Part IV

Example Applications

CHAPTER

TWENTYONE

Needle Biopsy

In this part of the book, we will show three example applications developed using IGSTK. These applications come from our observation of clinical procedures, such as needle biopsy, ultra-sound guided radio frequency ablation, and robot assisted needle placement. You will be able to learn the concepts and work flows of these common image-guided procedures as well as how to implement these applications under the IGSTK framework.

The first application is image guided needle biopsy. The definition for needle biopsy procedure on Society of Interventional Radiology website is:

> Needle biopsy is a medical test performed by interventional radiologists to identify the cause of a lump or mass, or other abnormal condition in the body. During the procedure, the doctor inserts a small needle, guided by X-ray or other imaging techniques, into the abnormal area. A sample of tissue is removed and given to a pathologist who looks at it under a microscope to determine what the abnormality is – for example, cancer, a noncancerous tumor, infection, or scar.(`http://www.sirweb.org/patPub/needleBiopsy.shtml`)

Needle biopsy is a widely used procedure for lung, breast, liver, and prostate cancer diagnosis. A typical image-guided needle biopsy procedure involves first acquiring a pre-operative CT image and then registering the CT image to the patient coordinate system. For this purpose, fiducial-based rigid body registration techniques are commonly used. During the biopsy phase, the needle is tracked by an optical tracking device with real-time visualization of its location overlaid on top of the CT image. This overlay image provides guidance to the surgeon for better targeting of the needle to its desired location.

Figure 21.1 shows the setup of the application. You will need the NDI Vicra tracker to run the program. You can also modify the program to use other supported trackers.

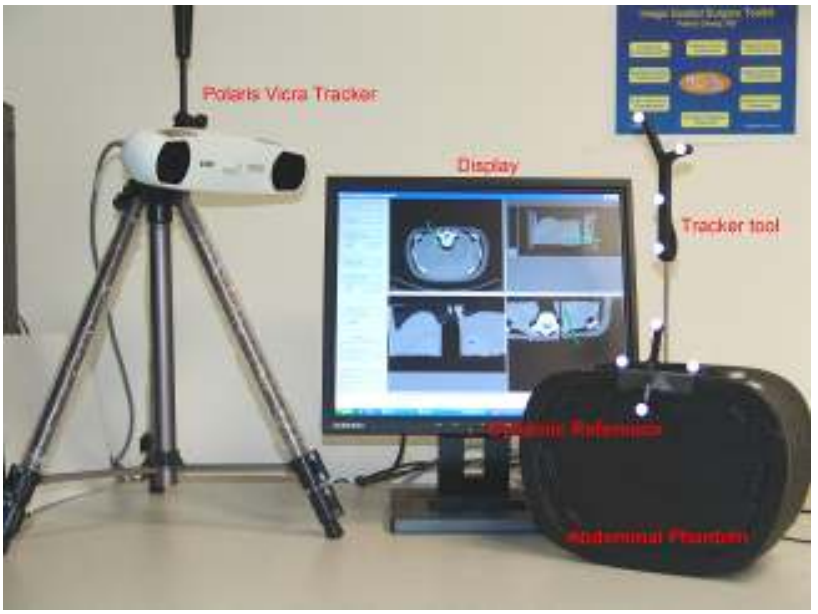

Figure 21.1: System setup for needle biopsy application.

21.1 Running the Application

This application can be found in the Examples/NeedleBiopsy. In order to build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you need to make sure that IGSTK_USE_FLTK is turned ON when running CMake.

The following steps outline the work flow or operating sequences of this application.

1. Obtain the patient demographic information (name, etc.).
2. Load in the pre-operative CT image using the DICOM file format. Fiducials (small markers) are usually placed on the anatomy prior to the scan for landmark based registration in Steps 4-7.
3. Verify the patient information against the information in the image. Prompt the surgeon if there is a discrepancy. This step is typical of the error checking that should be done and one should assume that if anything can go wrong it will go wrong and safeguards should be provided.
4. Identify the image landmarks by going through the CT image slices and selecting the fiducials using the mouse. For paired-point based registration, at least three points are required although at least four are preferred.
5. Initialize the tracking device.

