cisTracker Report

Cyrus Moon
Anand Viswanathan
Senior Thesis in CIS (600.546)
Prof.: Dr. Taylor

Table of Contents

I. Introduction

I-A. Overview

A tracking device is a machine/apparatus that tracks objects in 3D space and returns "real-time" information concerning the relative position and orientation of those objects to a given reference point, specified by the device. The modality by which objects are tracked can vary, but the tracking devices currently being used use optical or magnetic technology. Tracking devices can be used in medical applications for minimally invasive surgery to supply information previously inaccessible to the surgeon. In addition, tracking devices provide superior 3D localization, where in surgery precision beyond human limitations is essential. An example of an application involving the usage of a tracking device is illustrated in Figure 1 below.

Figure 1: The Pelvic Osteolysis Project – This figure shows a simulated minimally invasive surgery where the surgeon would in practice be unable to see the area of interest on the bone.

I-B. Tracking Device Technology

Tracker types that are currently being used in this project are optical and magnetic systems.

Optical based systems utilize two or more cameras that acquire the positions of markers in space. A set of markers can be rigidly attached to each other to define a tool that can potentially contain rotational information; meaning, two differentiable markers in a fixed

orientation can define an axis (two degrees of rotation), while three can define a plane (a complete orientation).

Markers for an optical tracker can be either active or passive. Active markers strobe light that can be detected by the cameras of an optical tracker. Passive markers are detected by their reflection of environmental light. Active markers are usually required to be attached to the tracking system and require a power source, while passive markers do not have these limitations. Examples of Optical tracking devices include Northern Digital's Optotrak and Polaris as well as Image Guided Technologies' FlashPoint.

Magnetic tracking systems involve a transmitter that emits a magnetic field and tools that detect their position within the transmitted field. Most tracking systems encountered use coils to detect a point and a vector-based orientation (which only describes two or three axes of rotation) in the tracking space. Therefore tools can either return five or six degrees of freedom depending if the tools contain one or more coils. Examples of magnetic tracking systems include Northern Digital's Aurora, the Polhemus, and Ascension Technologies' Flock of Birds.

Figure 2: Tracking devices – Northern Digital's Optotrak (Left), Northern Digital's Polaris (Upper Left), and Ascension Technologies' Flock of Birds (Lower Left).

I-C. The Need for a cisTracker and a Corba class

The goal of the cisTracker library is to create a common interface for all possible tracking devices. Inheritance of the cisTracker class allows different tracking devices to be used while maintaining a common interface. This allows an application programmer use of different tracking devices without having to master each tracker's individual API or protocol; it also allows the switching of different trackers in an application without the need for substantial changes in the application code; finally, it creates a uniform template for programmers to follow when writing code for additional trackers.

On top of the tracking layer, an Operating System independent network version of the tracking libraries is desired. A Networked tracker is needed if the tracker is attached to a remote computer or if multiple applications require data from a single tracker. An example of this would be medical planning or monitoring during surgery. CORBA is a networking library that allows client/server networking using high-level encapsulation of network code.

To summarize, the cisTracker library is to be used for standalone systems where the application requiring data from the tracker is run on the same computer that interacts with the tracker. On the other hand, the cisCorbaTracker library is required when applications requiring tracking data are run on a different machine from the computer controlling the tracker.

I-D. Project Background

When this project was first initiated, the CIS libraries already contained code for a general tracker class, as well as code for use of the Polaris and Optotrak optical tracking devices. However, since no real procedure for adding trackers was defined, the libraries for Polaris and Optotrak were written independently of the base class; moreover, since the original cisTracker class was written without any tracking devices in mind, it lacked much of the essential functionality and generalization needed in a tracker base class.

II. Approach

II-A. Problem

In order to successfully create a well-organized tracker library, the essential functionality and general properties of a tracking device had to be evaluated – the different uses of a tracker, modality, nature of objects that are tracked, etc. All similarities between tracking devices (both those currently available and those that will be used in the future), were to be expressed through the base class, while all properties and functionality specific to a given tracking device were to be implemented in its derived class, and for the most part masked from the user.

II-B. Conceptual Solution

We found that all tracking devices had similar properties. Each tracking device collected positional and orientational data on physical objects relative to a coordinate system whose origin was located in or around a certain component of the tracking hardware. We also found that the operation of tracking devices included these four general phases:

1. Initialization – user-changeable parameters of the tracking device are set, and tools to be tracked are defined. This information is all sent to the tracking device, which processes it in some way.
2. Start Tracking – the tracking device is told to begin collecting position/orientation information on the previously defined tools.
3. Refresh – request for transform information is made to the tracking device, which returns the most current data.
4. Stop tracking – the tracking device is told to stop the collection of tool position/orientation information.

We therefore wanted the structure of the cisTracker library to embody these general characteristics and force any other future additions to the library to adhere to this basic structure, while also allowing easy implementation of the specifics for any given tracking system.

