IGSTK: Framework and Example Application Using an Open Source Toolkit for Image-Guided Surgery Applications

Peng Cheng[a]*, Hui Zhang[a], Hee-su Kim[b], Kevin Gary[c], M. Brian Blake[d], David Gobbi[e], Stephen Aylward[f], Julien Jomier[f], Andinet Enquobahrie[f], Rick Avila[f], Luis Ibanez[f], Kevin Cleary[a]
[a]Imaging Science and Information Systems (ISIS) Center, Dept. of Radiology, Georgetown University, Washington, DC, USA
[b]Department of Computer Science, Kyungpook National University, Korea
[c]Division of Computing Studies, Arizona State University, Mesa, Arizona, 85212, USA
[d]Department of Computer Science, Georgetown University, Washington, DC, 20007, USA
[e]Atamai Inc., London, Ontario, N6B 2R4, Canada
[f]Kitware Inc., Clifton Park, NY, 12065, USA

ABSTRACT

Open source software has tremendous potential for improving the productivity of research labs and enabling the development of new medical applications. The Image-Guided Surgery Toolkit (IGSTK) is an open source software toolkit based on ITK, VTK, and FLTK, and uses the cross-platform tools CMAKE and DART to support common operating systems such as Linux, Windows, and MacOS. IGSTK integrates the basic components needed in surgical guidance applications and provides a common platform for fast prototyping and development of robust image-guided applications. This paper gives an overview of the IGSTK framework and current status of development followed by an example needle biopsy application to demonstrate how to develop an image-guided application using this toolkit.

Keywords: open source, image-guided surgery, surgical guidance, tracking, state machine, needle biopsy, application prototyping

1. PURPOSE

The field of image-guided surgery is rapidly expanding, as new techniques are being developed that minimize treatment invasiveness and thereby reduce trauma and speed recovery for patients. A typical image-guided system consists of tracking subsystems for capturing the movement of instruments and the patient, and a computer subsystem for integrating and displaying pre- and intra-operative images and instrument positions using the tracking data. Most of the development effort is focused on implementing a general software infrastructure that supports the basic capabilities and inter-operation of these components. As new trackers and data integration algorithms are developed, maintaining a general infrastructure that nevertheless allows new features to be exploited is even more challenging, as most software platforms are not designed to be extendable from the outset. Testing such systems can also be an extensive process, and the lack of proper testing expose risks to patients and surgeons. The purpose of IGSTK is to enable the creation and validation of reusable, robust image-guided software as open-source, free of commercial licensing restrictions. This component-based software toolkit should make it easy for researchers to prototype and develop image-guided surgery applications.

2. METHODS

IGSTK has been under development for over a year and a public beta release will be available by the time of this conference (February 2006). IGSTK is based on the following existing open source software toolkits: ITK for

* peng@isis.georgetown.edu; Telephone 1-202-687-2902; Fax 1-202-784-3479

segmentation and registration [1], VTK for visualization [2], CMake for cross-platform building and DART for cross-platform testing, and FLTK for the user interface [3] (though the user interface layer is designed to allow for future application development with other UI toolkits). The IGSTK toolkit contains the basic software components needed to construct an image-guided system, including support for trackers, data integration algorithms, and a four-quadrant view incorporating image overlay.

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 is designed based on the following principles [4]:

1. Requirements are generated by studying and documenting scenarios of the surgical procedures in a clinical environment. By analyzing these scenarios, features that are necessary for fulfilling the requirements are extracted and implemented.
2. All IGSTK components are governed by a state machine. A state machine is contained within the class to control the access to the class. Components are always in a valid state to ensure they will perform in a predictable manner. The use of a state machine also helps enforce high quality standards for code coverage and run-time validation.
3. IGSTK does not directly expose ITK and VTK objects APIs to the application developer. Instead, ITK and VTK objects are tightly encapsulated by IGSTK objects that provide a minimal but sufficient API for application development. This ensures that any APIs used by developers have been specifically tested for image guidance applications.
4. IGSTK does not utilize dynamic object typing. When IGSTK objects are created, they must be declared with full specialization, and all type checking is performed at compile time and never at run time. This reduces the number of software parameters that can be adjusted at run-time, and hence reduces the possibility that the software will be running in an untested configuration.
5. IGSTK uses events to communicate between components, where the events are produced by the state machine of one object and received as inputs to the state machine of one (or more) other objects. The mapping of events to state machine inputs is explicitly defined and deterministic.
6. IGSTK does not use a return value or exception at the application level. Instead, it uses events to pass information around.