6. Add patient landmarks by touching the physical fiducials attached to patient using the tracked pointer device.
7. Perform the image to patient landmark registration.
8. Path planning. The surgeon will select a target point and an entry point to plan the path for the needle puncture.
9. Provide a real-time display of the overlay of the needle probe and pre-operative images in the quadrant viewer window during the biopsy procedure.

21.2 Implementation

State machine is very important and distinct feature in IGSTK. Although it seems to be very complicated to many people, we are trying to provide some guidance here to help users to familiarize this concept and design pattern.

21.2.1 State Machine in Application

Given that IGSTK is intended for developing applications that will be used to treat patients, robustness and quality are the highest priorities in the design of the toolkit. To minimize the risk of harm to the patient resulting from misuse of the classes, IGSTK incorporates state machine design pattern into its components. All IGSTK components are governed by a state machine. A state machine is contained within the class to control the access to the class. Components are always in a valid state to ensure they will perform in a predictable manner. The use of a state machine also helps enforce high quality standards for code coverage and run-time validation.

In this example, state machine is being implemented at the application level(Figure 21.2 shows the partial state machine diagram of this application). While this is not mandatory, it is strongly recommended when using IGSTK. A state machine architecture gives the application developer an easier way to prototype the application and to control the work flow of the surgical procedure, and also adds an extra layer of security to the application to make it more robust. The following sections demonstrate how to write an application using the IGSTK framework.

21.2.2 Mapping clinical work flow to state machine

The first step to develop an application is to analyze the surgical procedure and develop a minimal specification. By analyzing a typical needle biopsy procedure, we identify a serial of tasks or work flow. Then it becomes relatively easy to translate clinical work flow into state machine logic. If we think the application as a state machine, the completion of each task will cause the application to enter a new state, and there will be a set of states to indicate the status of the application. The user interaction with the GUI can be translated into inputs to the state machine.

Figure 21.2: State machine diagram (partial) for the needle biopsy application(Circle for state and arrow for transition and corresponding input).

For instance, when we click on the register patient button, this will generate a “RequestSetPatientNameInput” to the state machine. The state machine will take this input and change its current state from “InitialState” to “WaitingForPatientNameState”, and the action is to pop up a window asking for input of the patient name. If the user inputs a valid name, then there will be a “PatientNameInput” which brings the state machine into “PatientNameReadyState”, otherwise there will be a “PatientNameEmptyInput”, which will return the state machine to the “InitialState”. Thus, we can map the application into series of states and inputs and this higher level abstraction will help the developer design a clear work flow for the application. Figure 21.2 shows the state machine diagram for the needle biopsy application which was generated automatically when the state machine is constructed using the ‘dot’ tool from Graphviz.

21.2.3 Coding the state machine

This section shows how to code the state machine into the needle biopsy application. IGSTK has a number of convenient macros to facilitate the programming of the state machine. The details of these macros can be found in `Source/igstkMacro.h`. More information on the state machine design pattern and guidelines can be found on the IGSTK wiki page under ”Development” section on “Design discussions” page.`http://public.kitware.com/IGSTKWIKI`

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

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

   ```
   igstkStateMachineMacro();
   ```

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

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

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

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

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

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

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

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

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

6. Now the state machine is setup and ready to run. We can then export the state machine description in dot format and generate the graphical visualization as shown in Figure 21.2. This graph will help us to examine the workflow of the application and the state transition table.

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

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

   ```
   >dot -T png -o SMDiagram.png DemoApplicationStateMachineDiagram.dot
   ```

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

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

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

21.2.4 Should I use the state machine in my application?

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

In comparison to traditional approaches, state machines will reduce the number of paths in the code, save the developers from convoluted conditional tests, and encourage them to focus on higher level design. From the above example, we can see that the state machine is easy to program and manage under the IGSTK framework. We encourage developers to design and code the state machine of their application first, and then generate the state machine diagram as shown in Figure 21.2. They can go through the diagram, examine and verify their design of the work flow. If they want to add or change a path of the application, it is just a matter of adding or deleting a transition table entry. This eliminates the level of difficulty required for going through the code and struggling with if-then-else logic. This will largely facilitate the application prototyping, and the implementation code can be plugged into the skeleton program later. These techniques should result in clearer designs and safer applications.

