VIEWS[i]->RequestSetTransformAndParent( transform,
                                                       m_WorldReference );
m_ImageSpatialObject->RequestSetTransformAndParent( transform,
                                                       m_WorldReference );
m_Needle->RequestSetTransformAndParent( transform, m_WorldReference );
m_NeedleTip->RequestSetTransformAndParent( transform,
                                            m_WorldReference );
```

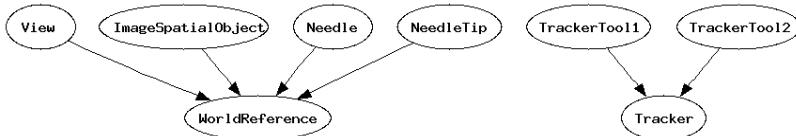


Figure 22.3: Adding Objects into the Scene Graph.

3. Pay attention when you add Tracker and TrackerTool into the scene graph tree. If you are using a reference tool, you should connect the reference TrackerTool to the world reference system. If you do not use a reference tool, you should connect Tracker to the world reference directly. Tracker and TrackerTools are automatically connected when you request attaching TrackerTools to a Tracker. The Tracker is the parent to all the TrackerTools. If you set one of the TrackerTools as the reference, this particular TrackerTool will become the parent of the Tracker in the scene graph.

```
/** Connect the scene graph with an identity transform first */
igstk::Transform transform;
transform.SetToIdentity( igstk::TimeStamp::GetLongestPossibleTime() );
if ( m_TrackerInitializer->HasReferenceTool() )
{
    refTool->RequestSetTransformAndParent( transform, m_WorldReference );
}
else
{
    tracker->RequestSetTransformAndParent( transform, m_WorldReference );
}
```

Now we also want our needle to move with TrackerTool1, so we attach the spatial object as the child of the TrackerTool1. To avoid circular connections, we designed the scene graph so that it refuses to connect to a parent when a circular connection is detected. If you want to change the scene graph tree structure, it is safe to detach it from its parents and then re-attach it.

```
igstk::Transform transform;
transform.SetToIdentity(igstk::TimeStamp::GetLongestPossibleTime());
m_Needle->RequestDetachFromParent();
m_NeedleTip->RequestDetachFromParent();
m_Needle->RequestSetTransformAndParent( transform, m_ActiveTool );
m_NeedleTip->RequestSetTransformAndParent( transform, m_ActiveTool );
```

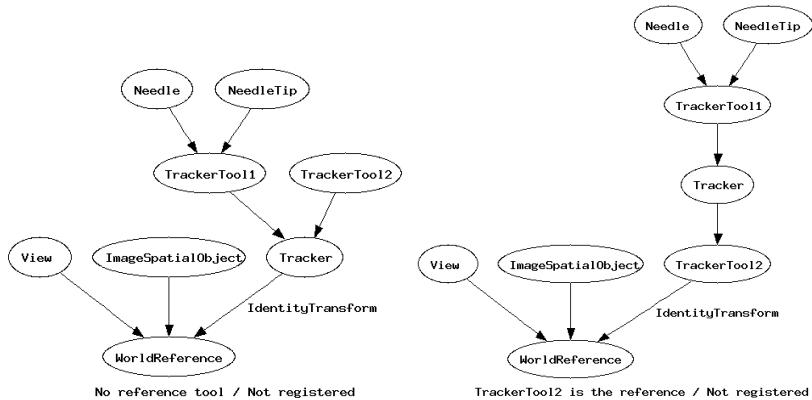


Figure 22.4: Connecting a Tracker to the Scene Graph.

4. After registering the pre-operative image with the tracking coordinate system using landmark registration, you can replace the transform in the previous code with the LandmarkRegistrationTransform. When doing the registration, be careful about which two frames you are registering. In the case of using TrackerTool2 as the reference tool, image landmark points are in the WorldReference, and tracker landmark points are in TrackerTool2's frame. Which means, when you are acquiring tracker landmark points using TrackerTool1, you should invoke the following:

```
trackerTool1->RequestComputeTransformTo( TrackerTool2 );
```

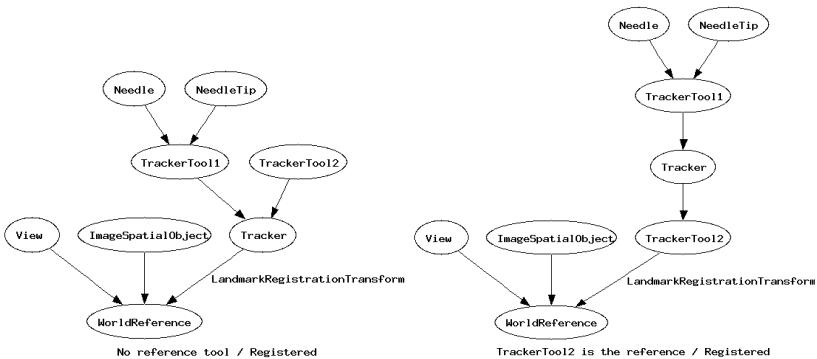


Figure 22.5: Setting the Registration Transform.

5. After starting tracking, if you want to get the overlaid position of TrackerTool in the image space, you can do either of the following. They both have the same effects (since ImageSpatialObject is attached to WorldReference using an identity transform).

```
trackerTool1->RequestComputeTransformTo( m_ImageSpatialObject );
trackerTool1->RequestComputeTransformTo( m_WorldReference );
```

You need to connect an observer to the trackerTool1 to listen to the returned transform.

```
// Connect observer
typedef igstk::TransformObserver ObserverType;
ObserverType::Pointer transformObserver = ObserverType::New();
transformObserver->ObserveTransformEventsFrom( m_ActiveTool );
transformObserver->Clear();

// Request for transform
m_ActiveTool->RequestComputeTransformTo( m_WorldReference );