2.1 State machine

To ensure safety and robustness, the state machine design pattern is incorporated into the IGSTK components. A state machine is defined by a set of states, a set of inputs, and a set of directed transitions between states. Transitions are changes from one state to another when a certain stimulus or input is received. An action may be taken along with a transition, to perform some task on entry to a state, or on exit from a state. State machines support the component-based architecture through encapsulation and extensibility. They ensure that a component is always in a valid state which can guarantee repeatable and deterministic behavior. A description of the state machine theory can be found in [6].

The following points summarize the advantage of using state machine in the toolkit [7]:

1. *Safety and Reliability*: A state machine ensures that component behavior is deterministic and that all components are in a known and error-free state at any given moment.
2. *Cleaner design*: Since developers must anticipate all possible inputs, states, and transitions, the state machine encourages and enforces a cleaner and more robust design, free of untested assumptions.
3. *API simplicity:* A focused, clearly expressed application programming interface (API) is a must for supporting robustness and reliability. In the context of surgical guidance, we believe that *flexibility* and *abundance of features* are undesirable, because they create more opportunities for things to go wrong during a surgical intervention.
4. *A consistent integration pattern*: The toolkit's value as it matures will undoubtedly be tied to the incorporation of additional functionality at the component level. This functionality will often take the form of reusable code from existing toolkits. State machines provide a consistent pattern for integrating this functionality while adhering to the safety-first principles necessitated by the application domain.
5. *Quality Control*: State machines facilitate code coverage in the sense of lines of code tested, as well as path coverage on a per component basis. Using code that is not based on state machines may result in applications

that exhibit unreliable behavior. At run time, they can easily enter into any number of untold states that were never explored by the developer, and can lead to error conditions that may or may not become visible to the users of the application.

The safety emphasis in IGSTK encourages explicit knowledge of component state and determines whether a given behavior may be executed while in that state. Hence, our application is a combination of traditional concepts in reliability (being in a known state) and the State Pattern [8] (for managing availability of behavior dynamically).

2.2 Architecture and components

Figure 1 presents a UML collaboration diagram of the major IGSTK components involved in a typical image-guided surgery application. The key components are from left to right [4, 5]:

1. View classes are used for presenting results and displays to the physician. An internal pulse generator is used to update the scene at specified refresh rate to keep the visual information up-to-date. Currently the IGSTK View classes are built with VTK classes and an FLTK window. They take FL events and translate them into VTK events to enable user interaction within the render window. View2D and View3D are subclasses of View with a different interaction style.
2. SpatialObject and SpatialObjectRepresentation. SpatialObjects classes model physical objects including image data and simple geometrical shapes such as anatomical structures and surgical devices, while SpatialObjectRepresentations classes are responsible for displaying their associated SpatialObject in the scene. IGSTK Spatial Objects can be attached to a Tracker Tool object, so that it can update its position as the Tracker Tool moves. The state machine in SpatialObjects classes enforces this association to be a one time operation. Once a Spatial Object is attached to a tracker tool it is not expected to get back to manual control nor to be re-attached to a second tracker tool. It is no longer possible to change the position of the object programmatically. IGSTK has a variety of SpatialObjects and corresponding SpatialObjectsRepresentations, including basic shapes (Cone, Cylinder, Ellipsoid and so on), ImageSpatialObject, UltrasoundProbeObject, and GroupObject to make composite SpatialObjects.
3. Tracker classes are an abstraction of tracking devices. They communicate with hardware and store data including position, orientation, and other relevant information pertaining to tracked surgical tools. The IGTSK Tracker component is developed based on code donated by Atamai Inc. and wrapped with a state machine. The internals of the Tracker require a separate thread to communicate with tracking hardware at an acceptable frequency. A pulse generator inside the class is used to automatically update the location information of the tracked device with an expiration time and propagate changes to SpatialObjects. IGSTK now supports NDI AURORA, POLARIS, and VICRA trackers (Northern Digital Inc., Waterloo, Canada) and Flock of Birds tracker (Ascension Technology Corporation, Burlington, VT USA), although not all features of these trackers are supported.