21.3 Result

Figure 21.3 shows the user interface of the needle biopsy application written in FLTK. The left side is the control panel, consists of a set of buttons corresponding to the series of tasks performed during the procedure. These buttons' callbacks should call the public request methods of the application, which will be translated into state machine inputs. The state machine will then take proper action according to its own state. For example, when the patient information is not set, the 'Load Image' button won't respond to the user click. There is no need for conditional checks or disabling of buttons here as these actions are already in the state machine transition table. On the right hand side, there are four standardized views, axial, sagittal, coronal, and 3D view. Here we loaded an abdominal phantom CT images. The green cylinder represents

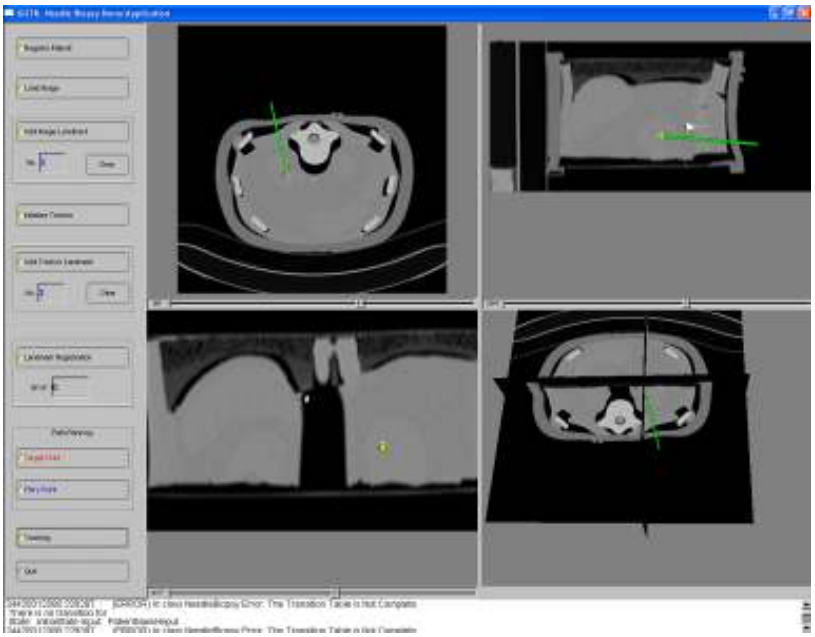

Figure 21.3: User interface for needle biopsy program.

the needle being tracked by the tracker. The viewer will automatically reslice the images as the needle tip is moving in the anatomy.

CHAPTER

TWENTYTWO

Ultrasound Guided Radio-Frequency Ablation

22.1 Introduction

Liver lesions suitable to be treated using radio-frequency ablation (RFA) are often clearly visible under CT/MR but not using Ultrasound (US) imaging. Therefore most of the RFA surgeries of the liver are performed under CT. Other alternatives consist of waiting until the lesions enlarge and show up under US or perform an open surgery. An ideal visualization system would show tumor contours under US to the surgeons. A typical workflow of an RFA surgery is described in figure 22.1.

This application registers a pre-operative model of the tumors with a 2D US slice in pseudo real-time. The system can be divided into three parts: a) tracking devices, b) registration algorithm and c) display. First, the 2D ultrasound probe is tracked using an optical tracker (Polaris from NDI). Second, an image-to-image registration algorithm registers each 2D slice with a the pre-operative CT. One can notice that this registration step should be performed as quickly as possible. Third and last, a display presents the actual 2D US slice with tumor outlines to the surgeon.

Next the different components of the application, the tracking systems and the registration algorithm are presented.

22.2 Running the Application

This application can be found in the Examples/UltrasoundGuidedRFA. In order to build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you need to make sure that IGSTK_USE_FLTK is turned ON when running CMake.