II-B-1. TrackerTool

Since the ultimate purpose of all tracking systems is simply to return the position and orientation of specific objects in space, regardless of what those physical tracked objects might

be, we decided that a class should be created to encapsulate this information into one coherent container. This container was to function only as a geometric container for position/orientation information returned by a given tracking device pertaining to a given tool. This abstraction insured the container's compatibility and usefulness with any tracking device implemented through cisTracker, as well as its compatibility with networked systems. Any other details pertaining to a given tracked object would be handled by the tracker class, which is implementation-specific.

When looking at tracked objects in this fashion, the following properties were deemed most relevant :

1. Current Position and Orientation
2. Status (Visibility) – whether or not a tool is currently "seen" by its respective tracking device.
3. Degrees of freedom – how much information pertaining to its local coordinate system a tool can give. This is defined for both rotation and translation with possible values between zero and three. An example of a tool with zero degrees of rotational freedom is a single, independent Optotrak LED. An example of a tool with two rotational degrees of freedom is an Aurora catheter tool, which has no defined rotation along the length of its cylindrical coil.
4. Number of markers – if the tool has markers, this is an indication of accuracy. This only applies to optical trackers.
5. Calibration Transform – if some other coordinate system static to the tool's given local coordinate system is the actual location of interest, this transform can be used to return the position/orientation of the location of interest. One example of the need for a calibration transform would be in the case of a pointer tool with a predefined local coordinate system not located at the tip of a pointer tool as shown in Figure 3. Storing the transform between the tip of the tool and the tool's original coordinate system would be useful.

Figure 3: A Tracking device tool.

6. Error – Measurement error of the positions reported by the tracker.
7. Active/Passive – whether the tool is active or passive. Active markers are usually required to be attached to the tracking system and require a power source, while passive markers do not have these limitations. Though this information is somewhat device dependent, it is required because active tools are usually more precise than passive. Active tools are also correctly identified by a tracking device, as opposed to two identical passive tools, between which an optical tracking device cannot distinguish (e.g. two individual passive markers).

Other information, such as the actual spatial orientation of markers on a tool and the orientation/position of the origin with respect to the physical tool object, was seen to be not useful to any given application requiring information from a tracking device.

II-B-2. Tracker

We want the use of the tracker class to follow this general protocol:

1. A set of configuration files (detailed in the following section) is read in order to define a tracker and its tools. This information is then used to initialize the tracking device.
2. The tracking device begins the acquisition of data.
3. Tool information is requested from the tracking device when desired (which is usually continuously).
4. The tracking device stops the acquisition of data.

The goal of the tracker library is to have all the general methods, which would be required by all types of trackers to be executed by one general class. This general class would enforce a consistent API for all trackers as well as reduce the amount of code that would have to be written for every tracking system. A more detailed outline of the protocol follows:

I. Configuration/Initialization

1. Base Class – A primary configuration file is opened, and the names of files containing more specific information for the particular application and tracking device along with general parameters is extracted. Memory is subsequently allocated for tool objects.
2. Derived Class – Based on the information contained in the primary configuration file, the tracker specific files would be loaded. The information is used to initialize the tracker system and its tools.

II. Tracking Initiation

1. Derived Class – Method calls are made to the tracking device's vendor-provided API in order to initiate the tracking of the tools.
2. Base Class – If the derived method executes without error, the base class will change the tracker status value to indicate that tracking has started.

III. Data Collection

1. Derived Class – Method calls are made to the tracking device's vendor-provided API in order to retrieve the current position and orientation information.

IV. Tracking Termination

1. Derived Class – Method calls are made to the tracking device's vendor-provided API in order to terminate the tracking of the tools.
2. Base Class – If the derived method executes without error, the base class will change the tracker status value to indicate that tracking has stopped.

All key methods in derived classes are derived from base class methods, which allows users working with an implementation class through a pointer to the base class. For information on the implemented protocol please see Section III-B.

II-B-3. Tracker Configuration Files

We decided that tracker configuration and tool information were to be inputted to the tracker library through configuration files. Doing this prevents the need for specific tracker and tracker tool parameters to be encoded directly in a tracker application, or inputted by the user at application run time.

A hierarchy of configuration files was created to allow the information from the first, primary file to be read in by the base class only; this file mostly contains generic information about the tracker, as well as the names of files containing more tracker-specific information. Initialization of the tracking device and the appropriate tools would then be done by implementation-specific methods taking in the more specific configuration file names as parameters. Tracker information is thus split into three different kinds of configuration files:

1. General configuration file – the reading of this configuration file was made to be independent of the type of tracking device it was created for, allowing it to be read by the base class. This file contains the following:
    - `TRACKER_NAME` – a user-defined name for the tracking device / tool setup defined by the configuration file. Potentially for debugging purposes and use in GUI applications.
    - `TRACKER_TYPE` – the actual name of the tracker (e.g. Flock of Birds).
    - `TRACKER_CONFIG_FILE` – the name of the file containing parameters needed to initialize the tracking device.
    - `NUM_TOOLS` – how many tools are to be tracked.
    - `TOOL` – the names of the tool configuration files (containing parameters needed to initialize the tools).
2. Tracker hardware configuration file – the structure of this file is specific to the tracking device in use. The reading of this file therefore must be implementation specific.