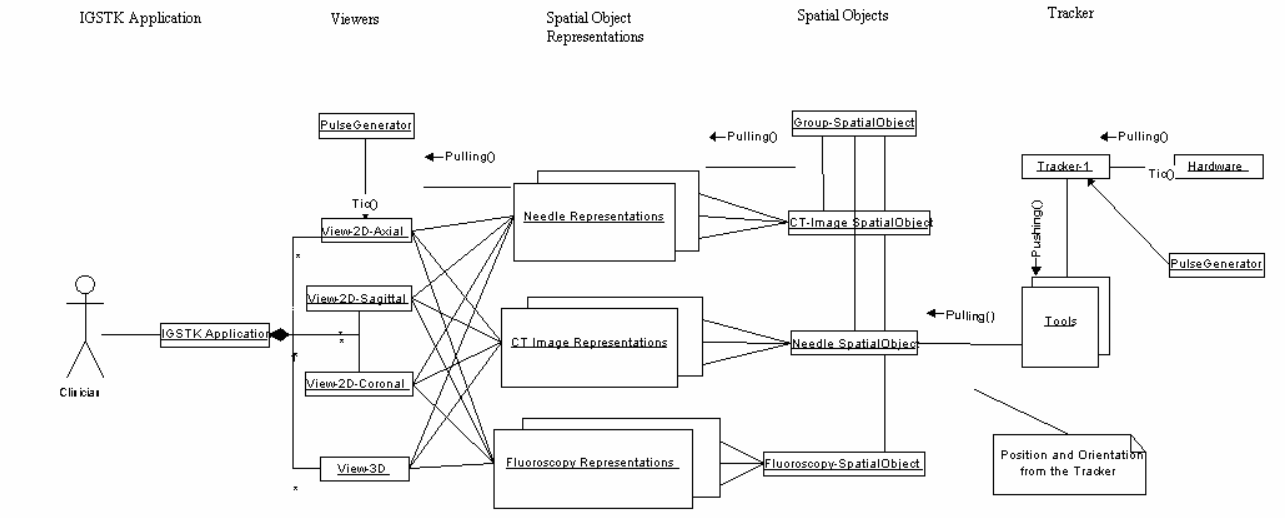

Figure 1: IGSTK Component Architecture

Some other components do not present in Figure 1 include: DICOMImageReader for reading in DICOM images, Landmark3DRegistration for 3D pair-point-based rigid body registration, Logger for logging debug or other information, and Calibration class for calibrating tracker tools. All these together provide essential functionalities to build an image-guided system.

3. EXAMPLE APPLICATION

This section describes the procedure to develop an image-guided biopsy application using IGSTK. A typical image-guided needle biopsy procedure involves first acquiring a pre-operative CT image and then registering the CT image to the patient coordinate system. For this purpose, fiducial-based rigid body registration techniques are commonly used. During the biopsy phase, the needle is tracked by an optical tracking device with real-time visualization of its location overlaid on top of the CT image. This overlay image provides guidance to the surgeon for better targeting of the needle to its desired location. In this example, state machine is being implemented at the application level. While this is not mandatory, it is strongly recommended when using IGSTK. A state machine architecture gives the application developer an easier way to prototype the application and to control the work flow of the surgical procedure, and also adds an extra layer of security to the application to make it more robust. The following sections demonstrate how to write an application using the IGSTK framework.