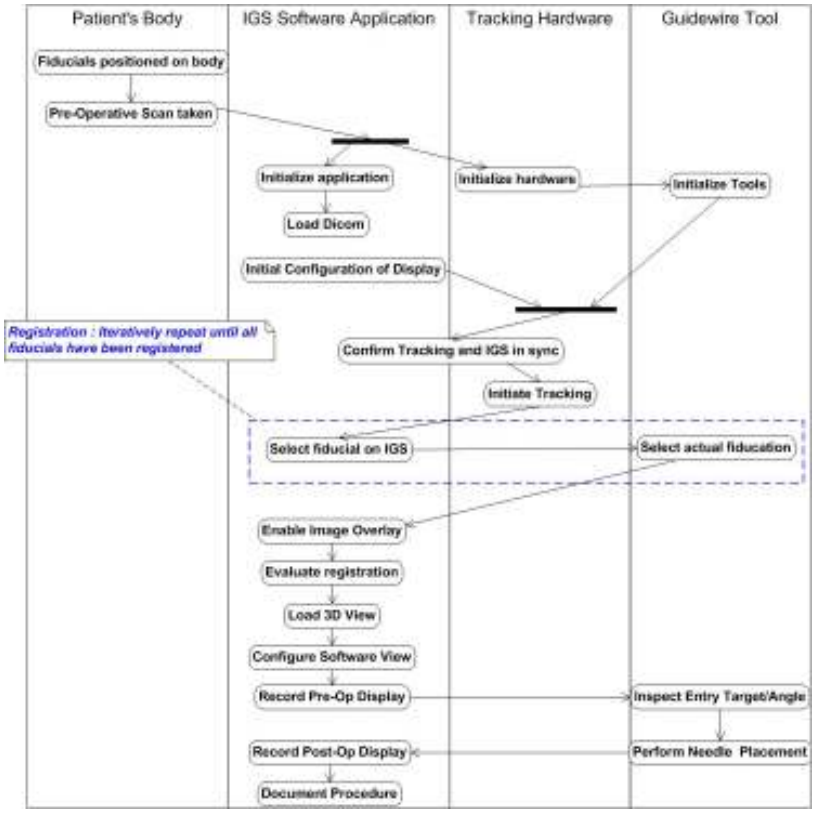

Figure 22.1: Typical RFA Ablation Surgery Workflow.

22.3 Implementation

22.3.1 Tracker

Using a tracker in IGSTK is quite easy. First we create a tracker object using smart pointers.

```
typedef igstk::PolarisTracker    TrackerType;
TrackerType::Pointer m_Tracker = TrackerType::New();
```

Then, we set a ToolCalibrationTransform which defines the relationship between the tracking device and the origin of the tool. In our case, the optical sensor is attached to arm the probe, therefore the calibration transform is defined as a rigid transform from the sensor position to the tip of the probe. This transform can be computed from a calibration experiment or using some heuristics.

```
m_Tracker->SetToolCalibrationTransform( TRACKER_TOOL_PORT, 0,
                                                                    ToolC
```

Next, we need to define the relationship between the tracking system origin and the actual patient position in the OR. Most of the surgical applications define the OR as the world coordinate origin. This PatientTransform is often assessed via calibration.

```
m_Tracker->SetPatientTransform(PatientTransform);
```

The IGSTK spatial object to be tracked is attached to the tracker using AttachObjectToTrackerTool. Therefore when the position and orientation of the tracking device is modified, the updated position of the spatial object is automatically computed. One can notice that this step involves the concatenation with the ToolCalibrationTransform and the PatientTrasnform.

```
m_Tracker->AttachObjectToTrackerTool( TRACKER_TOOL_PORT,
                                                          TRACKER_T
                                                  m_UltrasoundP
```

Our tracker is ready to be used, we start the tracking by first opening the serial communication port using `Open()`, then we initialize the tracker and start the tracking.

```
m_Tracker->Open();
m_Tracker->Initialize();
m_Tracker->StartTracking();
```

To stop the tracking device we just use the `StopTracking()` function.

```
m_Tracker->StopTracking();
```

One can notice that switching from one tracker to another can be done by modifying a single line of the code above, e.g. the tracker type definition.

22.3.2 Registration

Using the tracking information of the ultrasound probe the location of the US slice is roughly defined in the OR. To define a proper alignment of the US slice and the pre-operative CT volume registration is needed. The registration algorithm is an image-to-image technique based on the cross-correlation.

IGSTK makes use of registration algorithms already implemented in the Insight Segmentation and Registration Toolkit [**?**]. However, IGSTK propose algorithm already tuned from specific modalities and organs. Using class hierarchies, programmers can still makes use of higher level registration technique. For instance, the igstkMR3DImageToUS3DImageRegistration class performs registration of any 3D MR to 3D US. The parameters of the registration are already tuned to support most of the MR-to-US registrations but some other tuning might be required for different organs.