3. Tool parameter files – the structure of this file is also specific to the tracking device being used. However, there are some parameters that are required to be present in all tool configuration files, regardless of tracker type. They are the following:
    - `ID` – A user defined name for the tool.
    - `NUM_MARKERS` – The number of markers comprising the tool.
    - `ACTIVE` – Whether the tool is active or passive. See Section I-B for the definition of active/passive.
    - `TRANSLATION_DOF` – The number of translation degrees of freedom the tool is capable of returning. See Section I-B for the definition of degrees of freedom.
    - `ROTATION_DOF` – The number of rotation degrees of freedom the tool is capable of returning. See Section I-B for the definition of degrees of freedom.
    - `CALIBRATED_TRANSLATION` (optional) – defines the translation component of the static transformation from the desired frame to the tool frame. See description of the data member `CalibrationTransform` in section III-A for details.
    - `CALIBRATED_ROTATION` (optional) – defines the rotation component (in quaternion format) of the static transformation from the desired frame to the tool frame. See description of the data member `CalibrationTransform` in section III-A for details.

    -The reading of tool files is implementation specific.

An example of a primary configuration file follows:

```
TRACKER_NAME            MyTrackerName
TRACKER_TYPE            polaris
TRACKER_CONFIG_FILE     PolarisSetup.txt
NUM_TOOLS                    2
TOOL    1                    PolarisActiveTool.txt
TOOL    2                    PolarisActiveTool2.txt
```

An example of a tracker-specific hardware configuration file:

```
BAUD_RATE                    115200
SERIAL_PORT                  1
HAS_STRAY_PASSIVE_MARKERS    no
```

For more details of these Polaris specific parameters see Section III-C.

An example of a tool-specific hardware configuration file:

```
ID                      ActiveTool
NUM_MARKERS        4
PORT                    1
ACTIVE                  yes
STATIC                  no
TRANSLATION_DOF         3
ROTATION_DOF            3
```

For more details of these Polaris specific parameters see Section III-C.

III. Implementation

III-A. cisTrackerTool

Objects tracked by tracking devices ('tracker tools'), all have many identical attributes. Each has a current position and orientation (which is of course the main function of a tracking device); a given tool can also be 'active' or 'passive'; furthermore, since optical trackers are limited to tracking individual points which have only a position, many tools have a given number of 'markers' (the objects that are actually being tracked by the device), which are used to determine the orientation of the object. Therefore a 'tracker tool' container class (cisTrackerTool) was created for convenient storage of essential attributes of a given tool, as well as the tool's most current relevant transformation data.

III-A-1. Data Members – each cisTrackerTool object contains the following:

1. `cisVec3 RawPosition` – The position of the tool coordinate system in tracker space
2. `cisUnitQuat RawOrientation` – Orientation of the tool coordinate system in tracker space (quaternion format)

   -- Position and orientation are stored separately for reasons of efficiency. Some application might require only positional data, or a tool may consist of one marker (and thus not have a rotation), therefore cisFrame need not be created. Rotation is stored as a quaternion because most tracker vendor API's return rotations in quaternion format; also, having rotation stored in four terms (as opposed to nine in an Euler matrix) allows for faster writing/reading and sending/receiving in the tool object. Quaternion format was found to be an overall better solution over the Euler format. Conversion methods are provided for the user if different transformation representations are needed.

3. `cisFrame CalibrationTransform`– Static transformation between a desired coordinate system and the tool coordinate system.

   -- At times the user might desire information on a coordinate system that is static relative to the coordinate system returned by the tracking device (for example, if a tracking tool is attached to a pointer, then one would probably want to track the tip and orientation of the pointer, rather than that of actual tracker tool itself). This member was added for such a case; one can specify the constant transformation from the desired frame to tracker tool's frame here; setting a 'HasCalibrationTransform' flag to true will then cause all methods returning transformation data to output the location of the *desired* frame in tracker space, found by transforming the calibration transform with the most recent frame data on the tool itself:

   $$T_{Desired \leftarrow Tool} * T_{Tool \leftarrow Tracker} = T_{Desired \leftarrow Tracker}$$

   Whenever data is refreshed for a tool object containing a calibration transform, this transform is applied to the raw data acquired from the tracking device, and the resulting transformation is stored in the object. Though this means that the raw data for a given tool is not readily available, this pre-calculation prevents transform calculations to be performed by the application, and also minimizes the size of the tool object. The raw data can be found by multiplying a given position/orientation with the inverse of its calibration transform. Of course, if raw data is required for a given tool, then the application will suffer from slowdown due to the unnecessary calculations of $T_{Desired}$. Any user of the cisTracker library must be aware of this stipulation in cisTrackerTool.

4. `cisBool HasCalibration`– Indicates whether a frame relative to the actual tool frame is to be tracked

   -- See CalibrationTransform description above (#3).

5. `cisScalar Error` – Error value for current transformation data returned from the tracking device

   -- This value currently indicates positional root-mean-squared error, in millimeters.

6. `cisChar* ToolId` – Name of the tool (user defined); useful for debugging and GUI applications.
7. `cisChar* ConfigFileName` – Name of the configuration file containing tool information needed by the tracker

   -- The string member ToolId serves as a user-defined identifier for the tool object, which facilitates more readable messages when a specific tool object must be identified. String member ConfigFileName stores the name of the configuration file that was used by the tracker to create the tool object.

