CHAPTER

FIFTEEN

Calibration

In the image guided surgery, a surgeon uses tracked tools for navigation or intervention. A general transform procedure called calibration is required to align the output of the tracking device to the specific operation points on the tool, such as the tip of a surgical needle or the end of its handle. For surgeons, these specific operation points are more important and intuitive than the positions of the trackers which are often embedded in the middle or at the hind of the tool.

Generally speaking, calibration refers to the process of setting the magnitude of the output (or response) of a measuring instrument to the magnitude of the input property or attribute within a specified range of accuracy and precision [1]. When calibrating the tracker, those inputs are the positions and the orientation reported from the tracking device, and the outputs are the positions of the specific operation points. When calibrating both the tracker and an imaging device, such as an ultrasound probe, the inputs are the positions of the probe tracker and the image coordinate in the ultrasound imaging, and the outputs are the 3D coordinates of those image pixels in the global world coordinate system. Hence, in most cases, the calibration procedure produces the transform between the trackers and specific positions.

In the IGSTK calibration package, several calibration classes are currently provided in the main repository and sanbox, including igstkPivotCalibration, igstkPrincipalAxisCalibration and igstkLandmarkUltrasoundCalibration. Those calibration classes work closely with the tracker component to provide a safe and accurate environment for the image-guided surgical procedures. Some specialized features, such as the essential distortion calibration for the electromagnetically tracking device, will also be provided following the requirement process of IGSTK framework.

This chapter will firstly describe the rationale and design patterns for the existing calibration components. Next it will introduce the calibration data format used and its I/O classes in the sample application. Finally, it will discuss the future extension of the calibration package.

[1] http://en.wikipedia.org/wiki/Calibration

15.1 Pivot Calibration

15.1.1 Introduction

The reported positions from both optical and electro-magnetic trackers are generally at fixed points on their sensors. For example, in the AURORA tracking system, the reported position for a single sensor is at the center of the sensor coil that is embedded in the surgical tool. When multiple sensors are used, the reported position is determined by the configuration SROM file that is stored in the tools. Generally, this position is a specific point that is rigidly related to the tracked tool.

For a tracked needle, its tip and the end of its handle are important for the surgeon. Those points provide an intuitive means to visualize and locate the tool's body. In an image-guided surgery application, a point at the tip of the tool is always used to locate the spatial position of landmark points, such as skin fiducials or the internal bifurcation positions of the vessels. Additionally, for the navigation and validation, the tip of the tracked needle or the guide-wire indicates the surgeon's point of focus. Thus, a tracked tool can also serve as a 'locator' or a 'pointer'. The transform between the internal sensor or marker's positions to those specific operation points is accomplished by a procedure called pivot calibration.

Most tracking device manufacturers provide specific software to handle this pivot calibration; for example, Northern Digital Inc. provides '6D Architecture Aurora' and 'Toolviewer' for this purpose. This kind of software establishes communication, tracks the tools, visualizes the positions, and calculates the pivot calibration transform. The pivot calibration result is calculated before the experiment or the procedure, then used by the application for each specific tool. However, for the on-site pivot calibration, a general purpose class gives the developer more flexibility.

In a typical pivot procedure like the one shown in Figure 15.1 [2], the tip of the instrument is placed in a divot (a series of small holes) and the instrument is rotated back and forth (it pivots) while tracking data is collected with enough sample input, the transformation from the tracked sensor's point to the pivot point is calculated, along with the calibration error represented as a root mean square error.