3.1 Prototyping

The first step is to analyze the surgical application and develop a minimal specification. By analyzing a typical needle biopsy procedure, we identify the following essential tasks in this procedure [9].

1. Obtain the patient demographic information (name, etc.).
2. Load in the pre-operative CT image using the DICOM file format. Fiducials (small markers) are usually placed on the anatomy prior to the scan for landmark based registration in Steps 4-7.
3. Verify the patient information against the information in the image. Prompt the surgeon if there is a discrepancy. This step is typical of the error checking that should be done and one should assume that if anything can go wrong it will go wrong and safeguards should be provided.
4. Identify the image landmarks by going through the CT image slices and selecting the fiducials using the mouse. For paired-point based registration, at least three points are required although at least four are preferred.
5. Initialize the tracking device.
6. Add patient landmarks by touching the physical fiducials attached to patient using the tracked pointer device.
7. Perform the image to patient landmark registration.
8. Path planning. The surgeon will select a target point and an entry point to plan the path for the needle puncture.
9. Provide a real-time display of the overlay of the needle probe and pre-operative images in the quadrant viewer window during the biopsy procedure.

Since the application is implemented 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 “RequestSetPatientNameInput” to the state machine. The state machine will take this input and change its current state from “InitialState” to “WaitingForPatientNameState”, and the action is to pop up a window asking for input of the patient name. If the user inputs a valid name, then there will be a “PatientNameInput” which brings the state machine into “PatientNameReadyState”, otherwise there will be a “PatientNameEmptyInput”, which will return the state machine to the “InitialState”. Thus, we can map the application into series of states and inputs and this higher level abstraction will help the developer design a clear work flow for the application. Figure 2 shows the state machine diagram for the needle biopsy application which was generated automatically when the state machine is constructed using the ‘dot’ tool from Graphviz [10].

Figure 2: State machine diagram for needle biopsy application. (Circle for state and arrow for transition and corresponding input)

3.2 Coding the state machine

This section shows how to code the state machine into the needle biopsy application. IGSTK has a number of convenient macros to facilitate the programming of the state machine. The details of these macros can be found in Source/igstkMacro.h. More information on the state machine design pattern and guidelines can be found on the IGSTK wiki page under "Development" section on "Design discussions" page [12]. Note that these URLs are current as of the time of publication (Feb. 2006) but are subject to change in the future.

Once we have the higher level abstraction of the application and prototyped it in the state machine model, we need to take the following steps to program the state machine into the applications.

1. The first step is to use the state machine declaration macro in your class's header file. This macro defines types for state and input, creates a private member variable m_StateMachine, and two private member functions for exporting the state machine description into dot format for the state machine diagram visualization [10] and LTSA (Labeled Transition Systems Analyzer) format for state machine animation and validation [11].

```
igstkStateMachineMacro();
```

2. Then take the states and inputs mapped out during the prototyping stage and define them in the header file using the following macros. To enforce the naming conventions of the state machine, the declaration macros will append "State" or "Input" automatically after the variable name. For instance, the following two lines will define m_InitialState and m_RequestLoadImageInput.

```
igstkDeclareStateMacro( Initial );
igstkDeclareInputMacro( RequestLoadImage );
```

3. The next step is to construct the state machine in the constructor of the source file. First, we need to add all the states and inputs declared in the header into the state machine.

```
igstkAddStateMacro( Initial );
igstkAddInputMacro( RequestLoadImage );
```

4. The next step is a crucial step which creates the state machine transition table to control the logic and workflow of the application. This is done using the macro igstkAddTransitionMacro(From_State, Received_Input, To_State, Action). This means when the state machine is in the "From_State" and receives the "Received_Input", it will enter into the "To_State" and evoke the ActionProcessing() as an action for this transition. This macro requires the "ActionProcessing" method to be pre-defined in the class for the state machine to call. For example:

```
igstkAddTransitionMacro( Initial, RequestSetPatientName, WaitingForPatientName, SetPatientName );
```

In this case, we need to have a SetPatientNameProcessing() method defined in the class for this code to compile.