// Check the recipient
if ( transformObserver->GotTransform() )
{
    // Retreive the transform
    igstk::Transform transform = transformObserver->GetTransform() ;
}
```

6. As mentioned above, one can use any class that has a coordinate system API as the world coordinate reference. For example, one can use ImageSpatialObject as the world coordinate system, so the above example is equivalent to:

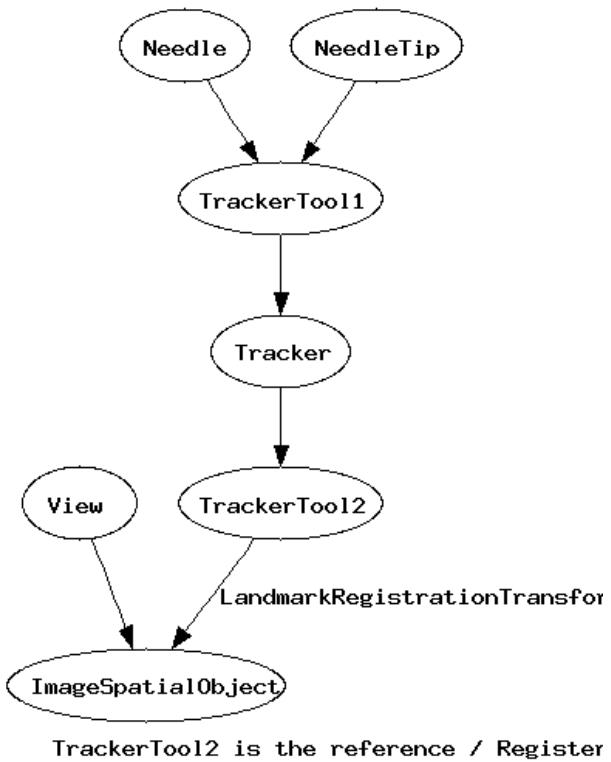


Figure 22.6: Using an Image Object as the Root of the Scene Graph.

7. The flexible scene graph design not only allows you to compute the transform between any two connected points in the scene graph tree but also allows you to see the world from different perspectives. For example, if you want to observe the scene from a needle tip point of view as it is moving in the patient anatomy, just attach the View to the NeedleTip as follows.

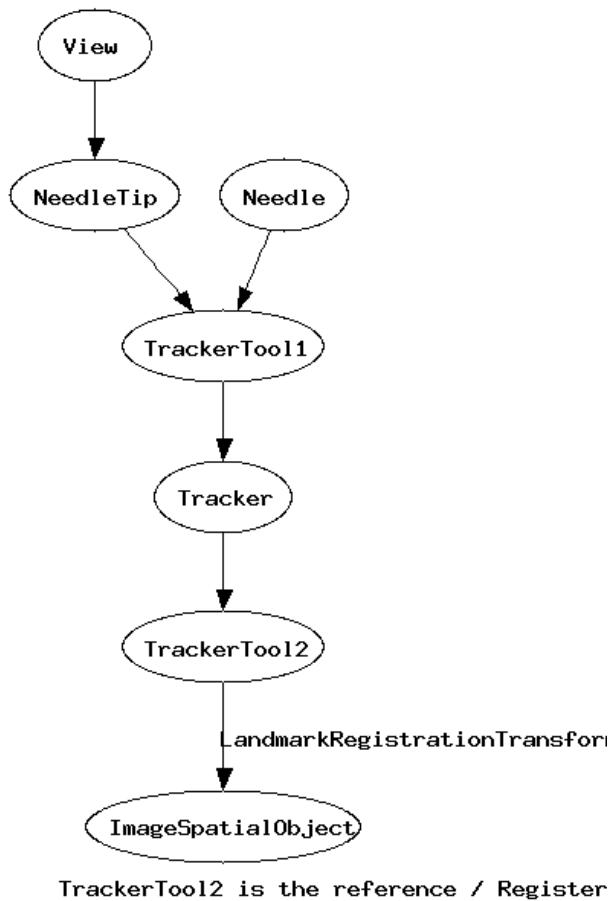


Figure 22.7: Attaching the View to the Needle Tip.

Navigator

23.1 Introduction

The Navigator application demonstrates the use of various IGSTK components. The application provides the basic functionality required by procedures dealing with rigid anatomical structures and is intended to serve as a starting point for developing procedure specific navigation systems. The application's user interface was developed using FLTK, and requires that the Cmakelists `IGSTK_USE_FLTK` flag be set to ON.

The application has built in logging facilities, using the framework described in Chapter 13. A description of user actions is written to a text file `logNavigator_DATE_TIME.txt` which documents the procedure. Navigation guidance is provided using three reformatted image slices through the volumetric image data and a three dimensional rendering of the data. In addition, mesh data representing segmented anatomical structures can be displayed. The application's user interface is shown in Figure 23.1. Prior to navigation the program will display images using the standard axial, sagittal, and coronal reformatting planes. After registration the user can choose to use off-axial or oblique reformatting views as described in Chapter 17.

We next describe the application using the workflow shown in Figure 23.2.

23.2 Load Image

The first step in the workflow is loading of a volumetric data set representing the underlying anatomy, in our case a CT. The user provides the path to the directory containing the data. It is assumed that the directory contains a single DICOM series. To load the volume the program uses the `igstk::CTImageReader` class. This ensures that only CT data is used for navigation. To use MR as the image type simply replace the reader with `igstk::MRIImageReader`. Once the image is loaded the user is asked to confirm that the patient's name appearing in the CT data corresponds to the actual patient.

In addition, if a file named "Fiducials.igstk" is found in the same directory as the image series

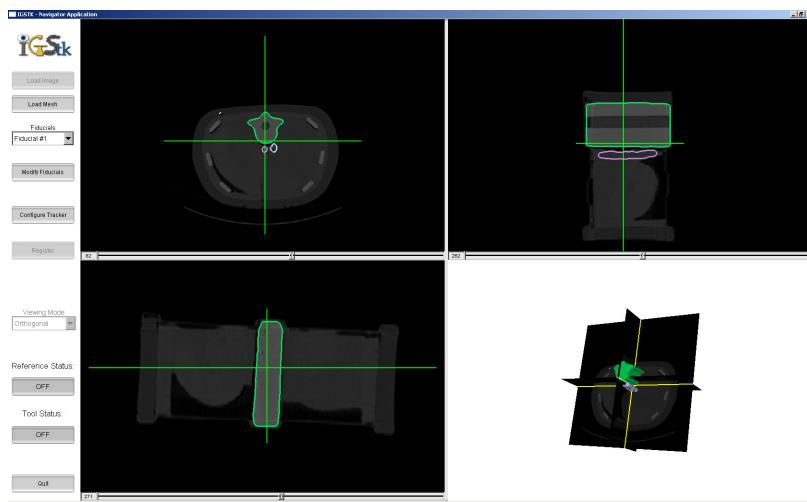


Figure 23.1: Navigator Application Interface.

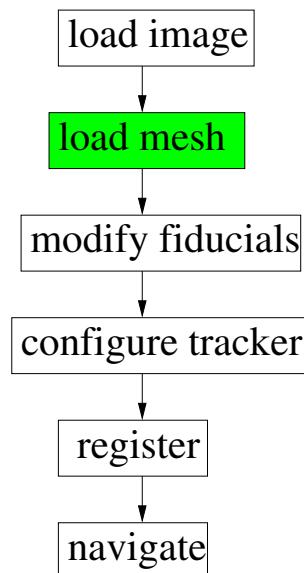


Figure 23.2: Navigator Application Workflow. The shaded (green) step is optional.

the coordinates given in that file are used as the fiducial coordinates in image space. The classes used to perform this task are `igstk::FiducialsPlanIO` and `igstk::FiducialsPlan`. After loading the point data from file the user can modify the data before proceeding to the next step. If no file is found the application will let the user specify four fiducials in image space. A sample file containing the coordinates of five fiducials is shown below:

```
# Fiducial points
-23.1555    -165.249    561.513
-83.5239    -163.348    561.525
-82.5732    -182.837    561.538
-25.5322    -184.738    561.55
-27.5322    -170.738    571.55
```

If there are more than four fiducials and you do not have their coordinates simply create a text file containing the appropriate number of fiducials with all coordinates set to zero. The program will then let you modify the coordinates of all fiducials to their correct values.

23.3 Load Mesh

This step is optional, allowing the user to load multiple mesh objects that are displayed in the context of the CT image. This is useful for displaying soft tissue structures segmented in MR while navigating using a CT data set. In the reformatted image views the contours of the mesh are overlayed onto the reformatted image, with the mesh object displayed in the 3D view. The classes used to perform this task are `igstk::MeshObject` and `igstk::MeshReader`. The program screen shot shown in Figure 23.1 shows the results of loading mesh data.

23.4 Modify Fiducials

This step allows the user to enter or modify the coordinates of fiducials in image space. Press the "Modify Fiducials" button to start the process. Select the fiducial from the choice box. Find the fiducial in the reformatted images. If you want to zoom in/out you need to press the right mouse button and move up/down. To pan the image you need to press the middle mouse button. Finally, press the left button on the fiducial location. Repeat this for all fiducials available in the choice box. Once you are done press the "Modify Fiducials" button. The fiducial data will be used in the registration step and is also saved to the directory containing the DICOM data in the "Fiducials.igstk" ASCII file.

23.5 Configure Tracker

Press the "Configure Tracker" button to load additional data required for navigation. You will first be asked to choose a mesh representation for the tool that will be used dur-

ing the procedure. Sample mesh files can be found in the IGSTK source directory under Testing/Data/Input/TrackerToolRepresentationMeshes. You will then be requested to select an XML file containing the tracker configuration. Note that the application is written so that it is independent of the specific tracker type, as described in Chapter 21.

The application expects that the configuration file contain two tools, one designated as a dynamic reference frame and the other is the tool used to perform the intervention. The XML tag *name* for the tool used as the dynamic reference frame is "reference" and for the intervention tool it is "tool". Also the reference tool must have the *usage* tag set to "reference". A sample configuration using the Vicra (Northern Digital Inc.) tracking system is shown below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

    <!-- the tracking system type -->
<tracking_system type = "polaris vicra">

    <!-- acquire tracking data at this rate [Hz] -->
<refresh_rate> 20 </refresh_rate>

    <!-- all the serial communication settings -->
<serial_communication>
    <com_port>5</com_port>
    <baud_rate>115200</baud_rate>
    <data_bits>8</data_bits>
    <parity>N</parity>
    <stop_bits>1</stop_bits>
    <hand_shake>0</hand_shake>
</serial_communication>

    <!-- the dynamic reference frame -->
<tool usage="reference">
    <name> reference frame </name>
    <srom_file>drf.rom</srom_file>
</tool>

    <!-- the pointer tool -->
<tool>
    <name> tool </name>
    <srom_file> bayonetProbe.rom</srom_file>
    <calibration_file>bayonetCalibration.xml</calibration_file>
</tool>

</tracking_system>
```

Once the tracker configuration has been read the tracking device will be initialized and the "Register" status button will change color to red, denoting that the image space data is available

but that the patient space data has not been acquired. The reference and tool status buttons are initially yellow (OFF) until the tracking device is able to acquire a measurement. When the tools are localized the status buttons are green (tracking), and if the localization fails the button turns red (not visible).