15.1.2 Principle

The information from the tracker consists of the position and the orientation. Some systems also provide the measurement error at that position. Related with the original point, the position is located by a translation vector and a quaternion that represents the rotation from the default principal axis (mostly along the z-axis from the manufacturer settings). Depending on the sensor or sensors, the reported tracker information may have three, five, or six degrees of freedom (DOF). The first three degrees are represented by the translation, and the others are determined by the quaternion. The transformation from the original point to the tool tip in the tracking coordinate system is represented by:

[2] NDI 6D Architecture Aurora

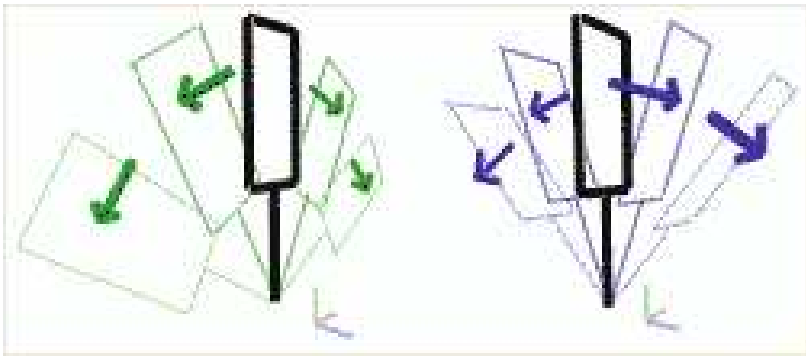

Figure 15.1: Pivot Calibration Routine.

$$\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{15.1}$$

In this equation, R is the rotation matrix, T is the translation vector and (x_0, y_0, z_0) is the pivot position. In most pivoting cases, the pivot position is the tip of the tool. Typically, we record several hundred samples while pivoting the tools. Equation 15.1 can be re-written as follows where the constraints of offset and (x_0, y_0, z_0) are arranged as:

$$\begin{aligned} r_{00} \cdot offset_x + r_{01} \cdot offset_y + r_{02} \cdot offset_z - 1 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 &= -t_x \\ r_{10} \cdot offset_x + r_{11} \cdot offset_y + r_{12} \cdot offset_z + 0 \cdot x_0 - 1 \cdot y_0 + 0 \cdot z_0 &= -t_y \\ r_{20} \cdot offset_x + r_{21} \cdot offset_y + r_{22} \cdot offset_z + 0 \cdot x_0 + 0 \cdot y_0 - 1 \cdot z_0 &= -t_z \end{aligned} \tag{15.2}$$

With several input samples, those equations can then be accumulated as:

$$M \cdot \begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ x_0 \\ y_0 \\ z_0 \end{bmatrix} = N \tag{15.3}$$

Since M is not a square matrix, the unknowns are solved using the singular value decomposition (SVD) or Moore-Penrose inverse:

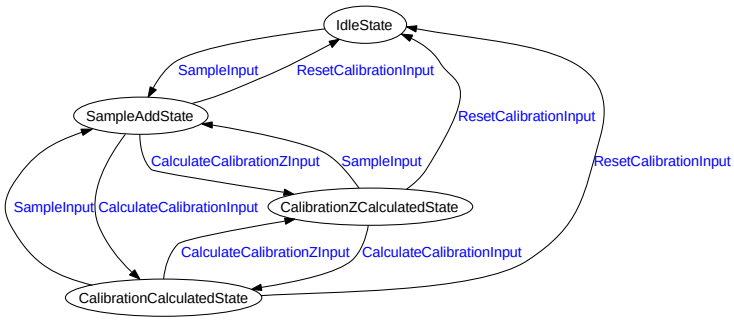

Figure 15.2: State Diagram of the igstkPivotCalibration class.

$$\begin{bmatrix} offset_x \\ offset_y \\ offset_z \\ x_0 \\ y_0 \\ z_0 \end{bmatrix} = (M^T \cdot M)^{-1} \cdot M^T \cdot N \tag{15.4}$$

Additionally, the root mean square error is computed as:

$$RMS = \sqrt{|M \cdot [offset_x \;\; offset_y \;\; offset_z \;\; x_0 \;\; y_0 \;\; z_0]^T - N|^2 / num} \tag{15.5}$$

where *num* is the number of samples. Note that in the pivot calibration procedure, the final transform only contains only a translation factor and no rotation factor.