22.3.3 Display

IGSTK propose several visualization techniques based on the Visualization Toolkit [**?**]. Basically for a given IGSTK spatial object, several representation objects can be created and added the display as shown in figure **??**.

Image volumes such as CT or MR datasets can be renderered as textured oblique slices or can be volume renderered. Mesh objects such as segmented tumors can be renderered in 3D as triangle surfaces and in 2D as contours. The update of the display is done in real-time when the number of objects in the scene is not excessive and does not require extensive computation, i.e volume rendering of large datasets.

22.3.4 Implementation

Here we show an example on how to read and display a vasculature extracted from CT.

First we create a vascular network reader using smart pointers.

```
typedef igstk::VascularNetworkReader  VascularNetworkReaderType;
VascularNetworkReaderType::Pointer m_VascularNetworkReader;
m_VascularNetworkReader = VascularNetworkReaderType::New();
```

Then we set the vasculature filename and we ask the reader to read the file using the RequestReadObject() function.

```
m_VascularNetworkReader->RequestSetFileName(VasculatureFilename);
m_VascularNetworkReader->RequestReadObject();
```

In order to get a spatial object from a reader, IGSTK uses the *event/observer* mechanism. We declare a specific observer to get the vasculature from the reader and we ask the reader to return the object.

```
VascularNetworkObserver::Pointer vascularNetworkObserver
                                          = VascularNetworkObserver::New();
m_VascularNetworkReader->AddObserver(
                         VascularNetworkReader::VascularNetworkModifiedEvent(),
                         vascularNetworkObserver);
m_VascularNetworkReader->RequestGetVascularNetwork();
```

Next, we instantiate an object representation for the VascularNetwork object.

```
 typedef igstk::VascularNetworkObjectRepresentation
                                        VascularNetworkRepresentationType;
 VascularNetworkRepresentationType::Pointer m_VascularNetworkRepresentation =
                                                                        VascularN
```

Then we set the spatial object to the object representation. Internally the object representation creates a suitable visualization of the object from its internal geometry.

```
m_VascularNetworkRepresentation->RequestSetVascularNetworkObject(
                               vascularNetworkObserver->GetVascularNetwork() );
```

Finally, we add the object to the display.

```
this->Display3D->RequestAddObject( m_VascularNetworkRepresentation );
```

22.4 Conclusion

CHAPTER

TWENTYTHREE

Robot Assisted Needle Placement

In the previous needle biopsy application (Chapter 21), with the image guidance the physician might still be limited by the view of the exact position of any surgical instruments in the interventional field, and they might need to spend a fair amount of time to align the instruments with the planned path. In this chapter we present an image-guided platform for precision placement of surgical instruments based upon a small four degree-of-freedom robot shown in Figure 23.1(B-RobII; ARC Seibersdorf Research GmbH, Vienna, Austria). The robot has two joints (upper box and lower box, which can move parallel to each other) and 4 degree of freedoms. Its unique shape gave it the name deck of card robot. This platform includes a custom needle guide with an integrated spiral fiducial pattern as the robot's end-effector and uses pre-operative computed tomography (CT) to register the robot to the patient directly before the intervention. The robot can then automatically align the instrument guide to a physician-selected path for percutaneous access. The path is chosen by the physician before the intervention using an established graphical user interface built using open-source toolkits such as the Image-Guided Surgery Toolkit (IGSTK). Potential abdominal targets include the liver, kidney, prostate, and spine.

Figure 23.2 shows the setup of the whole system. The robot is mounted on the CT table after patient (we use phantom in the picture) is in place. Robot arm is adjusted to position the needle holder close to the biopsy area. A CT scan is then acquired and loaded into robot assisted needle biopsy application. Surgeon can go through the image slices, identify tumors, and plan an optimal biopsy path by setting proper target and entry points to avoid important and vulnerable organs and tissues. The robot will then move the needle holder and align it with the planed path. Surgeon can advance the needle manually to hit the target. The deck of card robot can be operated remotely by multiple clients through TCP/IP communication. Client application should first connect to the server application as an active client before it can command the robot.