5. After we have setup the transition table, the next step is to select an initial state, and flag the state machine to be ready to run. After the state machine is ready to run, we cannot change the state machine transition table in the code. This is designed this way to enhance the safety of the state machine and prevent accidentally changes to the state machine behavior in the code.

```
igstkSetInitialStateMacro( Initial );
m_StateMachine.SetReadyToRun();
```

6. Now the state machine is setup and ready to run. We can then export the state machine description in dot format and generate the graphical visualization as shown in Figure 2. This graph will help us to examine the workflow of the application and the state transition table.

```
std::ofstream ofile;
ofile.open("DemoApplicationStateMachineDiagram.dot");
const bool skipLoops = false;
this->ExportStateMachineDescription( ofile, skipLoops );
ofile.close();
```

This will output the state machine into a dot file when we execute the application. If you have the dot tool installed in your system, then you can run the following command, which will take the dot file and generate a png format picture named "SMDiagram.png" for the state machine.

>dot –T png –o SMDiagram.png DemoApplicationStateMachineDiagram.dot

7. All the requests to a state machine should be translated into inputs and the state machine will response to those inputs depending on its current state. These actual actions should be protected methods and only called by the state machine directly. In the code, a click on the load image button will be translated to a 'RequestLoadImageInput', and then we call ProcessInputs() to let the state machine handle this request.

```
igstkPushInputMacro( RequestLoadImage );
m_StateMachine.ProcessInputs();
```

If the state machine is in the right state to load the image, a protected method associated with this transition (eg. LoadImageProccessing()) will be evoked by the state machine as defined in the transition table constructed in the constructor.

3.3 Discussion

From the computational theory point of view, all computers are state machines, and all computer programs are state machines regardless of whether the developers used the state machine programming pattern or not. Traditional programming approaches represent the states ambiguously by using a large number of variables and flags, which result in many conditional tests in the code (if-then-else or switchcase statements in C/C++). Programmers could neglect to consider all possible paths in the code while struggling with if-else conditional tests and flag checks. These practices may result in unpredictable behavior and limit safety in the design of the underlying applications. Since predictability is critical for mission critical applications running in the surgery room, this approach is not suitable for our purpose.

In comparison to traditional approaches, state machines will reduce the number of paths in the code, save the developers from convoluted conditional tests, and encourage them to focus on higher level design. From the above example, we can see that the state machine is easy to program and manage under the IGSTK framework. We encourage developers to design and code the state machine of their application first, and then generate the state machine diagram as shown in Figure 2. They can go through the diagram, examine and verify their design of the work flow. If they want to add or change a path of the application, it is just a matter of adding or deleting a transition table entry. This eliminates the level of difficulty required for going through the code and struggling with if-then-else logic. This will largely facilitate the application prototyping, and the implementation code can be plugged into the skeleton program later. These techniques should result in clearer designs and safer applications.

Figure 3 shows the user interface of the needle biopsy application written in FLTK. The left side is the control panel, consists of a set of buttons corresponding to the series of tasks performed during the procedure. These buttons' callbacks should call the public request methods of the application, which will be translated into state machine inputs. The state machine will then take proper action according to its own state. For example, when the patient information is not set, the 'Load Image' button won't respond to the user click. There is no need for conditional checks or disabling of buttons here as these actions are already in the state machine transition table. On the right hand side, there are four standardized views, axial, sagittal, coronal, and 3D view. Here we loaded an abdominal phantom CT images. The green cylinder represents the needle being tracked by the tracker. The viewer will automatically reslice the images as the needle tip is moving in the anatomy.