When calculating the calibration only along the Z-axis only for a cylinder-like track tools, sometimes users may only want to get the calibration offset along the Z-axis. In this condition, the $offset_x$, $offset_y$ variables are restricted to 0 and the computation formulas are slightly changed.

15.1.3 State Machine Diagram

Figure 15.2 is a state machine diagram of the `igstk::PivotCalibration` class. There are four states inside this class: initial 'Idle' state, 'SampleAdd' state, 'CalibrationCalculated' state and 'CalibrationZCalculated' state. The 'Idle' state is the initial state for all IGSTK classes. When the position samples from pivoting the tracker are input into the class, the internal state will be invoked to the 'SampleAdd' state. As described in the previous section, there are two ways to calculate the calibration matrix, either in the full translation mode or in the Z-axis only mode. The final stages are the 'CalibrationCalculated' and CalibrationZCalculated' states respectively. Only in these two states can the PivotCalibration class return a valid calibration transform.

15.1.4 Component Interface

The core functions of the igstkPivotCalibration class include:

1. Input the samples (translation and quaternion) from the pivoting trackers;
   - igstkPivotCalibration::RequestAddSample();
2. Calculate the calibration transform;
   - igstkPivotCalibration::RequestCalculateCalibration();
   - igstkPivotCalibration::RequestCalculateCalibrationZ();
3. Return the final calibration transform and pivot position;
   - igstkPivotCalibration::GetValidCalibration();
   - igstkPivotCalibration::GetCalibrationTransform();
   - igstkPivotCalibration::GetPivotPosition();
4. Calculate the root mean square error to evaluate whether a feasible calibration transform has been computed;
   - igstkPivotCalibration::GetRootMeanSquareError();
5. Provide the convenient function to retrieve the input sample;
   - igstkPivotCalibration::GetNumberOfSamples();
   - igstkPivotCalibration::RequestGetInputSample();
6. Provide the convenient function to simulate the pivot position for any input translation and quaternion from the calculated calibration transform;
   - igstkPivotCalibration::RequestSimulatePivotPosition();

15.1.5 Example

The source code for this section can be found in the file `Examples/PivotCalibration/PivotCalibration1.cxx`.

This example illustrates how to use IGSTK's pivot calibration class to determine a calibration matrix for the tracker tools.

To use the pivot calibration component, the header file for `igstk::PivotCalibration` should be added.

```
#include "igstkPivotCalibration.h"
```

After defining the headers, the main function implementation is started.

```
int main( int argc, char * argv[] )
{
```

All the necessary data types in the pivot calibration are defined. VersorType and VectorType are used to represent the quaternion and translation inputs from the tracker; PointType is used to represent the position coordinate of the specific point; and ErrorType is used to represent the root mean square error.

```
  typedef igstk::PivotCalibration           PivotCalibrationType;
  typedef PivotCalibrationType::VersorType  VersorType;
  typedef PivotCalibrationType::VectorType  VectorType;
  typedef PivotCalibrationType::PointType   PointType;
  typedef PivotCalibrationType::ErrorType   ErrorType;
  typedef itk::Logger                       LoggerType;
  typedef itk::StdStreamLogOutput           LogOutputType;
```

At the beginning, a pivot calibration class is initialized as follows:

```
  PivotCalibrationType::Pointer pivot = PivotCalibrationType::New();
```

A logger is then created for logging the process of calibration computation, and then attached to the pivot calibration class

```
  LoggerType::Pointer                       logger = LoggerType::New();
  LogOutputType::Pointer                    logOutput = LogOutputType::New();

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

  pivot->SetLogger( logger );
```

To use the pivot calibration class, some input samples from the tracker should be provide. Those samples come directly from tracker tools. In our example, those samples are read from the record data file in the IGSTK data directory.

```
  input.open( argv[1] );

  if (input.is_open() == 1)
    {
    std::cout << "PivotCalibration data open sucessully!" << std::endl;
    }
  else
    {
    std::cout << "PivotCalibration data open error!" << std::endl;

    return EXIT_FAILURE;
    }
```