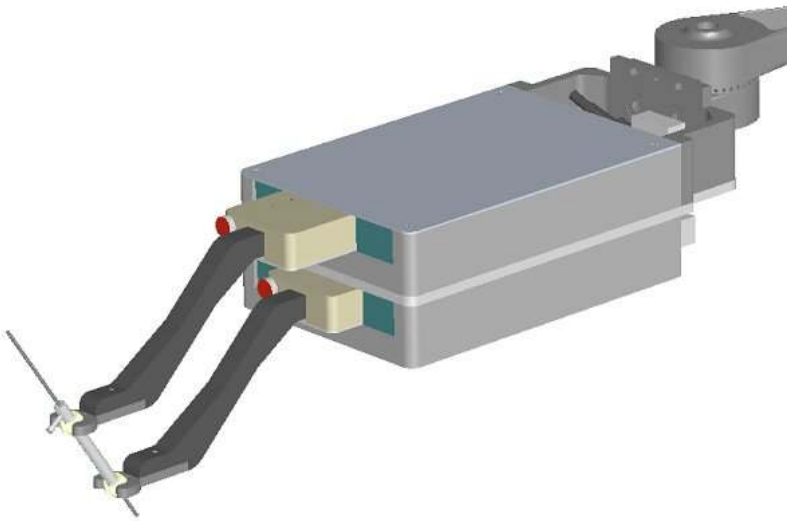

Figure 23.1: B-RobII four degree-of-freedom precision placement modules for needle positioning and orientation.

Figure 23.2: Robot assisted needle placement phantom study setup.

23.1 Running the Application

This application can be found in the `Examples/DeckOfCardRobot`. In order to build this application you will need the cross-platform FLTK GUI toolkit available from www.fltk.org. Then you need to make sure that IGSTK_USE_FLTK is turned ON when running CMake.

As shown in Figure 23.3, the workflow of this application are(suppose we have a lung biopsy procedure):

1. Place the phantom on the CT table and mount the robot to the CT gantry.
2. Position the robot needle holder close to the region of interest.
3. Scan the phantom together with the robot.
4. Load CT images into the robot control software.
5. Using the control software, segment out the fiducials in the CT image and perform the paired-point registration.
6. In the display window, plan the needle insertion path.
7. If the planned path is within the robot's working range, then command the robot to align the needle to the planned path. Otherwise, go back to step 2 and reposition the robot closer to the biopsy entry point.
8. Advance the needle by hand. The depth of insertion will also be calculated by the system and this depth can be judged by observing depth graduations on the needle itself.

23.2 Implementation

This application is unique in a way that it is using 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 another word, how to interface and extend IGSTK library.

23.2.1 Pass IGSTK image objects to ITK filters

One task of the application is to automatically detecting the fiducial pattern in the CT images. It makes sense to leverage 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 loaded "event", and we need a "observer" to catch that "event".