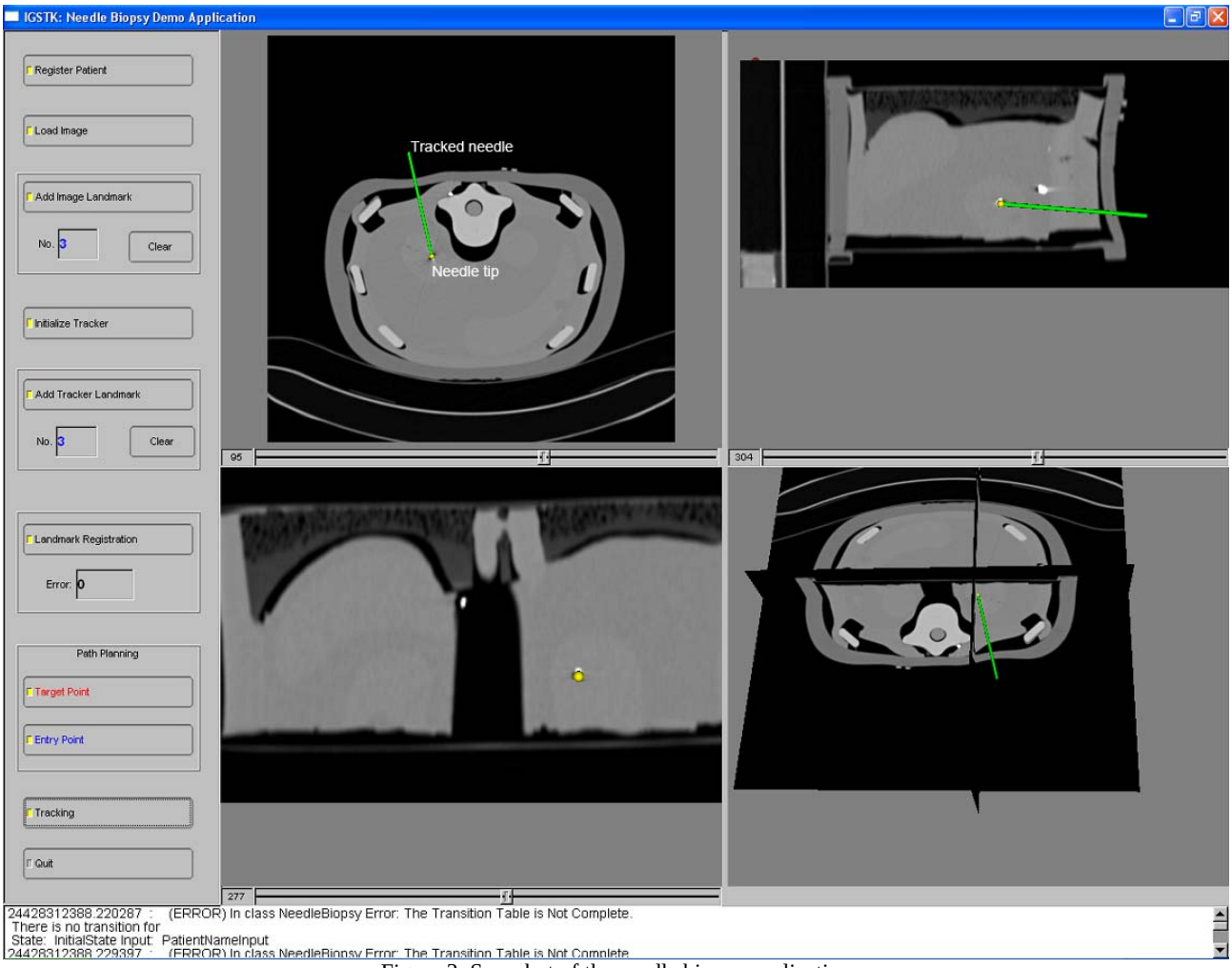

Figure 3: Snapshot of the needle biopsy application

4. CONCLUSIONS

The major components of the toolkit have been completed. Automated nightly testing results can be found at http://public.kitware.com/dashboard.php?name=igstk. The current code coverage on tests is around 95.7%. A state machine validation tool is being developed using LTSA [11], which can animate the transition of the state machine. This tool can be found on the IGSTK wiki page under "Quality" section on "IGSTK State Machine Validation" [12].
This tool is being developed to test all possible combination of inputs to a state machine, which will allow automated testing of applications implemented with state machine. We are planning on releasing three demo applications with this toolkit. The "Needle Biopsy" application as illustrated in the previous sections is available. "Ultrasound Guided RFA" and "Guidewire Tracking" will be available before this phase of the project ends later this year.

The IGSTK toolkit is open source software distributed under a BSD-like license. Basic information related to IGSTK can be found at the website www.igstk.org and the wiki pages [12]. Instructions for configuring and building the toolkit are available at IGSTK wiki page under "Documentation" section on "How to build IGSTK" and "IGSTK: Tutorial". You are welcome to try the software, review the source code and send your comments and report bugs to the IGSTK user list, igstk-users@public.kitware.com. Note that this software should only be used in clinical cases under IRB approval. You are allowed to use IGSTK for free in academic and commercial applications but it is your responsibility

to perform the tests and validations required by regulatory bodies such as the U.S. Food and Drug Administration (FDA).

ACKNOWLEDGMENTS