23.6 Register

To perform registration we need to digitize the fiducials in the physical space (patient space). From the choice box you can select the fiducial that you currently want to digitize. Place the calibrated tool's tip on the fiducial and press the letter "h" on the keyboard. This will acquire the tip location and associate it with the selected fiducial. When the location is successfully acquired you receive audible feedback. The choice box will automatically change to the next fiducial on the list. Once you have acquired three fiducials the registration status button will turn green (Ready), indicating that the minimal number of fiducials for registration have been successfully digitized. You can continue to digitize the rest of the fiducials.

Once you have digitized the fiducials you can invoke the registration by pressing the letter "g" on the keyboard. This will perform the registration, report the resulting root mean squared error (RMS), and you will be asked to confirm whether to continue or not. Once you accept the registration you can start navigating.

Note that the registration RMS is no more than a sanity check and should never be used as the quality measure of the target registration error, as these quantities are not correlated [8].

23.7 Navigate

Once the registration has been accepted the program is in navigation mode. That is, if the tool and reference are not visible the image data is not displayed. When the images are displayed the reformatting is driven by the tool's position and orientation. The user can select a specific reformatting approach from the choice box. Detailed descriptions of the reformatting options are given in Chapter 17.

Part VI

Legacy Applications

CHAPTER

TWENTY FOUR

Needle Biopsy

The following three examples are from the previous release of the toolkit. As the toolkit continues to evolve, they are now outdated. However, we preserve these chapters in this edition of the book to show the extensibility and potential applications of the toolkit.

In this part of the book, we will present three example applications developed using IGSTK. These applications come from our observation of clinical procedures, such as needle biopsy, ultrasound-guided radio frequency ablation, and robot-assisted needle placement. You will be able to learn the concepts and workflows 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. This definition for needle biopsy procedure is from the Society of Interventional Radiology's website:

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.¹

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 overlaid image provides guidance to the surgeon for better targeting of the needle to its desired location.

Figure 24.1 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.

¹<http://www.sirweb.org/patPub/needleBiopsy.shtml>

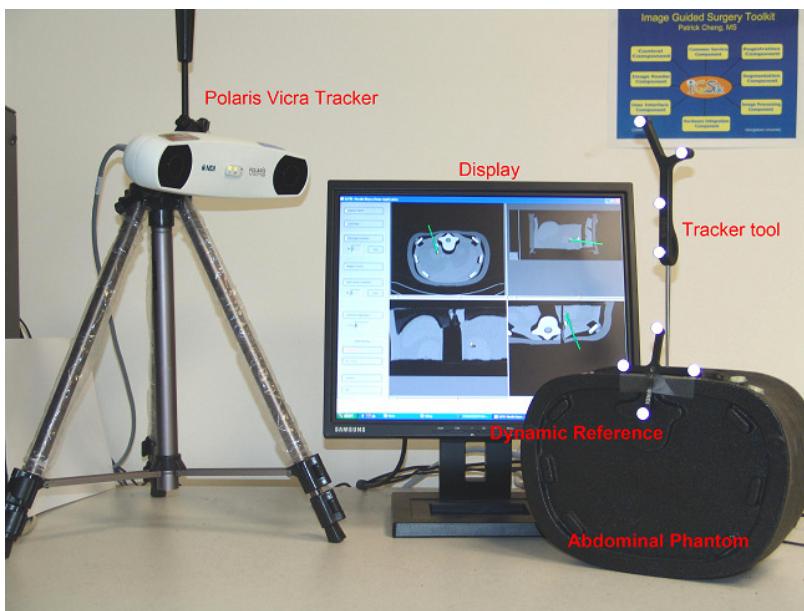


Figure 24.1: System Setup for Needle Biopsy Application.

24.1 Running the Application

This application can be found in `Examples/NeedleBiopsy`. To build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you will need to make sure that `IGSTK_USE_FLTK` is turned ON when running CMake.

The following steps outline the workflow 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 through 7.
3. Verify the patient information against the information in the DICOM tags. 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 four are preferred.
5. Initialize the tracking device.

6. Add patient landmarks by touching the physical fiducials attached to the patient using the tracked pointer device.
7. Perform the image to patient landmark registration.
8. 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.

24.2 Implementation

The state machine is a very important and distinct feature in IGSTK. We try to provide some guidance here to help users to become familiar with this concept and design pattern.

24.2.1 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 a 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 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, a state machine is implemented at the application level (Figure 24.2 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 workflow of the surgical procedure, and it 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.

24.2.2 Mapping Clinical Work Flow to a 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 series of tasks or workflow. Then it becomes relatively easy to translate clinical workflow into state machine logic. If we think of 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. For instance, when we click on the register patient button, this will generate a “RequestSetNameInput” to the state machine. The state machine will take this input and change its



Figure 24.2: State Machine Diagram (Partial) for the Needle Biopsy Application (circles indicate states and arrows for transitions and corresponding inputs).

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 workflow for the application. Figure 24.2 shows part of the state machine diagram for the needle biopsy application which was generated automatically when the state machine was constructed using the ‘dot’ tool from Graphviz.

24.2.3 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`.

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 application.

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, RequestSetName,
                        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 set up 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 ensure the safety of the state machine and to prevent accidental 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 24.2. 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 respond 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.

24.2.4 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 use the state machine programming pattern. 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 24.2. They can go through the diagram and examine and verify their design of the workflow. 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.

24.3 Results

Figure 24.3 shows the user interface of the needle biopsy application written in FLTK. The left side is the control panel, consisting 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 will not 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 views. Here we loaded CT images of an abdominal phantom. The green cylinder represents the needle being tracked by the tracker. The viewer will automatically reslice the images as the needle tip moves in the anatomy.

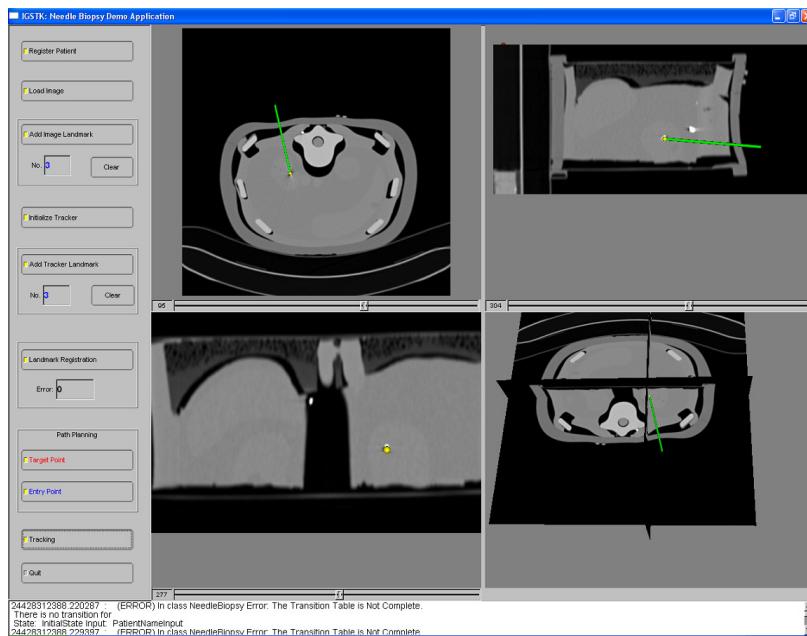


Figure 24.3: User Interface for Needle Biopsy Program.

Ultrasound-Guided Radiofrequency Ablation

This example is from the previous release of the toolkit. As the toolkit continues to evolve, it is now outdated. However, we preserve this chapter in this edition of the book to show the extensibility and potential applications of the toolkit.

25.1 Introduction

Liver lesions suitable to be treated using radiofrequency ablation (RFA) are often clearly visible under CT or MR but not ultrasound imaging. Therefore, most RFA procedures of the liver are performed under CT. An ideal visualization system would show tumor contours under ultrasound to the physician. A typical workflow of an RFA procedure is shown in Figure 25.1.

The ultrasound-guided RFA application registers a pre-operative model of the tumor with a 2D ultrasound slice in pseudo real-time. This system can be divided into three parts: a) tracking, b) registration, 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 the pre-operative CT. One can see that this registration step should be performed as quickly as possible. Third, a display presents the actual 2D ultrasound slice with tumor outlines to the surgeon.

Next, the different components of the application, the tracking systems and the registration algorithm, are presented.

25.2 Running the Application

This application can be found in the Examples/UltrasoundGuidedRFA directory. To build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org.