8. `cisInt Status` – Current status of the tool

   --The possible statuses for a tracker tool, in sequential order, are:

   1. `cisTrackerTool_INVALID` – Tool definition was found to be invalid (e.g. the parameters in a given tool configuration file were on correct)
   2. `cisTrackerTool_NOT_AVAILABLE` – Tool definition is valid, but the tool is either missing or cannot be found by the tracker (e.g. the tool specified is not plugged into the tracking device)
   3. `cisTrackerTool_AVAILABLE` – Tool is valid, and recognized by the tracker
   4. `cisTrackerTool_INITIALIZED` – Tool is initialized (ready to be tracked)
   5. `cisTrackerTool_TRACKING` – Tool is currently being tracked (though not visible)
   6. `cisTrackerTool_VISIBLE` – Tool is currently being tracked and is visible to tracker

   Evaluation of tracker tools showed that general tool status could be sufficiently represented in successive stages, and bit representation was not necessary. This allows the statuses to be used with comparison symbols (e.g. if (Status < `cisTrackerTool_AVAILABLE`) ✍ tool cannot be initialized by the tracking device).

9. `cisBool IsActive` – Indicates whether the tool is active or passive

   -- See Section I-B for a description of the active/passive tool attribute.

10. `cisInt NumMarkers` – Number of positional markers on the tool

    -- See TrackerTool introduction paragraph for the definition of a tool ‘marker’.

11. `cisUShort NumberOfTranslationDOF`– Number of translational degrees of freedom the tool has (i.e. what is known given the geometry of the tool)
12. `cisUShort NumberOfRotationDOF`– Number of rotational degrees of freedom the tool has (i.e. what is known given the geometry of the tool)
    -- See Section II-B-1 for a description of translational and rotational degrees of freedom. Most tracked objects will have all six degrees of freedom (three translational, three rotational).

13. `cisLong TimeStamp` – Time at which the current transformation data held was received
    -- Since most tracking applications will be real-time, it is essential to maintain some time reference for each tool transform returned by the tracking device. Therefore a time variable was created to store the time at which the application requested and received the transform information currently being held by the tool object. The time variable is of type cisLong, due to the fact that the time reference implemented by cisTime is of this format.

III-A-2. Methods – Methods of the cisTrackerTool class generally consist of 'set' and 'get' methods that allow users access to a tool object's data members. In 'set' methods the parameters are all made `const` while all 'get' methods are `const` methods, in order to ensure that any data that should not be overwritten is not changed. Most set/get methods pertain to individual data members, with the exception of the following methods that are of particular interest:

```
1. void SetFrame (const cisVec3 newPosition,
                  const cisUnitQuat newOrientation,
                  const cisInt newStatus,
                  const cisULong newTimeStamp);
2. void SetFrame (const cisFrame newFrame,
                  const cisInt newStatus,
                  const cisULong newTimeStamp);
```

-- These set methods are used to set the current position and orientation of a given tool object, along with its related timestamp and status after a refresh call to a tracking device. The method is overloaded so two different representations of a transform can be used as parameters. The setting of all dynamic data in a cisTrackerTool object was chosen over individual set methods in order to facilitate multithreading, as well as maintain consistency of data within the tool object (e.g. one

implementation of cisTracker accidentally omits the setting of a new timestamp during a refresh method, or does not set a new status with every writing of new transform data).

```
3. void SetStatus (const cisInt newStatus,
                   const cisULong newTimeStamp);
```

-- This set method is used to set a new tool status when no new frame data is related to the status change. The most notable example of this is when a tool is not visible to a tracking device, and the status of the tool must be changed to indicate this. Other examples include tool status changes during the initialization process. Including this method also allows the tool's most recent valid position and orientation to be preserved, which would prove useful in many applications.

```
4. void GetFrame(cisVec3& newPosition,
                 cisUnitQuat& newOrientation,
                 cisInt& newStatus,
                 cisULong& newTimeStamp) const;
5. void GetFrame(cisFrame& newFrame,
                 cisInt& newStatus,
                 cisULong& newTimeStamp) const;
```

-- These methods are used to return (by reference, to allow the return of multiple data members), all dynamic data in a given tool object. The method is also overloaded like its corresponding 'set' method so two different representations of a transform can be used as parameters. The retrieval of multiple data members was again chosen over individual retrieval methods to facilitate multithreading and maintain consistency of data within the tool object.

III-B. cisTracker

This generic tracker class defines a basic outline of what actions tracking devices should be able to perform. The class is neither tracking device nor operating system dependent. Classes derived from this general class must be tailored to the hardware specifications of a specific tracker. This library is to be used for standalone systems, where the application using the tracker library is run on the same computer that the tracking device is physically attached to.

The following changes reflect the decision based on previous implementation of cisTracker and code review meetings:

Functions that are implementation specific are virtual to indicate the need for the creation of derived methods. Functions that change tracker status are virtual; the tracker status is only changed in the base class (a derived method returns result of its corresponding base class method). Implementation-specific methods that do not result in status changes are pure virtual to force their creation in implementation classes. Functions that could be generalized are not virtual.