Before the computation, it is better to reset the calibration class to to remove all necessary information which may come from the previous codes.

```
pivot->RequestReset();
```

Then, the sample frame is read from the data file and input to calibration class

```
while ( !input.eof())
  {
  double vx;
  double vy;
  double vz;
  double vw;

  input >> frame >> temp >> time;
  input >> pos[0] >> pos[1] >> pos[2];
  input >> vw >> vx >> vy >> vz;

  versor.Set( vx, vy, vz, vw );
  pivot->RequestAddSample( versor, pos );
  }
```

After this, a simple request will invole the class to compute the calibration transform.

```
pivot->RequestCalculateCalibration();
```

Before the final calibration transform is retrieved, the user should check the tag to see whether a valid calibration has been computed. The final calibration result is stored in the translation factor in the transform matrix. The pivot position is also retrievable. The sample code is as follows:

```
if ( !pivot->GetValidPivotCalibration())
  {
  std::cout << "No valid calibration!" << std::endl;

  return EXIT_FAILURE;
  }
else
  {

  // Get the calibration transformation
  VectorType translation = pivot->GetCalibrationTransform().GetTranslation();

  // Get the pivot focus position
  PointType position = pivot->GetPivotPosition();
```

```
// Get the calibration RMS error
ErrorType error = pivot->GetRootMeanSquareError();

// Dump the calibration class information
std::cout << "PivotCalibration: " << std::endl;
std::cout << "NumberOfSamples: " << pivot->GetNumberOfSamples()
                                 << std::endl;
std::cout << "Translation: " << translation << std::endl;
std::cout << "Pivot Position: " << position << std::endl;
std::cout << "Calibration RMS: " << error << std::endl;

}
```

For only computing the calibration along Z-axis, another function is used instead:

```
pivot->RequestCalculateCalibrationZ();
```

15.2 Principal Axis Calibration

15.2.1 Introduction

The pivot calibration provides only the translation information of the tracked tool and thus is based on the assumption that the geometry coordinate and the default tracking coordinate are the same. In most cases, the principal axis is along the Z-axis. If the principal axis of the tracked tool is not well aligned with the geometry representation, a rotation transform is required to work together with the pivot calibration routine to construct the full transform matrix. Since IGSTK works closely with ITK, and the transform is based on the itk::Vector and itk::Versor classes, the itk::Versor can be used to represent the rotation directly. For example, if the tracked tool's principal axis is along the Z-axis, but the spatial geometry object is along the Y-axis, an itk::Versor::SetRotationAroundX() function solves the problem. This approach works for these specific rotations, but for an arbitrary rotation between two unspecific directions it is hard to use a combination of rotations around the X, Y or Z-axes to produce the exact result. In this case, a general purpose igstkPrincipalAxisCalibration class provides an intuitive means to set the initial and desired orientations (tracker tool's principal axis and spatial object's principal axis) and to return the rotation between them.

For cylindrical tracker tools, the spatial principal axis runs mostly along the cylinder geometry axis. In some different configurations, this alignment may change. For example, the ITK spatial object's default axis is along the Y-axis, but the IGSTK spatial object's default axis is along the Z-axis. For the 5DOF and 6DOF trackers, the default principal axis in the tracking coordinate system, as defined by most hardware manufacturers, is along the Z-axis. Note that this default principal axis in the tracking coordinate system often is not along the tracker tool's geometry principal axis. A combination tracker tool is shown in Figures 15.3 and 15.4 [3]. For the tracker

[3] Traxtal Inc.

handle, the optical markers are arranged along the handle and the tracking coordinate system's principal axis is well aligned with the spatial geometry shape. But for the tracker probe tip, the specially designed curves make them different. In an image-guided surgery application, these probe tips are of greater concern to the surgeon, for they are real operational parts and will touch the patient. When the tracker probe tip and the handle are attached together, the principal axis in the tracking coordinate system is different than the operational part's principal axis. For those tracker tools, an igstkPrincipalAxisCalibration class provides an intuitive means to calculate the rotation matrix.