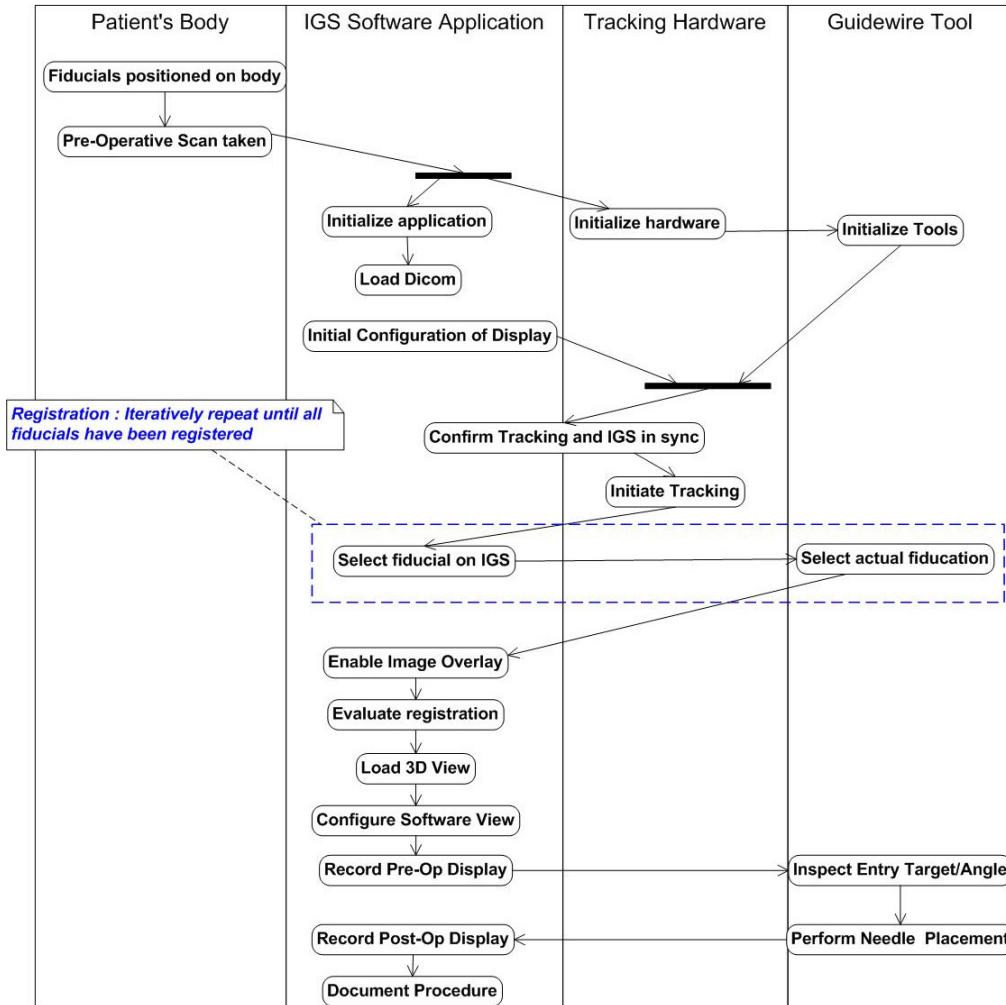


Figure 25.1: Typical RFA Ablation Procedure Workflow.

Then you will need to make sure that `IGSTK_USE_FLTK` is turned ON when running CMake, and that the `FLTK_DIR` and `FLTK_FLUID_EXECUTABLE` variables are pointing to the right locations.

25.3 Implementation

25.3.1 Tracker

In this section we present how the tracker class is used in the application.

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 the arm of a 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. For example:

```
m_Tracker->SetToolCalibrationTransform( TRACKER_TOOL_PORT, 0,
ToolCalibrationTransform);
```

Next, we define the relationship between the tracking system origin and the actual patient position in the operating room. Most of the surgical applications define the operating room as the world coordinate origin. This `PatientTransform` is often assessed via calibration. For example:

```
m_Tracker->SetPatientTransform(PatientTransform);
```

The `IGSTK` spatial object to be tracked is attached to the tracker using the `AttachObjectToTrackerTool()` function. Therefore, when the position and orientation of the tracking device is modified, the position of the spatial object is automatically updated. One can notice that this step involves the concatenation of the `ToolCalibrationTransform` with the `PatientTransform`. For example:

```
m_Tracker->AttachObjectToTrackerTool( TRACKER_TOOL_PORT,
                                         TRACKER_TOOL_NUMBER,
                                         m_UltrasoundProbe);
```

Now the tracker is calibrated and ready. We start the tracking by opening the serial communication port using the function `Open()`. Then we initialize the tracker and start the tracking:

```
m_Tracker->RequestOpen();  
m_Tracker->RequestInitialize();  
m_Tracker->RequestStartTracking();
```

To stop the tracking device we use the `StopTracking()` function:

```
m_Tracker->RequestStopTracking();
```

One can see that switching from one tracker to another can be done by modifying a single line of code, i.e. the tracker type definition.

25.3.2 Registration

Using the tracking information of the ultrasound probe, the location of the ultrasound slice is roughly defined in the patient. To define a proper alignment of the ultrasound 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 also proposes algorithms already tuned for specific modalities and organs. For instance, the `igstkMR3DImageToUS3DImageRegistration` class performs registration of any 3D MR to 3D ultrasound image. The parameters of the registration are already tuned to support most of the MR to ultrasound registrations, but some other tuning might be required for different organs. Using class hierarchies, programmers can still make use of higher level registration techniques.

25.3.3 Read and Display

In this section, we give an example of 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 file name, and then ask the reader to read the file using the `RequestReadObject()` function:

```
m_VascularNetworkReader->RequestSetFileName(VasculatureFilename);  
m_VascularNetworkReader->RequestReadObject();
```

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. For example:

```
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 =
        VascularNetworkRepresentationType::New();
```

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 );
```

25.4 Conclusion

We have shown that a simple image-guided surgery application can easily be implemented using IGSTK. The integration of a tracker, an image reader, and a display class allows for simple interaction between the components and fast prototyping. This application is still in development.

Robot-Assisted Needle Placement

This example is from the previous release of the toolkit. As the toolkit continues to evolve, it is now outdated. However, we preserve this chapter in this edition of the book to show the extensibility and potential applications of the toolkit.

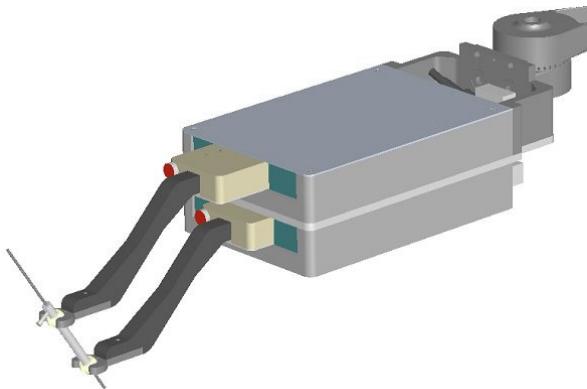


Figure 26.1: B-RobII Four Degree-of-Freedom Precision Placement Modules for Needle Positioning and Orientation. Courtesy of Gernot Kronreif and Martin Furst, ARC Seibersdorf Research GmbH, Vienna, Austria.

In the example image-guided needle biopsy application (Chapter 24), precision placement of the instrument still relies on the hand-eye coordination of the physician. We are also investigating robotic systems to see if they might provide more accuracy in instrument placement. In this chapter, we present an image-guided platform for precision placement of surgical instruments based upon a small four degree-of-freedom robot as shown in Figure 26.1 (B-RobII, ARC Seibersdorf Research GmbH, Vienna, Austria). The robot has two joints (upper box and lower box, which can move parallel to each other) and 4 degrees-of-freedom. 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 at the start of the intervention. Potential abdominal targets include the liver, kidney, prostate, and spine.

Figure 26.2 shows the setup of the whole system. The robot is mounted on the CT table after the patient (phantom) is positioned. The robot arm is adjusted to position the needle holder close to the biopsy area. A CT scan is then acquired and loaded into the software application. The physician 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 move the needle holder and automatically align it with the planned path. The physician can then advance the needle manually towards the target. The robot can be operated remotely by a client through a TCP/IP communication. The client application should first connect to the server application as an active client before it can command the robot.



Figure 26.2: Robot Assisted Needle Placement Phantom Study Setup. Only phantom studies have been done so far.

26.1 Running the Application

This application can be found in `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_FLT` is turned ON when running CMake.

As shown in Figure 26.3, the workflow of this application for a lung biopsy procedure is:

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.
4. Load the 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 graduation on the needle itself.

26.2 Implementation

This application is unique in that it uses some classes that are not in the core IGSTK library. We have already introduced the concept of this application. In the implementation section, we will be focusing on the topic of writing your own code under the IGSTK framework. In other words, we will teach you how to interface with and extend the IGSTK library.

26.2.1 Pass IGSTK Image Objects to ITK Filters

One task of the application is to automatically detect the fiducial pattern in the CT images. It makes sense to borrow from ITK, as it has an extensive library for a wide variety of image analysis algorithms. IGSTK uses `itkOrientedImage` inside the `igstkImageSpatialObject`, but ITK objects are encapsulated under IGSTK API, so we need to get the native ITK object out and pass it to the ITK filters to perform segmentation or registration. In IGSTK we pass information using a loaded “event,” and we need an “observer” to catch that “event”.

1. First, we need to use the `igstkObserverConstObjectMacro` to define the observer:

```
igstkObserverConstObjectMacro( ITKImage,
                               ImageSpatialObjectType::ITKImageModifiedEvent,
                               ITKImageType)
```

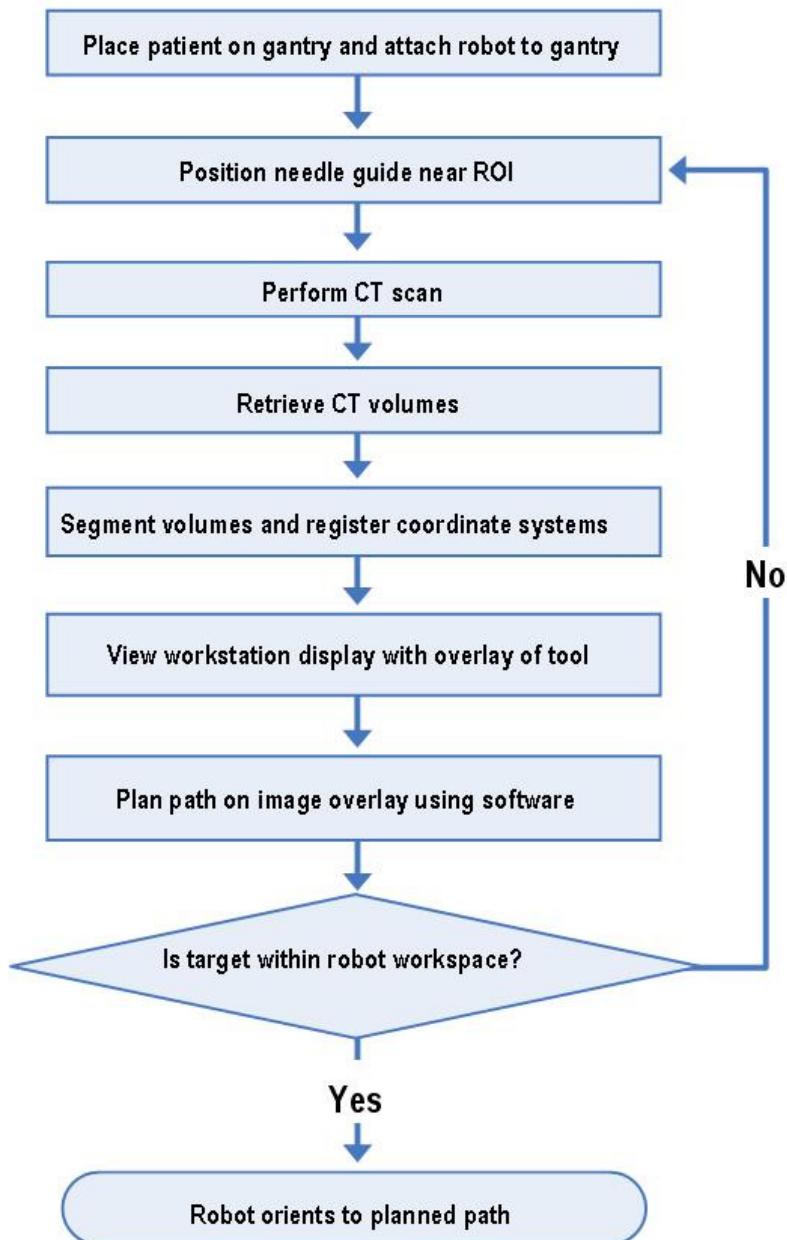


Figure 26.3: Clinical Workflow for Robot Assisted Needle Placement.

The first parameter “ITKImage” is just a string used to concatenate with “Observer” to construct a new observer class. In this case, we will have a observer class named “ITKImageObserver.” This macro also specifies the observer will be observing the “ITKImageModifiedEvent,” and this event will carry an object of type “ITKImageType”. The expansion of this macro can be found in `igstkMacro.h`.

2. Next, you will need to instantiate the observer, and tell the observer to catch the event from the IGSTK image spatial object using the `AddObserver` function.

```
ITKImageObserver::Pointer itkImageObserver = ITKImageObserver::New();
m_ImageSpatialObject->AddObserver(
    ImageSpatialObjectType::ITKImageModifiedEvent(),
    itkImageObserver );
```

3. You will need to call the `RequestGetITKImage` method to cause the IGSTK image spatial object class to send out the event carrying the ITK image.

```
m_ImageSpatialObject->RequestGetITKImage();
```

4. The final step is to check the observer to see if it catches any event. If it does, then we can “Get” the loaded object out of the caught event. `m_ITKImage` is an ITK image pointer.

```
if ( itkImageObserver->GotITKImage() )
{
    m_ITKImage = itkImageObserver->GetITKImage();
}
else
{
    return false;
}
```

5. After getting the ITK image, we can pass it to any ITK filter for further processing. In this example, we use the following filters:

```
itk::BinaryThresholdImageFilter
itk::ConnectedComponentImageFilter
itk::RelabelComponentImageFilter
```

The detailed code can be found in the `Example/DeckOfCardRobot/FiducialSegmentation` class.

26.2.2 Write Your Own Representation Class

If you want to have your own visualization class to get a special rendering effect, you may want to write a new representation class inherited from one of the IGSTK representation classes. Here we give an example of implementing the `igstkImageSpatialObjectVolumeRepresentation` class. The code can be found in the `Example\DeckOfCardRobot` directory.

We can work on changing the code in `igstkCTImageSpatialObjectRepresentation` class. It gives us a good starting point, and we can reuse its frame and old state machine logic since these two classes perform similar tasks except they use different rendering techniques.

1. First, we can make a copy of the code and rename the file and class names to the new class name.
2. Second, we need to include some headers for the volume rendering. Here we use 3D texture mapping; this feature is supported by recent graphics cards only.

```
#include "vtkImageShiftScale.h"
#include "vtkColorTransferFunction.h"
#include "vtkPiecewiseFunction.h"
#include "vtkVolumeTextureMapper3D.h"
#include "vtkVolumeProperty.h"
#include "vtkVolume.h"
```

3. The next thing to do is declare member variables necessary for volume rendering:

<code>vtkPiecewiseFunction *</code>	<code>m_OpacityTransferFunction;</code>
<code>vtkColorTransferFunction *</code>	<code>m_ColorTransferFunction;</code>
<code>vtkImageShiftScale *</code>	<code>m_ShiftScale;</code>
<code>vtkVolumeTextureMapper3D *</code>	<code>m_VolumeMapper;</code>
<code>vtkVolumeProperty *</code>	<code>m_VolumeProperty;</code>
<code>vtkImageData *</code>	<code>m_ImageData;</code>
<code>vtkVolume *</code>	<code>m_ImageActor;</code>
<code>unsigned</code>	<code>m_ShiftBy;</code>
<code>unsigned</code>	<code>m_MinThreshold;</code>
<code>unsigned</code>	<code>m_MaxThreshold;</code>

4. The most important function you need to work on is `CreateActors()`. The `m_ImageData` in the following code is a `vtkImageData` object, which can be obtained from IGSTK image spatial object in a similar fashion as mentioned in Section 26.2.1:

```
igstkLogMacro( DEBUG, "igstk::ImageSpatialObjectRepresentation\\"
```

```
    ::CreateActors called...\n");

//To avoid duplicates we clean the previous actors
this->DeleteActors();

//Create new actor
m_ImageActor = vtkVolume::New();
this->AddActor( m_ImageActor );