III-B-1. Data Members

1. `vector<cisTrackerTool*> Tools`– STL Vector holding the cisTrackerTool objects
   -- The STL vector class is being used to store cisTrackerTool objects (previously described). This replaces the previous cisSet container implementation that was not Operating System friendly, and eliminates much unnecessary code dealing with ToolSets and ToolSetSpecs, classes that existed in the previous cisTracker. The advantage of using the Vector class is that the STL Vector handles its own memory allocation and deallocation. To limit the memory space used by the tools Vector, instead of increasing the size dynamically when the Vector is full, the `reserve` function of the Vector class is called to increase the capacity of the Vector by 1. When adding to the Vector, Null is pushed at the first open slot using the `push_back()` function call then replaced by a reference to a cisTrackerTool Object. When a Reset is called on a tracker object, the tool Vector is emptied of cisTrackerTools by the STL vector's `clear()` method.

2. `vector<char**> ToolFileList`– STL Vector holding the filenames of TrackerTool configuration files.
   -- This vector holds the configuration file of each TrackerTool object to be used in the application. This list is filled by the `LoadConfiguration` method. The name of each tool file is loaded into the vector, to be used in the `InitializeTools` method to load the necessary information contained in each tool configuration file. These tool files are not files

provided by the manufacturer of the given tracker tool (e.g. a Polaris passive tool '.rom' file); rather, these files are user defined. If a manufacturer-provided file is required for the tool, the file name will be given inside the TrackerTool configuration file. This Vector also increases its size dynamically when the Vector is full, the `reserve` function of the Vector class is called to increase the capacity of the Vector by 1.

3. `cisTrackerStatus Status` – Status variable (`typedef cisInt cisTrackerStatus`) that contains the current state of the tracker.

   -- This variable holds the current status of the tracker; it replaces the cisTrackerStatus class, as having a class for the sole purpose of holding tracker status was found to be unnecessary. A bit-implementation of status was also found to be unnecessary, as possible tracker statuses progress in an incremental fashion. Below are the possible tracker statuses:

   - `cisTracker_INVALID` – Tracker is invalid (e.g. the configuration file contained invalid parameters)
   - `cisTracker_VALID` – Tracker is created but unavailable (e.g. the tracker is not connected to the host computer)
   - `cisTracker_AVAILABLE` – Tracker is available but not initialized (i.e. parameters specified by the configuration file have not been inputted to the tracking device yet)
   - `cisTracker_INITIALIZED` – Tracker is initialized, but tools are not yet initialized
   - `cisTracker_VALID_TOOLS` – Tracker and tools are initialized but tracker is not tracking
   - `cisTracker_TRACKING` – Tracker & Tools are initialized and the tracking device is collecting position/orientation information on cisTrackerTools contained in the tools Vector.

   Evaluation of trackers showed that general tracker status could be sufficiently represented in successive stages, and bit representation was not necessary. This allows the statuses to be used with comparison symbols (e.g. if (Status < `cisTracker_INITIALIZED`) ✍ tracker was not initialized.

4. `int NumberOfTools` – Contains the Number Of Tools loaded from the Tool Configuration files

   -- This variable is initialized to zero in the constructor of the object. Then is incremented by 1 for each Tool that is loaded in the `InitializeTools` method.

5. `char* TrackerName` – Contains a user-defined I.D. for the tracker object

   -- Loaded from the Tracker Configuration File in `LoadConfiguration` method.

6. `char* TrackerType` – String containing the name of the type of tracking device (e.g. Polaris, Optotrak, Flashpoint)
   -- Used in the `CheckType` method to confirm whether the current implementation (derived classes) corresponds to the correct TrackerType.

7. `char* TrackerFile` – Contains the name of the Tracker Hardware Setup File
   -- `TrackerFile` is read from the general Tracker Configuration File during the `LoadConfiguration` method. The derived class of TrackerVirtual uses this information during the `Initialize` method to open the appropriate tracker hardware configuration file.

III-B-2. Methods

1. `cisBool LoadConfiguration(const cisChar* config_filename_in)` – A hierarchy of configuration files is used for all information that had to be given to a tracker (tool info, tracker setup parameters). The primary configuration file contains no tracker-specific information and is thus read by the base class' `LoadConfiguration` method. Files obtained from this method are used to initialize the tracker and its tools by the `Initialize` and `InitializeTools` methods from the names of the filenames for tracker- specific hardware configuration files tracker.

2. `cisBool StartTracking(void)` – Initializes the tracker to start the tracking of the tools. This sets the status bits of the tracker to `cisTracker_TRACKING` and the status bits of the tracked tools are set to `cisTrackerTool_TRACKING.`

3. `cisBool StopTracking(void)` – Instructs the tracker to terminate the tracking of the tools. This sets the status bits of the tracker to `cisTracker_VALID_TOOLS` and the status bits of the tracked tools are set to `cisTrackerTool_INITIALIZED.`

4. `cisTrackerStatus GetStatus(void)` – A base class method that returns the current status of the tracker. Can be used to test whether the tracker is in the proper state. The Status is represented by `cisTrackerStatus` described above.