This research is supported by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) at the National Institutes of Health (NIH) under grant R42EB000374 and by U.S. Army grant W81XWH-04-1-0078. The content of this manuscript does not necessarily reflect the position or policy of the U.S. Government. The authors would like to thank the IGSTK advisory board for their advice throughout the project: Will Schroeder of Kitware; Ivo Wolf of the University of Heidelberg; Peter Kazanzides and Anton DeGuet from John Hopkins University; and Ingmar Bitter, Matt McAuliffee, and Terry Yoo from the NIH.

REFERENCES

1. L. Ibanez and W. Schroeder, “The ITK Software Guide”, ISBN 1-930934-10-6, 2005 http://www.itk.org/ItkSoftwareGuide.pdf
2. W. Schroeder and K. Martin, B. Lorensen, “The Visualization Toolkit, An Object Oriented Approach to 3D Graphics”, Kitware Inc., 1998
3. FLTK a cross-platform C++ GUI Toolkit, http://www.fltk.org/documentation.php/doc-1.1/toc.html
4. L. Ibanez, J. Jomier, D. Gobbi, R. Avila, M. B. Blake, H. Kim, K. Gary, S. Aylward, and K. Cleary. "IGSTK: A State Machine Architecture for an Open Source Software Toolkit for Image-Guided Surgery Applications." Insight journal, Aug 2005
5. K. Cleary, L. Ibanez, S. Ranjan, and M.B. Blake. "IGSTK: A Software Toolkit for Image-Guided Surgery Applications." CARS 2004
6. J.E. Hopcroft, et al., Introduction to Automata Theory, Languages, and Computation, Addison Wesley (2nd ed.), 2000.
7. K. Gary, M. B. Blake, L. Ibanez, D. Gobbi, S. Aylward, and K. Cleary.” IGSTK: An Open Source Software Platform for Image-Guided Surgery.” Submitted to IEEE Computer
8. E.Gamma, et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
9. F. Banovac, N. D. Glossop, D. Lindisch, D. Tanaka, E. Levy, and K. Cleary: Liver Tumor Biopsy in a Respiring Phantom with the Assistance of a Novel Electromagnetic Navigation Device. MICCAI (1) 2002: 200-207
10. Graphviz - Graph Visualization Software, http://www.graphviz.org
11. LTSA - Labelled Transition System Analyser, http://www.doc.ic.ac.uk/~jnm/book/ltsa-v2
12. IGSTK wiki papge, http://public.kitware.com/IGSTKWIKI