//Shift the data to desired range
m_ShiftScale = vtkImageShiftScale::New();
m_ShiftScale->SetInput( mImageData );
m_ShiftScale->SetShift( m_ShiftBy );
m_ShiftScale->SetOutputScalarTypeToUnsignedShort();

//Pass the image data to the volume mapper
m_VolumeMapper = vtkVolumeTextureMapper3D::New();
m_VolumeMapper->SetInput( m_ShiftScale->GetOutput() );

//Create opacity transfer function
m_OpacityTransferFunction = vtkPiecewiseFunction::New();
m_ColorTransferFunction = vtkColorTransferFunction::New();

m_OpacityTransferFunction = vtkPiecewiseFunction::New();
m_OpacityTransferFunction->AddPoint(0, 0.0);
if( m_MinThreshold > 0 )
{
    m_OpacityTransferFunction->AddPoint(m_MinThreshold, 0.05);
}
m_OpacityTransferFunction->AddPoint(m_MaxThreshold, 0.1);
m_OpacityTransferFunction->AddPoint(m_MaxThreshold+1, 0.0);

//Create color transfer function
m_ColorTransferFunction = vtkColorTransferFunction::New();
m_ColorTransferFunction->AddRGBPoint(m_MinThreshold, 0.0, 0.0, 0.0);
m_ColorTransferFunction->AddRGBPoint(m_MaxThreshold/4, 1, 0, 0);
m_ColorTransferFunction->AddRGBPoint(m_MaxThreshold/2, 0, 0, 1);
m_ColorTransferFunction->AddRGBPoint(m_MaxThreshold/4*3, 0, 1, 0);
m_ColorTransferFunction->AddRGBPoint(m_MaxThreshold, 1, 1, 1);

//Pass opacity and color transfer function to volume property
m_VolumeProperty = vtkVolumeProperty::New();
m_VolumeProperty->SetColor(m_ColorTransferFunction);
m_VolumeProperty->SetScalarOpacity(m_OpacityTransferFunction);
```

```
//Push an input to state machine and request it to process it
igstkPushInputMacro( ConnectVTKPipeline );
m_StateMachine.ProcessInputs();
```

This piece of code specifies the color and opacity transfer functions and passes them to a vtkVolumeProperty. It also passes the VTK image data to a vtkVolumeTexture3DMapper. The last two lines of code generate an input ConnectVTKPipeline to the state machine, and call for the state machine to process this input. The state machine will decide whether the representation class is ready to render the image or not.

5. You should also look into the code of the `ConnectVTKPipelineProcessing()` function, which will be called when the representation class is ready to visualize the image.

```
m_ImageActor->SetMapper(m_VolumeMapper);
m_ImageActor->SetProperty(m_VolumeProperty);
m_ImageActor->SetVisibility( 1 );
m_ImageActor->SetPickable( 0 );
```

This passes the volume and volume property to an ITK actor for rendering.

26.2.3 Using the Socket Communication Class

The communication between the application and robot server is through the TCP/IP protocol. For this purpose, the socket communication component of IGSTK is used. You can write your own command interpreter class for your specific hardware. Here we give a simple example on how to implement the robot control class. The detailed code can be found in the `Example/DeckOfCardRobot/RobotCommunication` class.

1. First, create a client object:

```
typedef igstk::SocketCommunication           SocketCommunicationType;
SocketCommunicationPointerType               m_Client;
m_Client = SocketCommunicationType::New();
```

2. Second, initialize the socket communication and connect to the host and port:

```
m_Client->RequestOpenCommunication()
m_Client->RequestConnect( IPADDRESS, PORT)
```

3. For this particular robot, we need to login first:

```
m_Client->RequestWrite( "@AUTH;Team;A\r\n" );
m_Client->RequestRead( buffer, 100, num, READ_TIMEOUT );
```

4. Before operating the robot, we should “home” the robot first. This is a self-calibration method.

```
snprintf( sendmessage, ROBOT_MAX_COMMAND_SIZE, "@HOME;%d;%d\r\n",
           TRIGGER_IMMEDIATE, WRITE_TIMEOUT );
m_Client->RequestWrite( sendmessage );
m_Client->RequestRead( buffer, 100, num, READ_TIMEOUT );
```

5. Now we can command the robot to move to certain translations (X,Y, and Z) with certain rotations (A,B, and C):

```
snprintf( sendmessage, ROBOT_MAX_COMMAND_SIZE,
           "@MAW;%d;%d;%d;%f;%f;%f;%f;%f\r\n", TRIGGER_IMMEDIATE,
           INTERRUPT_IMMEDIATE, WRITE_TIMEOUT, X, Y, Z, A, B, C );
m_Client->RequestWrite( sendmessage );
m_Client->RequestRead(buffer, 100, num, READ_TIMEOUT);
```

6. When we are done, we log out of the robot and close the communication.

```
m_Client->RequestWrite("@QUIT\r\n");
m_Client->RequestCloseCommunication()
```

26.3 Result

Figure 26.4 shows the user interface of the application with the control panel on the left, and three standard 2D slice views and a 3D volume rendering on the right. The yellow cylinder is the needle holder, and the purple square indicates the robot’s working region. A path is being planned to target the tumor while avoiding the ribs. The figure shows the robot aligning with the planned path.

Figure 26.5 shows a TeraRecon rendered image of the robot needle holder.

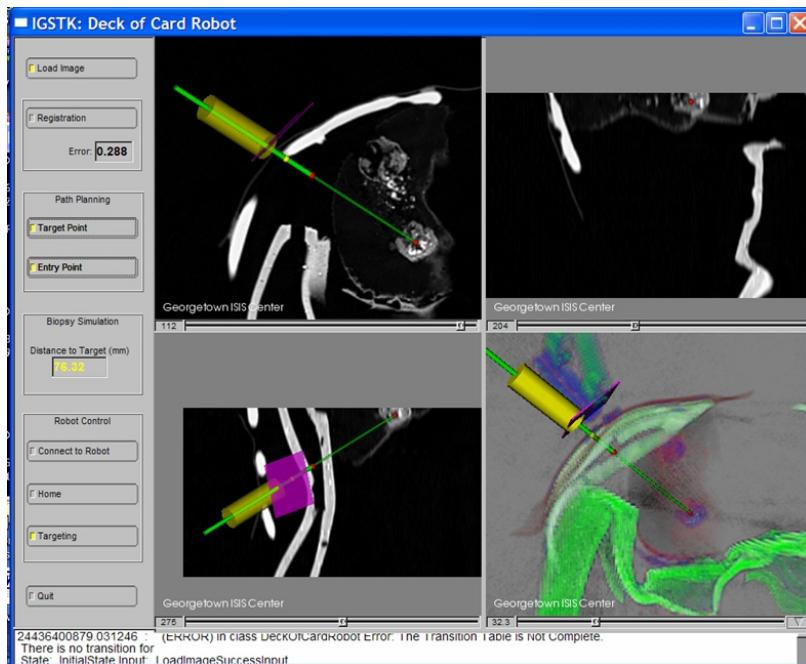


Figure 26.4: User Interface for the Robot Application.

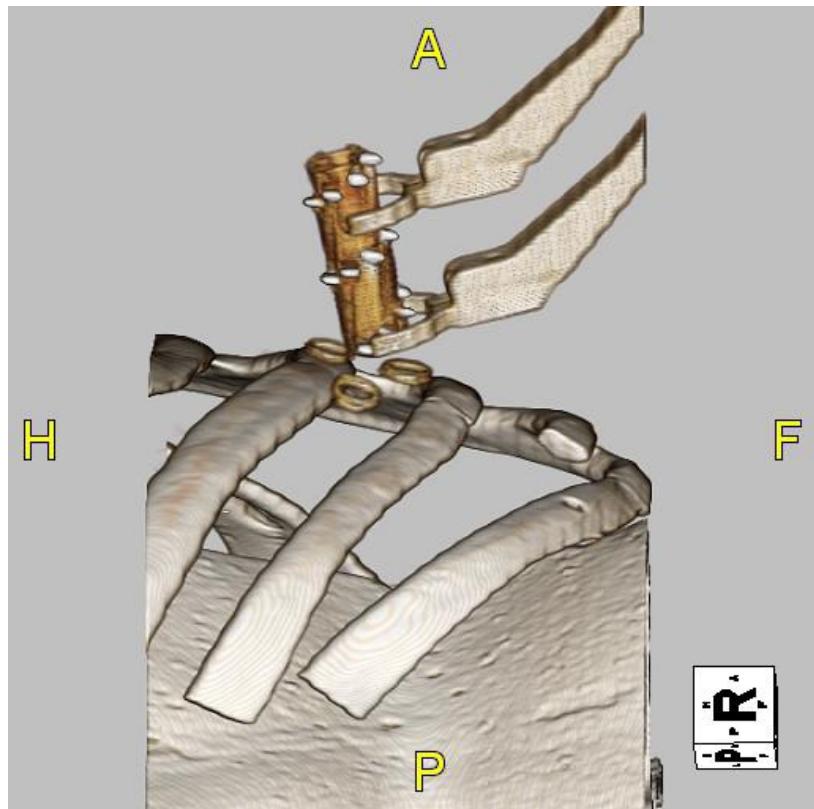


Figure 26.5: 3D Rendering of CT Scan Showing Spiral Fiducial Pattern for Registration of Robot to CT Images.

Part VII

Appendices

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 the NLM Insight Registration and Segmentation Toolkit (ITK) with minor modifications where needed.

A.2 Implementation Framework

A.2.1 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.2.2 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 an 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 inadvertent castings, which may result in run-time errors.

A.2.3 Portability

IGSTK is designed to compile on a set of target operating system/compiler combinations. Whatever the list of compilers we select to support, we must make sure that we set up nightly builds for all of them. The defacto definition of a supported compiler is a *compiler that is submitting a green nightly build to the Dashboard*. The list of supported compilers can be found in Chapter 2.

A.2.4 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 and native build tools for a particular operating system/compiler combinations. (www.cmake.org).

A.2.5 Doxygen Documentation System

The Doxygen open-source system is used to generate online documentation. Doxygen requires the embedding of simple comments in the code, which are in turn extracted and formatted into documentation. (<http://www.stack.nl/dimitri/doxygen>). It is important for developers to be 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, and including links to URLs.

A.2.6 vnl Math Library

IGSTK uses vnl Math Library for math-related functions. These libraries 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. Never expose vnl at the IGSTK API level (in the same way that we should avoid exposing ITK or VTK).

A.2.7 Reference Counting and SmartPointers

IGSTK has adopted reference counting via smart pointers to manage object references. Smart pointers automatically increase and decrease 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.2.8 CVS Environment

CVS is used for the version control, software updates, and downloads, although we may move to Subversion in the future.

A.2.9 CDash Dashboard Testing Environment

IGSTK intends to have testing for 100 percent of its code. This is achieved by writing test code which uses every line of the IGSTK code. Test code should preferably be written before the main code is written or along with the main code. The CDash/Dashboard would be used as the testing environment. The notion of 100 percent testing is not limited to 100 percent coverage. It goes beyond 100 percent coverage. Coverage only indicates if a line of code has been executed as part of a test. In the case of IGSTK we must test all possible combinations of function calls to ensure the robustness of the toolkit against inappropriate usage. The use of state machines is fundamental to achieve this goal. The stringent testing must be enforced if we expect IGSTK to be used in a surgery room.

A.3 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: 2009-05-27 19:14:31 $
Version:   $Revision: 1.18 $

Copyright (c) ISC Insight Software Consortium. All rights reserved.
See IGSTKCopyright.txt or http://www.igstk.org/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 : 2009 – 05 – 27 19 : 14 : 31*, *Revision : 1.18*. These should be used in each source file under CVS control.

The copyright reads as follows:

```
/*=====
```

Copyright (c) 1999-2007 Insight Software Consortium. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * The name of Insight Software Consortium, 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.4 File Organization

Classes are created and usually organized into a single class per file set. A file set consists of a .h header file, a .cxx implementation file, and/or a .txx templated implementation file. Helper classes may also be defined in the file set, and 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, *.ttx and *.cxx code of the example (application) code}
Utilities
{all *.h, *.ttx and *.cxx code of the third-party libraries}
Testing
{all *.h, *.ttx and *.cxx code of the test code}
Data
Baseline
{all sample result data for comparing}
Input
{ all input data for the test code }
```

A.5 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 functions.

A.6 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, and the use of abbreviations is discouraged. (Abbreviations are allowed when they are commonly used, 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 are explained in the following subsections.

A.6.1 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 reads 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, so please, review existing names to catch the spirit of the naming convention. Example names include: DicomReader, TrackerInterface, DefaultImage-Traits.

A.6.2 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.6.3 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 being 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 the exact method, where it originates, and where it is being invoked. Similarly the global namespace (:) should be used when referring to a global function.

A.6.4 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 referring to a member variable, an explicit *this* pointer should be used. Derived classes must use the combination of *this* and Get/Set methods.

A.6.5 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 variables.

A.6.6 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.6.7 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 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, with the 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.6.8 Using Underscores

Do not use underscores. 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.6.9 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.7 Const Correctness

Const correctness is important. Please use it as appropriately for 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 has some flexible spots. IGSTK must cover these eventual const-correctness failures and enforce complete const-correctness verification.

A.8 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.8.1 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, which are 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 it will wreak havoc in others' editors.

The declaration of variables within classes, methods, and functions should be one declaration per line.

```
int i;  
int j;  
char* stringname;
```

A.8.2 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.8.3 Method Definition

Methods are defined across multiple lines. This procedure 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. The fourth line is the name of the method.

A.8.4 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.8.5 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.9 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 then 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.9.1 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.9.2 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 written only 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 to ensure that it is still applicable to the new modified method.

A.10 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.11 Exception Handling

C++ exceptions will not be used because they make it difficult to guarantee that the state of the classes is valid after recovering from an exception.

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.12 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 they 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.13 Programming Practices

A.13.1 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.13.2 Choice of signed or unsigned

Be careful for signed/unsigned mismatch warnings. In general, for *for* loops, you want to use *unsigned int*.

Glossary

API	Application Programming Interface
BSD	Berkeley Software Distribution
CDash	An open-source, distributed, software quality system http://www.CDash.org
CMake	A cross-platform make system http://www.cmake.org
CT	Computed Tomography
CVS	Concurrent Version System: a source code control system
DICOM	Digital Imaging and Communications in Medicine
DSM	Deterministic State Machine
FLTK	Fast Light ToolKit: a GUI toolkit http://www.fltk.org
FSM	Finite State Machine
GUI	Graphical User Interface
I/O	Input / Output
IGS	Image-Guided Surgery
IGSTK	Image-Guided Surgery Toolkit http://www.igstk.org

ITK	Insight segmentation and registration ToolKit http://www.itk.org
LTSA	Labeled Transition Systems Analyzer
MFC	Microsoft Foundation Classes
MRI	Magnetic Resonance Imaging
NDI	Northern Digital Incorporated: vendor of tracking devices
NLM	National Library of Medicine
PNG	Portable Network Graphics
RFA	Radiofrequency Ablation
Qt	A GUI software toolkit from Trolltech Inc.
VTK	Visualization ToolKit http://www.vtk.org
Wiki	A website that allows people to easily change the content
XML	Extensible Markup Language

BIBLIOGRAPHY

- [1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal a tool suite for automatic verification of real-time systems. In *Proceedings of Hybrid Systems III*, pages 232–243. Springer-Verlag, 1996. 20.2
- [2] Eric Boix, Mathieu Malaterre, Benoit Regrain, and Jean-Pierre Roux. *The GDCM Library*. CNRS, INSERM, INSA Lyon, UCB Lyon, <http://www-creatis.insa-lyon.fr/Public/Gdcm/>. 14.1
- [3] A. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. Wiley Interscience, 2002. 5.1, 6.2
- [4] R. Kikinis D. T. Gering, A. Nabavi and N. Hata. An integrated visualization system for surgical planning and guidance using image fusion and an open mr. *J Magn Reson Imaging*, 13(967), 2001. 19.2.2
- [5] B. P. Douglas. *Real-Time Design Patterns: Robust Scalable Architecture for RealTime Systems*. Addison-Wesley Professional, 2002. 5.1, 6.1, 6.2.7, 7.1
- [6] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), 1976. 3.2.2
- [7] Tokuda J Fischer GS, Iordachita I and Hata N. MRI-compatible pneumatic robot for transperineal prostate needle placement. *IEEE/ASME Trans Mechatronics*, 13(3), 2008. 19
- [8] J. Michael Fitzpatrick. Fiducial registration error and target registration error are uncorrelated. volume 7261. SPIE, 2009. 23.6
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995. 7.1
- [10] K. Gary, L. Ibanez, S. Aylward, D. Gobbi, B. Blake, and K. Cleary. IGSTK: An open source software toolkit for image-guided surgery. *IEEE, Computer*, 39(4), 2006. 1