5. `cisBool GetTool(const cisUInt index,cisTrackerTool& ToolOut) const`– A base class method in which a tool object is passed into the method and then the requested index in the tools vector is returned.

6. `cisBool GetTools(vector<cisTrackerTool*>& ToolsOut)`– A base class method in which a tool vector object is passed into the method and then a copy of the current Tool Vector is returned.

7. `cisInt GetNumberOfTools(void)`– Returns the `NumberOfTools` class member. This value represents the total number of tools in the current application as given from the configuration files.

8. `cisBool CheckType(void)` – A base class method that checks the current tracker type of the specific tracker object trying to be created by calling the `GetType` method versus the `TrackerType` member read in from the primary configuration file.

9. `cisTrackerTool* InitializeTool(const cisChar* ToolConfigFile)`– This initializes a newly created cisTrackerTool object to the specifications in the tool configuration file acquired from the primary configuration file. This is defined as a pure virtual method since configuring a tool would be based on the type of tracking system used.

10. `cisBool InitializeTools(void)`– A base class method that initializes the Tools Vector by creating cisTrackerTool objects, then initializing the newly created tool objects with the `InitializeTool` method. Finally, the configured tool is added to the end of the vector using the `push_back` STL-vector function call. This is not implementation specific since all tracking devices will need to create and initialize the vector of tools.

11. `cisBool Initialize(void)`– A virtual method that in the base class sets the status of the tracker to `cisTracker_INITIALIZED`. The derived class version reads in tracker parameters from the hardware setup file. This information is then used to configure the tracking device. If no errors occur during the initialization, then the base class method is returned in order to set the status of the tracker.

12. `cisBool Refresh(const cisUInt ToolIndex)` – A pure virtual method that updates an individual tool's position and orientation data. Checks if that tool is visible or not and sets the appropriate status for the tool.

13. `cisBool Refresh(void)` – A pure virtual method that updates all the positions tools in the cisTrackerTool vector 's position and orientation data. Checks if the tools are visible or not and sets the appropriate statuses for the tools.

14. `cisBool Reset(void)` – A virtual method in which the base class calls a similar sequence of method calls as the cisTracker constructor. This method is designed to reinitialize both the tracker and its associated tools. The derived method would perform reinitialization steps that are tracker-specific and then return the `Reset` method for the base class.

15. `cisChar* GetType(void)` – A pure virtual method that the derived class returns the tracker-specific implementation type.

III-C. cisTrackerPolaris

The Polaris is an optical tracking device that communicates with its host computer through a serial connection. It has the ability to track both active and passive tools, as well as returning positional information on all individual passive markers visible to the detector (referred to as passive stray markers).

This implementation class handles passive stray markers by the addition of an extra flag parameter to the tracker-specific hardware configuration file. Setting this flag to true will result in the creation of 50 extra cisTrackerTool objects, which added to the end of the `Tool` vector after the tools specified in the general configuration file are pushed. During a refresh, these 50 objects will be filled in order (from `NumberOfTools` to `NumberOfTools` + (the number of visible passive stray markers)), according to the number of individual passive markers the Polaris camera sees. The user must keep in mind that each of the 50 passive-stray-marker tool objects are in no way associated with individual markers from one refresh to the next – if marker A's position was stored in object 1 and marker B's position was stored in object 2 for one refresh, the next refresh might have marker A's position stored in object 2, and B's position in object 1. Furthermore, Polaris may count markers comprising a defined passive *tool* as stray passive markers, and return those positions along with independent passive markers. An application

programmer can determine how many passive stray markers were seen for a given refresh by searching through the passive-stray-marker tool objects in the vector until reaching the first one with a `cisTrackerTool_TRACKING`(not visible) tool status.

III-C-1. Configuration Files

Tracking device parameters that are specific to Polaris are the following:

- `BAUD_RATE` – the desired speed at which the host computer will communicate with the tracking device. Valid baud rates for Polaris are 9600, 14400, 19200, 38400, 57600, and 115200 bps.
- `SERIAL_PORT` – the port number on the host computer to which the tracking device is connected.
- `HAS_STRAY_PASSIVE_MARKERS`– since the indiscriminate tracking of passive markers is unique to Polaris, this parameter had to be implemented through the tracking device configuration file. Setting this flag to 'true' will append 50 new identical tool objects to the end of the list of tools specified in the general configuration file (see class member `HasStrayMarkers` below for more details).

An example of a Polaris hardware setup file would then be:

```
BAUD_RATE                 115200
SERIAL_PORT               1
HAS_STRAY_PASSIVE_MARKERS yes
```

Tracker tool parameters that are specific to Polaris are the following:

- `PORT` – each tool must have a port number associated with it. If the tool is active, then the port number specified must be the port that the tool is physically connected to (currently ports 1-3, on the front face of the Polaris controller box). If the tool is passive, then it should be assigned a port number ranging from 4 to 12. Any number in this range is valid, as long as no other passive tool being used is assigned the same port number.
- `STATIC` – Polaris tracks 'static' and 'dynamic' objects differently. A static tool is considered to be relatively immobile (e.g. reference emitter) whereas a dynamic tool is considered to be in motion (e.g. probe). This tool characteristic must therefore be specified.
- `ROM_FILE` – Polaris uses .rom files to define the specifics of tools (e.g. relative position of markers on the tool, location/orientation of the local coordinate system's origin, etc.). While active tools may not need an accompanying .rom file (some can be identified by

Polaris hardware through its physical connection), all passive tool configuration files must specify a .rom file.