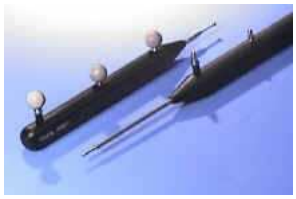

Figure 15.3: Tracker Handle

Figure 15.4: Tracker Probe Tip

15.2.2 Principle

The purpose of computing the principal axis calibration is to find the rotation between two defined orientations. Our method is based on the rotation matrix multiplication. In IGSTK, the rotation factor is stored in the quaternion format, and the convert between the quaternion and the rotation matrix is unified. When a point is rotated, in the mathematically representation, the vector is multiplied by a rotation matrix. So, the difference between two orientations can be calculated by the divide operation of the two matrices, as follows:

$$M = M_{ori1} \cdot M_{ori2}^{-1} \tag{15.6}$$

M_{ori1} is the rotation matrix from the desired orientation, and M_{ori2} is the rotation matrix from the initial orientation.

The igstkPrincipalAxisCalibration class provides an intuitive means to set the orientation, which is defined by the principal axis and the normal axis of the tool. The principal axis is generally the major axis along the cylindrical tools that are very popular in clinical procedures. The normal vector defines the view-up direction of the system coordinate. Those two parameters are easy to measure by the geometry shape design profile of the tools. The rotation matrix can be built from the principal axis and normal vector using the following equation:

$$M_{ori} = \begin{bmatrix} p_x & p_y & p_z \\ n_x & n_y & n_z \\ l_x & l_y & l_z \end{bmatrix} \tag{15.7}$$

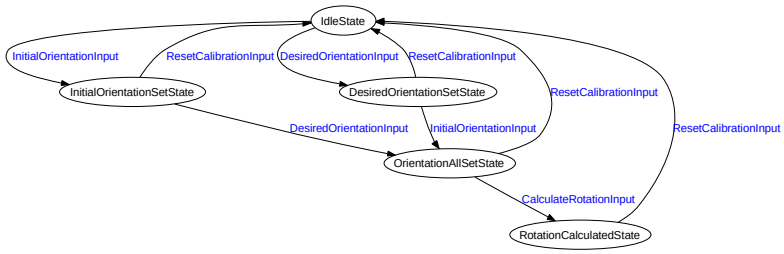

Figure 15.5: State Diagram of the PrincipalAxisCalibration class.

where (p_x, p_y, p_z) is the normalized principal axis vector, (n_x, n_y, n_z) is the normalized orthogonal view-up vector, and (l_x, l_y, l_z) is the vector along the third direction which is made by the cross production of the first two vectors.

15.2.3 State Machine Diagram

Figure 15.5 illustrates the State Machine of the `igstk::PrincipalAxisCalibration` class. There are five states inside this class: 'Idle,' 'InitialOrientationSet,' 'DesiredOrientationSet,' 'OrientationAllSet,' and 'RotationCalculated.' 'Idle' is the initial state for all IGSTK classes. Subsequently, a class's state will be 'InitialOrientationSet,' 'DesiredOrientationSet,' or 'OrientationAllSet,' depending on whether the initial or desired orientations of the tracker are put into the class. Only when a class has reached the 'OrientationAllSet' state will a request to calculate the calibration function bring the class into the final 'RotationCalculated' state.

15.2.4 Component Interface

From the description, the core functions of the igstkPrincipalAxisCalibration class include:

1. Set the initial principal axis and view-up normal;
   - igstkPrincipalAxisCalibration::RequestSetInitialOrientation();
2. Set the desired principal axis and view-up normal;
   - igstkPrincipalAxisCalibration::RequestSetDesiredOrientation();
3. Automatically adjust the plane normal to make it perpendicular with the principal axis;
4. Calculate the rotation matrix from those two orientations;
   - igstkPrincipalAxisCalibration::RequestCalculateRotation();

15.2.5 Example

The source code for this section can be found in the file
`Examples/PrincipalAxisCalibration/PrincipalAxisCalibration1.cxx`.