- [11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. 20.3.1, 20.3.4
- [12] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003. 20.2
- [13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001. 6.2
- [14] B. K. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4:629–642, April 1987. 15.1
- [15] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, <http://www.itk.org/ItkSoftwareGuide.pdf>, 2003. 3.2.6
- [16] A. Knapp, S. Merz, and C. Rauh. Model checking timed uml state machines and collaborations. In W. Damm and E.R. Olderog, editors, *7th Intl. Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002. 20.2
- [17] L. Kohn, J. Corrigan, and M. Donaldson, editors. *To Err is Human: Building a safer health system*. National Academy Press, 2001. 5.2
- [18] D. C. Kozen. *Automata and Computability*. Springer, 1997. 6.1
- [19] Jon T. Lea, Dane Watkins, Aaron Mills, Michael A. Peshkin, Thomas C. Kienzle, and David Stulberg. Registration and immobilization in robot-assisted surgery. *J. Image Guid. Surg.*, 1(2):80–87, 1995. 16
- [20] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*, 2nd edition. J. Wiley and Sons, 2006. 20.2, 20.4.3
- [21] R. C. Martin. *Clean Code, A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. 5.1, 5.4
- [22] Calvin R. Maurer, J. Michael Fitzpatrick, Matthew Y. Wang, Robert L. Galloway, Robert J. Maciunas, and George S. Allen. Registration of head volume images using implantable fiducial markers. *IEEE Transaction on Medical Imaging*, 16(3):447–462, 1997. 16
- [23] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. 8.6
- [24] J.W. Spence. There has to be a better way! [software development]. In *Proceedings of the Agile Conference*, July 2005. 20.1
- [25] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2009. 16.2
- [26] A.H. Watson and T.J. McCabe. Structured testing: a testing methodology using the cyclomatic complexity metric. Technical Report 235, NIST, 1996. 20.4.3, 5
- [27] 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. 15.2, 15.2.3

- [28] K.E. Wiegers. *Peer Reviews in Software*. Addison-Wesley, 2002. 3.2.2
- [29] L. Wingerd and C. Seiwald. High-level best practices in software configuration management. Technical report, Proceedings of the Eighth International Symposium on Systems Configuration Management, 1998. 3.2.4

INDEX

- agile development, 24
- Alan Turing, 71
- Algorithmic Complexity
 - State Machine, 75
- Architecture
 - Timing, 63
- best practices, 23
- Calibration, 225
 - Pivot Calibration, 227
- CDash, 24, 33
- CMake, 11, 24
 - downloading, 11
- Code Coverage
 - Error management, 77
 - If conditionals, 77
 - State Machines and Testing, 76
- code coverage, 24, 33
- code review, 26
- codeline, 24
- Communication, 134
- communication
 - blocking, 135
- Complexity
 - State Machine, 75
- Components, 53
 - Calibration, 61
 - Display, 55
 - GeometricRepresentation, 56
 - Infrastructure, 61
 - Readers, 60
 - Services, 62
- Tracking, 59
- VisualRepresentation, 58
- Contract-Based Programming, 98
- Coordinate Systems, 107
 - API, 112
 - Events, 114
 - Examples, 118
 - Macros, 113
 - Rendering, 115
 - Scene Graph, 109
 - Transform, 108
 - Transform and Timestamps, 111
 - Transform Computation, 110
- CVS, 28
 - dashboard, 33
 - defect tracking, 36
- Determinism
 - State Machine, 71
- dot, 116
- Doxygen, 24, 25
 - Entscheidungsproblem, 71
- Error Handling
 - Events, 93
- Error prediction, 220
- Events
 - API Containment, 92
 - Background, 91
 - Class Decoupling, 92
 - Code Reuse, 92
 - Definition Macros, 95
 - Error Handling, 93

Event to Input Transduction, 96
Hierarchy, 94
IGSTKEvent base class, 94
Implementation, 93
Inheritance from ITK, 93
Motivation, 91
Observers, 96
Payload Events, 94
Relationship with State Machines, 95
Request-Observe Pattern, 98
Safely Returning Data, 98
Usage, 97
Use in other toolkits, 91

FriendClassMacro
 State Machine, 86

GraphViz, 116
Graphviz
 State Machine Diagram, 77

IGSTK
 Downloading, 5
 downloading the development release, 6
 downloading the stable release, 6
 mailing list, 7
igstk::AxesObject, 120, 123, 127, 159
igstk::BoxObject, 161
igstk::BoxObjectRepresentation, 173
igstk::ConeObject, 161
igstk::CTImageSpatialObject, 163
igstk::CylinderObject, 162
igstk::EllipsoidObject, 162
igstk::MeshObject, 165
igstk::ReadVascularNetworkObject, 170
igstk::SpatialObjectHierarchy, 155
igstk::TubeObject, 166
igstk::VascularNetworkObject, 168
Image I/O, 207
 CT, 208
 Gantry tilt, 210
 GDCM, 208
 MRI, 208
 Screenshot, 212
 Ultrasound, 208
Installation, 9

Configuration, 13
KWStyle, 24, 26, 33
Landmark-based registration, 213
Layered Architecture, 52
logging, 193
 BaseClasses, 196
 Extending LogOutput, 201
 FLTKTextBufferLogOutput, 199
 FLTKTextLogOutput, 200
 flushing, 194
 itk::OutputWindow, 201
 Logger, 193, 196
 LoggerBase, 196
 LoggerManager, 196
 LoggerThreadWrapper, 202
 LogOutput, 198
 Multi-threaded Logging, 202
 MultipleLogOutput, 200
 OutputProperties, 194
 Overriding vtkOutputWindow, 202
 PriorityLevel, 194
 Redirecting ITK, VTK logs to Logger, 201
 StdStreamLogOutput, 198
 ThreadLogger, 203

LTSA
 State Machine Diagram, 77

mailing list, 7, 23, 27
Medical Errors, 51

Observers
 Events, 96

open source development, 27

OpenIGTLLink, 257
 Example, 262
 Library, 261
 Protocol, 258
 Data Body, 260
 Data Header, 259

UserCase, 257

Pulse Generator
 Implementation, 68

- Reachability
 - State Machine, 78
- Registration, 213
- release cycle, 30
- Reslicing, 231
 - API, 234
 - automatic mode, 233
 - Examples, 237
 - Hierarchy, 231
 - manual mode, 233
 - Oblique, 233
 - OffOrthogonal, 233
 - Orthogonal, 233
 - State Machine, 234
- Safety by design, 50
- sandbox, 30
- Scene Graph, 107, 291
 - Visualization, 115
- SCXML
 - State Machine Diagram, 77
- SharedObjectRepresentation, 182
- source code control, 28
- Spatial Objects, 155
- State Machine, 71
 - Alan Turing, 71
 - API, 79
 - Architecture Motivation, 49
 - Background, 71
 - Characteristics, 72
 - Code Complexity, 75
 - Determinism, 71
 - Deterministic Behavior, 72
 - Documentation, 77
 - Entscheidungsproblem, 71
 - Exporting its description, 82
 - Exporting to Graphviz, 77
 - Exporting to LTSA, 77
 - Exporting to SCXML, 77
 - Facilitating testing, 76
 - friend of the class, 85
 - Helper Macros, 80
 - Implementation, 78
 - Instantiation, 86
 - Integration with IGSTK class, 85
- Loops exclusion, 78
- Motivation to use it, 72
- owner class member variables, 86
- PreclueWrongUse Wrong Use, 73
- Private interface, 73
- Processing methods, 73
- Programming, 79
- Public interface, 73
- Reachability, 78
- Request methods, 73
- Request/Processing pairs, 88
- Robustness to misuse, 74
- Testing and Code Coverage, 76
- this pointer, 87
- Traceability, 76
 - Traits, 85
 - Versus latent logic, 75
- StateMachine
 - FriendClassMacro, 86
- stereotaxis, 133
- Timing, 63
 - Collaborations, 64
 - Fine tuning, 68
- Traceability
 - State Machine, 76
- Tracker, 133
- TrackerTool, 134
- tracking, 133
 - buffer, 137
 - communication, 134
 - localizer, 133
 - position measurement system, 133
 - PulseGenerator, 135
 - reference, 138
 - simulation, 146
 - state machine, 141
 - tracking system, 133
- Transform, 111
- unit test, 24, 33
- VideoImager, 241
 - API, 245
 - Buffering, 243
 - Events, 247

Frames and Timestamps, 244
Hierarchy, 244
State Machine, 245
Threading, 243
Tool, 249
View, 185
 GUI, 185
 Pulse Generator, 185

wiki, 23, 26, 27