An example of a Polaris tool configuration file would then be:

```
ID                  PassiveTool
NUM_MARKERS         4
PORT                4
ROM_FILE            ta002-4.rom
ACTIVE              no
STATIC              no
TRANSLATION_DOF     3
ROTATION_DOF        3
```

-See section II-B-3 for information on non-tracker-specific parameters.

III-C-1. Class Members

Implementation-specific changes include the addition of the following:

1. `cisSerialPort* SerialPort` – A cisSer object for communication between host and Polaris

   -- Since Polaris uses a serial connection, the CIS implementation of the serial protocol is used.

2. `cisChar PortNames[12]` – Stores tool port names as recognized by Polaris

   -- When command strings are sent to Polaris, its 3 active and 9 passive tool ports are referred to as ports '1-3' and 'A-I', respectively. This array is used simply to convert port #'s given in tool configuration files into the appropriate port names to be transmitted to Polaris.

3. `cisChar SerialPortString[20]` – String containing name of the communications port the host computer is using to connect to Polaris

   -- This string variable is used by the cisSer object `SerialPort`.

4. `cisLong BaudRate` – connection speed between Polaris and the host computer

   -- See 'Configuration Files' section above for more details on Polaris baud rate.

5. `cisInt Port2ToolList[12]` – conversion table between tool port numbers and their respective index in the `Tools` vector.

   -- Since Polaris uses a single string to return all tool transformation data ordered by port number, this array was created in order to efficiently fill each tool object in the `Tools` vector

with their respective new transformation data. The array contains 12 elements (since Polaris uses a maximum of 12 ports), and each element in `Port2ToolList` contains the index of the appropriate tool in the `Tools` vector; so if Tool #2 uses port 3 on Polaris, then `Port2ToolList[2]` (3 minus 1) will equal 1 (2 minus 1). If no tool exists in a particular port, that element in `Port2ToolList` will equal -1.

6. `cisBool HasPassiveTools` – flag indicating whether passive tools are being used or not

7. `cisBool IsPassiveExtended` – flag indicating whether passive tools are present in ports D–I

   -- This flag is present due to the fact that parameters for the refresh command sent to Polaris may be different depending on whether more than 3 passive tools are being used (or if for some reason the user wishes to declare his/her tools as occupying higher number tool ports).

8. `cisBool HasStrayMarkers` – indicates whether Polaris is to return data on any visible individual passive markers with each refresh command.

   -- Whether or not Polaris is tracking stray individual passive markers or not also affects the parameters of the refresh command sent to the tracking device, therefore this flag variable is necessary. The setting of this flag to true will also result in the creation of 50 empty tool objects onto the `Tools` vector, following the creation of the tools specified in the general configuration file. These 50 objects are used to hold any stray passive marker positional information returned by Polaris.

9. `cisUInt FirstPassiveStrayMarkerID` – stores the `Tools` array index of the first passive stray marker tool object

III-D. CisTrackerOptotrak

The cisTrackerOptotrak is a tracker implementation of the cisTracker class for the Northern Digital Optotrak tracking device. This Optotrak class tracks loose LED markers, rigid bodies obtained from Northern Digital and user defined rigid bodies. These markers are flashing IR leds, which are controlled by strober units. The rigid bodies are markers, which are in a permanently fixed relative orientation with each other. The description of actual spacing of the markers in the Rigid Body is contained in its own Rigid Body Description File.

The system control unit of the Optotrak communicates to the host computer using an ISA card or a SCSI device. This cisTrackerOptotrak class relies on the Northern Digital library (oapi.lib) for low-level API function calls to the Optotrak's system control unit

? Update by Anton Deguet: As of May 22, 2002 mkProj is used to create the workspaces and projects for CIS, including cisTracker. To compile cisTrackerOptotrak, the compiler must have the options `/D CIS_OPTOTRAK /D __WINDOWS_H /I <include_path>` and the linker must have the option `<lib_path>/oapi.lib`. See the mkProj documentation for more details.

The Optotrak API Library Function calls return a bool. If this bool is false then the API places an error message into the `char* ErrorString` variable. The tool configuration files of individual loose markers should indicate to the cisTrackerOptotrak class that this loose marker is to be tracked for its positional information only, and the rotational information will always be set to the identity transformation.

The `Refresh` method calls the Optotrak API function call of `DataGetLatest3D` to obtain the 3D positions of all of the markers connected to the strober daisy chain. The markers are flashed in a sequential order determined by their order in the strober daisy chain. The `Refresh` method then calls the Optotrak API function `DataGetLatestTransforms` to obtain the latest 6D information for all the rigid bodies that were added to the Optotrak's tracking list during the `InitializeTool` method. The 3D and 6D information are stored separately into two Northern Digital defined data structures. This needed data is then set for each cisTrackerTool object in the `Tools` vector. The rigid body configuration file gives the absolute positions in 3D space of the markers on the rigid body with respect to each other. This spatial information is used by the Optotrak when detecting the led markers during tracking to compare between the root mean squared error values in the positions of the leds to the 3D maximum error value given in the rigid body configuration file. If the calculated error value is greater than the maximum allowed then the positional data returned is considered out of range and the tool's position is not updated but their status is set to `cisTrackerTool_TRACKING` to indicate that the tool is not visible. If the calculated error value is less than the maximum allowed error value then the tool positions and calculated orientations are stored into the `Tools` vector.