This example illustrates how to use IGSTK's princiapl axis calibration class to determine the rotation matrix for the tracker tools.

To use the principal axis calibration component, the header file for `igstk::PrincipalAxisCalibration` should be added.

```
#include "igstkPrincipalAxisCalibration.h"
```

After defining the headers, the main function implementation is started.

```
int main( int argc, char * argv[] )
{
```

All the necessary data types in the principal axis calibration are defined. VectorType and CovariantVectorType are used to represent the vectors along the principal axis and the plane normal.

```
  typedef igstk::PrincipalAxisCalibration   PrincipalAxisCalibrationType;

  typedef PrincipalAxisCalibrationType::VectorType          VectorType;
  typedef PrincipalAxisCalibrationType::CovariantVectorType CovariantVectorType;
  typedef itk::Logger                                       LoggerType;
  typedef itk::StdStreamLogOutput                           LogOutputType;
```

At the beginning, a principal axis calibration class is initialized as follows:

```
  PrincipalAxisCalibrationType::Pointer principal
                                     = PrincipalAxisCalibrationType::New();
```

A logger is then created for logging the process of calibration computation, and then attached to the principal axis calibration class

```
  LoggerType::Pointer                          logger = LoggerType::New();
  LogOutputType::Pointer                  logOutput = LogOutputType::New();

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

  principal->SetLogger( logger );
```

Before the computation, it is better to reset the calibration class to to remove all necessary information which may come from the previous codes.

```
principal->RequestReset();
```

Some parameters, such as axis and normal, are defined to store the input information to determine the initial and desired orientations

```
VectorType          axis;
CovariantVectorType normal;
```

An initial orientation is given as the default one for the tracker tools

```
axis[0] = 0.0;
axis[1] = 1.0;
axis[2] = 0.0;
normal[0] = 0.0;
normal[1] = 0.0;
normal[2] = 1.0;
principal->RequestSetInitialOrientation( axis, normal );
```

A desired orientation of the tracker tools are also specified

```
axis[0] = 0.0;
axis[1] = 0.0;
axis[2] = 1.0;
normal[0] = 0.0;
normal[1] = 1.0;
normal[2] = 0.3;
principal->RequestSetDesiredOrientation( axis, normal );
```

Then a RequestCalculateRotation function is invoked to compute the final results

```
principal->RequestCalculateRotation();
```

Before the final calibration transform is retrieved, the user should check the tag to see whether a valid calibration has been computed. The sample code is as follows:

```
if ( !principal->GetValidRotation())
  {
  std::cout << "No valid calibration!" << std::endl;

  return EXIT_FAILURE;
  }
```

```
else
  {
  std::cout << "Initial Principal Axis:"
            << principal->GetInitialPrincipalAxis() << std::endl;
  std::cout << "Initial Plane Normal:"
            << principal->GetInitialPlaneNormal() << std::endl;
  std::cout << "Desired Principal Axis:"
            << principal->GetDesiredPrincipalAxis() << std::endl;
  std::cout << "Desired Plane Normal:"
            << principal->GetDesiredPlaneNormal() << std::endl;
  std::cout << "Calibration Transform:"
            << principal->GetCalibrationTransform() << std::endl;

  principal->Print( std::cout);

  }
```

15.3 Calibration Data I/O

15.3.1 Data Format

IGSTK needs a common file format for storing tool calibration transformations. Because the tools must be calibrated before the surgery, it is necessary to verify that the correct calibration file is applied to the correct tool.