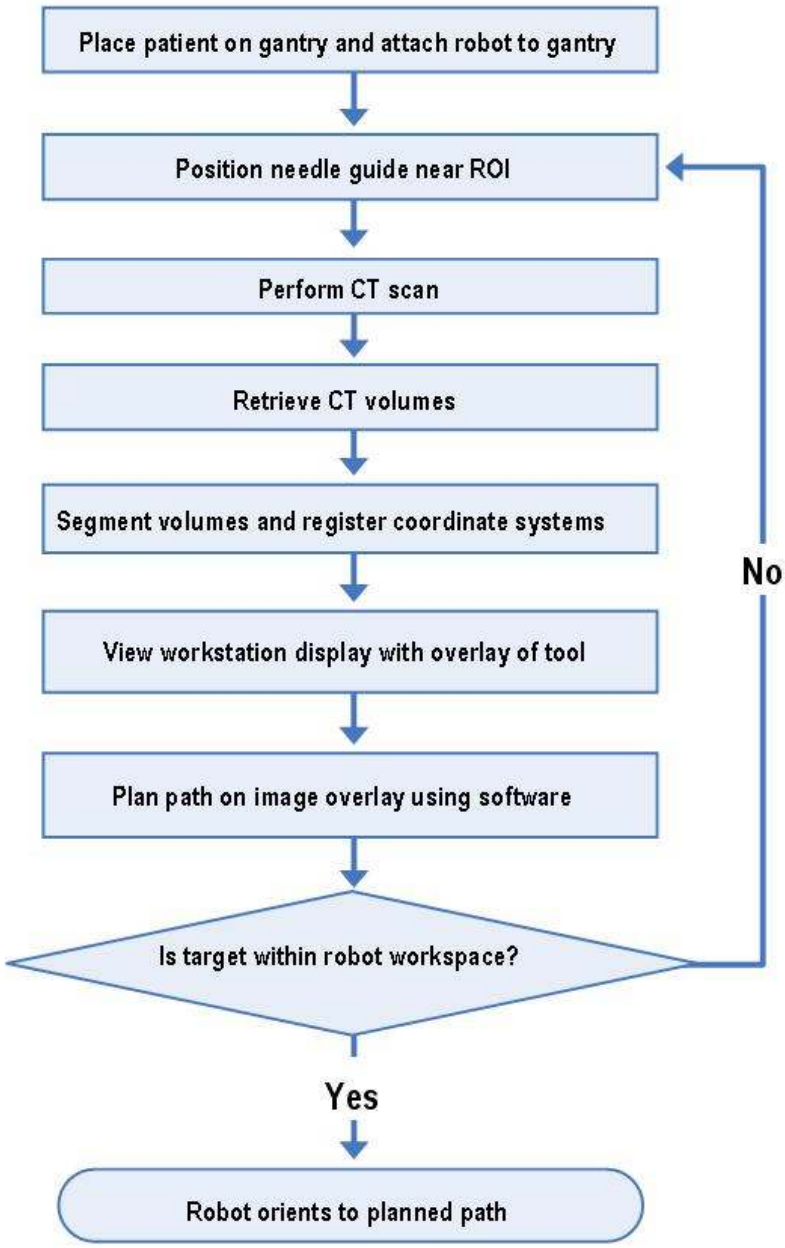

Figure 23.3: Clinical Workflow for Robot Assisted Needle Placement.

1. First step, we need to use the igstkObserverConstObjectMacro to define the observer

```
igstkObserverConstObjectMacro( ITKImage,
                               ImageSpatialObjectType::ITKImageModifiedEvent,
                               ITKImageType)
```

   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 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 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 catched 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.

23.2.2 Write your own representation class

If you want to have your own visualization class to get special rendering effect, you might 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 reused its frame and old state machine logic since these two classes perform similar tasks except using different rendering technique.

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 only supported by recent graphics cards.

```
#include "vtkImageShiftScale.h"
#include "vtkColorTransferFunction.h"
#include "vtkPiecewiseFunction.h"
#include "vtkVolumeTextureMapper3D.h"
#include "vtkVolumeProperty.h"
#include "vtkVolume.h"
```

3. Next thing to do is declaring member variables necessary for volume rendering

```
vtkPiecewiseFunction *                  m_OpacityTransferFunction;
vtkColorTransferFunction *              m_ColorTransferFunction;

vtkImageShiftScale *                    m_ShiftScale;
vtkVolumeTextureMapper3D *              m_VolumeMapper;
vtkVolumeProperty *                     m_VolumeProperty;

vtkImageData *                          m_ImageData;
vtkVolume *                             m_ImageActor;

unsigned                                m_ShiftBy;
unsigned                                m_MinThreshold;
unsigned                                m_MaxThreshold;
```

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 mentioned in section 23.2.1

```
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( m_ImageData );
  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 pass them to a `vtkVolumeProperty`, and 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. 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.

23.2.3 Using the socket communication class

The communication between the application and robot server is through TCP/IP protocol. For this purpose, socket communication component of IGSTK was 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 we 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;%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 done, log out of the robot and close the communication.