The tool calibration file must contain the following information:

1. The date and time that the calibration was performed (in DICOM date/time format: YYYYMMDD HHMMSS.SSSS);
2. Information about the method and the equipment used to calibrate the tool;
3. Identification information for the tool, including the manufacturer, part number, and serial number for the tool;
4. The transform type (which will be limited to rigid quaternion transforms for now);
5. The transform parameters;
6. A description of the error associated with the calibrated transform.

A sample pre-computed calibration file is like:

<?xml version=”1.0”?>

<IGSTKFile type=”ToolCalibration” version=”0.1”>

```
  <Creation date="20050824" time="070907.0705" method="PivotCalibration"
/>
  <Tool type="Pointer" manufacturer="Traxtal" partNumber="023-X" serialNum-
ber="200501268" />
  <Transform type="Rigid3D">
    <ParameterNames>
     translation_x translation_y translation_z
     quaternion_x quaternion_y quaternion_z quaternion_w
    </ParameterNames>
    <ParameterValues>
     5.0 2.0 3.0
          9.7467943448089631  -0.20519567041703082  0.9233805168766388
0.30779350562554625
    </ParameterValues>
    <ErrorParameterNames>
     rms
    </ErrorParameterNames>
    <ErrorParameterValues>
     0.187876234
    </ErrorParameterValues>
  </Transform>
<IGSTKFileCRC32>
 4f6a3b2d
</IGSTKFileCRC32>
</IGSTKFile>
```

The *CRC*32 is a 32-bit *CRC* that can be checked to validate the integrity of the data. The *CRC* is calculated from the start of the <IGSTKFile> tag to the end of the </IGSTKFile> tag, not for the whole file. A proper *DTD* is specified for the above XML file. The following error parameters could be defined:

1. *rms*: the root-mean-square error for the translation (Fiducical Registration Error for landmark registration)
2. *centroid_x*, *_y*, *_z*: the landmark centroid (or the center of rotation that was used for image registration)
3. Additional parameters to express rotational error

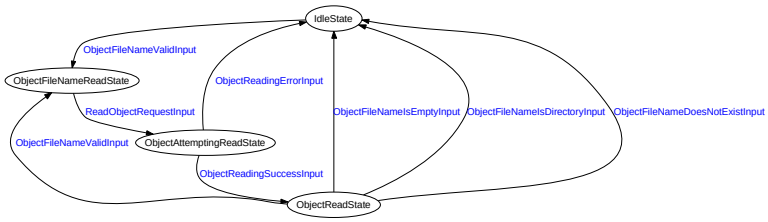

Figure 15.6: State Diagram of the PivotCalibrationReader class.

As the calibration data format is to handle all calibration information and is still improving, it is better to check the online update of the current calibration file format [4].

15.3.2 Data Reader

For the convenience of reading/writing the calibration data, IGSTK also provides some utility classes to handle the input and output of calibration data. Figure 15.6 illustrates the State Machine of the `igstk::PivotCalibrationReader` class. This class provides the function to input the calibration from the pivoting routine.

15.3.3 Example

The source code for this section can be found in the file `Examples/PivotCalibrationReader/PivotCalibrationReader1.cxx`.

This example illustrates how to use IGSTK's pivot calibration reader class to read the calibration matrix from an offline calibration file.

To use the pivot calibration reader component, the header file for `igstk::PivotCalibrationReader` should be added.

```
#include "igstkPivotCalibrationReader.h"
```

At the very beginning of the program, two kinds of event and observers are defined to track the information from the reader. The first event is igstk::CalibrationModifiedEvent, which is to retrieve the calibration class from the reader. The second one is igstk::StringEvent, which is to retrieve some string-like information from the general calibration class.

```
namespace ToolCalibrationTest
{
igstkObserverMacro(Calibration,::igstk::CalibrationModifiedEvent,
```

[4] http://public.kitware.com/IGSTKWIKI/index.php/Calibration_Data

```
                                          ::igstk::PivotCalibration::Pointer)
igstkObserverMacro(String,::igstk::StringEvent,std::string)
}
```

After defining the headers, the main function implementation is started.

```
int main( int argc, char * argv[] )
{
```

A pivot calibration reader is created and then a logger is attached.

```
  LoggerType::Pointer                       logger = LoggerType::New();
  LogOutputType::Pointer                    logOutput = LogOutputType::New();

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

  // Create the pivot calibration reader and attach the logger
  igstk::PivotCalibrationReader::Pointer reader =
                             igstk::PivotCalibrationReader::New();

  reader->SetLogger( logger );
```

The pivot calibration file's name is passed through the argument. After the filename is designated, a RequestReadObject function is invoked to parse the data file. The information in the reader can be easily dumped by default Print function.

```
  reader->RequestSetFileName(argv[1]);
  reader->RequestReadObject();

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

To retrieve the whole calibration data information, the previous defined observer is attached to the reader class. After the RequestGetCalibration function is called, the calibration info is passed by observer's GetCalibration function. The sample code is as follows.

```
  typedef ToolCalibrationTest::CalibrationObserver CalibrationObserverType;
  CalibrationObserverType::Pointer calibrationObserver
                                            = CalibrationObserverType::New();

  reader->AddObserver(::igstk::CalibrationModifiedEvent(),calibrationObserver);
  reader->RequestGetCalibration();

  igstk::PivotCalibration::Pointer calibration = NULL;
```

```
std::cout << "Testing Calibration: ";
if( calibrationObserver->GotCalibration() )
  {
  calibration = calibrationObserver->GetCalibration();
  }
else
  {
  std::cout << "No calibration!" << std::endl;
  return EXIT_FAILURE;
  }

std::cout << "[PASSED]" << std::endl;
```

To retrieve some specific information, like serial number and manufacturer, from the trackers, another string even and observer is attached to the calibration class we just get. For each request, the information content will be passed by observer's GetString function. Some sample codes are shown as below:

```
typedef ToolCalibrationTest::StringObserver StringObserverType;
StringObserverType::Pointer stringObserver = StringObserverType::New();
calibration->AddObserver( ::igstk::StringEvent(), stringObserver );

std::cout << "Testing Date: ";
calibration->RequestGetDate();
if(stringObserver->GotString())
  {
  std::cout << stringObserver->GetString().c_str() << std::endl;
  std::cout << "[PASSED]" << std::endl;
  }
else
  {
  std::cout << "No date!" << std::endl;
  return EXIT_FAILURE;
  }

std::cout << "Testing Manufacturer: ";
calibration->RequestGetToolManufacturer();
if(stringObserver->GotString())
  {
  std::cout << stringObserver->GetString().c_str() << std::endl;
  std::cout << "[PASSED]" << std::endl;
  }
else
  {
  std::cout << "No tool manufacturer!" << std::endl;
  return EXIT_FAILURE;
  }
```

```
std::cout << "Testing Serial Number: ";
calibration->RequestGetToolSerialNumber();
if(stringObserver->GotString())
  {
  std::cout << stringObserver->GetString().c_str() << std::endl;
  std::cout << "[PASSED]" << std::endl;
  }
else
  {
  std::cout << "No tool serial number!" << std::endl;
  return EXIT_FAILURE;
  }
```

15.4 Future Extension

Calibration, as well as registration, play an important role in the IGSTK toolkit. It serves like a broker between the tracker device and the physical and image space, and its role is to precisely guide the surgical tools and display them in the correct positions.

Currently igstkPivotCalibration, igstkPrincipalAxisCalibration and igstkLandmarkUltrasound-Calibration are implemented in IGSTK's main and sandbox repositories. These classes works with surgical tracker tools and a tracked ultrasound probe. The calibration data I/O classes, which provide the off-line processing of those calibration transforms, are another important part for calibration components. Definitely it covers only a small part of the calibration field. Following IGSTK's requirement-driven implementation style, some other features, such as distortion calibration for the specific electro-magnetically tracking, can be provided when it is required by the user.