```
m_Client->RequestWrite("@QUIT\r\n");
m_Client->RequestCloseCommunication()
```

23.3 Result

Figure 23.4 shows the user interface of the application with control panel on the left, 3 standard 2D slice views and a 3D volume rendering on the right. The yellow cylinder is the needle holder, the purple square indicates robot's working region, a path is being planed to target the tumor while avoiding the ribs, and it is showing the robot being aligned with the planned path.

Figure 23.5 shows TeraRecon rendered image of the robot needle holder.

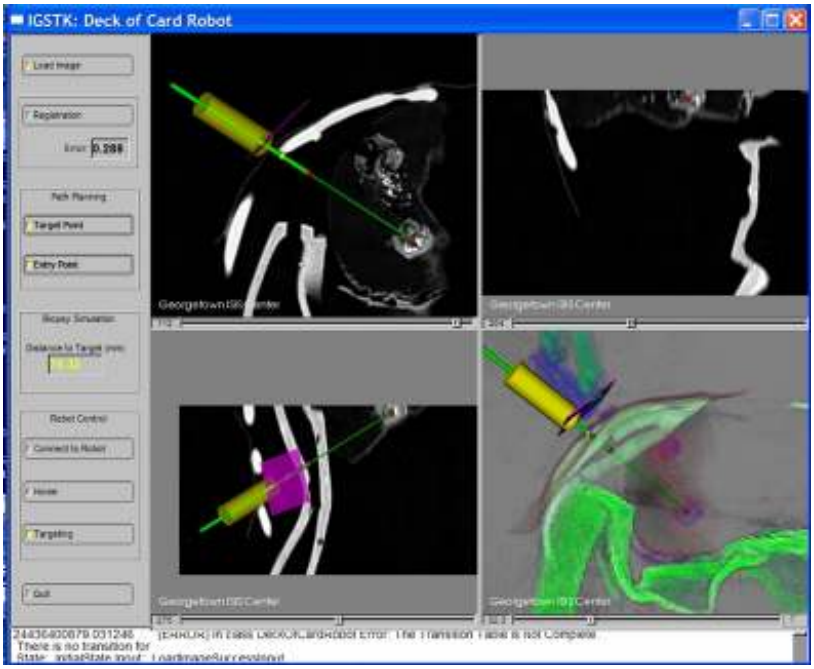

Figure 23.4: User interface for the robot application.

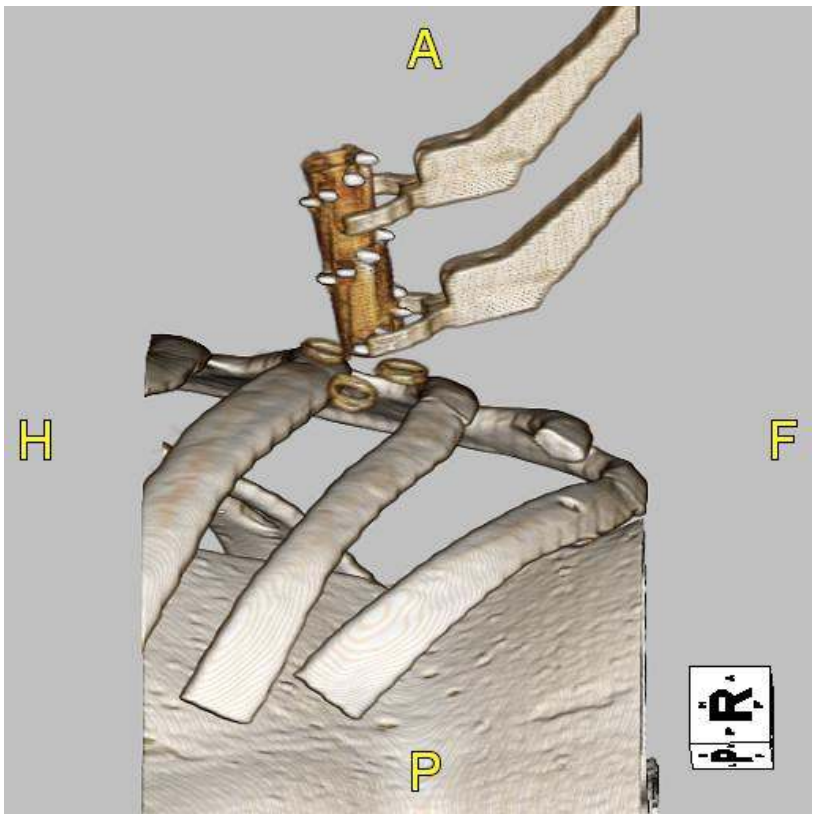

Figure 23.5: 3D rendering of CT scan showing spiral fiducial pattern for registration of robot to CT images.