III-D-1. Configuration Files

Tracking device parameters that are specific to Optotrak are the following:

- NUMBER_OF_MARKERS – the Optotrak requires the total number of markers for initialization. This number is necessary in initialize.
- MARKER_FREQUENCY – This is the Marker Frequency for maximum on-time used in the Optotrak API function `OptotrakSetupCollection`. This parameter is used for the `StartTracking` method.
- SAMPLING_FREQUENCY – Frequency at which to collect data frames. This is another parameter used for the `StartTracking` method.

An example of an Optotrak hardware setup file would then be:

```
NUMBER_OF_MARKERS  6
MARKER_FREQUENCY   2500.0
SAMPLING_FREQUENCY 100
```

Tracker tool parameters that are specific to Optotrak are the following:

- RIGID_BODY – This flag indicates whether the tool is a single led, a user defined rigid body or a Northern Digital defined rigid body. If the Rigid_Body flag is set to YES then this is a Northern Digital-defined rigid body and the Optotrak vendor API `RigidBodyAddFromFile` function is called. If the flag is set to NO but there is more than 1 marker then the `LoadRBody` method is called to load a user-defined rigid body onto the Optotrak's tracking list. If the flag is set to NO and there is only one marker then the tool object is considered a single led.
- `RIGID_BODY_FILE` – This is the name of the .rig file where the Optotrak loads pertinent tool information in order to compute a rotation and translation for the rigid body.

An example of an Optotrak tool configuration file would then be:

```
RIGID_BODY                yes
NUMBER_OF_MARKERS         6
RIGID_BODY_FILE           Rb06073
ID                        CalibratedPointer
NUMBER_OF_TRANSLATION_DOF 3
NUMBER_OF_ROTATION_DOF    3
```

-See section II-B-3 for information on non-tracker-specific parameters.

Class Members – Implementation-specific changes include the addition of the following:

1. `cisUInt SingleLedToOptoLed[cisTrackerOptotrak_MAX_MARKERS], cisUInt SingleLedToTrkTool[cisTrackerOptotrak_MAX_MARKERS]` – The information for the order of LED's in the strober daisy chain and the `Tools` vector is stored in the `SingleLedToOptoLed[]` and the `SingleLedToTrkTool[]` arrays; These arrays were created in order to decrease the time during the `Refresh` method for searching and filling both the marker positions and the rigid body transformations and translations into the appropriate cisTrackerTool object in the `Tools` vector. The arrays of `SingleLedToOptoLed` and `SingleLedToTrkTool` are parallel arrays that link the index positions in the `Tools` vector of a single led tool to its corresponding marker position in the Optotrak strober daisy chain.
2. `cisUInt RigidBodyToTrkTool[cisTrackerOptotrak_MAX_NUMBER_OF_RIGID_BODIES]` – The relation between the `cisTrackerTool` vector and RigidBody order is stored in the `RigidBodyToTrkTool[]` array. These arrays were created in order to decrease the time during the `Refresh` method for searching and filling both the marker positions and the rigid body transformations and translations into the appropriate cisTrackerTool object in the `Tools` vector. The `RigidBodyToTrkTool` array contains the index positions in the `Tools` vector of rigid bodies led tool, in order to facilitate the updating of 6D information for rigid bodies only.
3. `char ErrorString[MAX_ERROR_STRING_LENGTH + 1]` – This is a string to contain any errors returned by the Optotrak vendor API. `MAX_ERROR_STRING_LENGTH` is an Optotrak API defined variable that indicates the size of the error string returned by the Optotrak.
4. `RigidBodyDataType RigidBodyData` – A Northern Digital defined typedef Struct that holds the data for vendor API function calls that refresh marker positions.
5. `int NumberOfMarkers` - The total number of markers in the application. This is read in from the hardware setup file during the `Initialize` method. This number is of the data collection parameters needed to initialize the Optotrak.
6. `float SamplingFrequency` – The frequency at which to collect data frames. This is read in from the hardware setup file.
7. `float MarkerFrequency` - The marker frequency for maximum on-time. This is read in from the hardware setup file.
8. `int NumberOfRigidBodies` – Indicates the total number of rigid bodies in the application.

The following data members are iterators only used for the initialization of the tools in the `InitializeTool` method:

1. `int CurrentRigidBodyID` – Indicates the current rigid body being loaded into the Optotrak's tracking list.
2. `int CurrentMarkerPosition`– The current marker position. This is used for the tool conversion arrays.
3. `int CurrentTrkToolID-` Indicates the current position in the array for all 3 tool conversion arrays.
4. `int CurrentMarkerID-` Indicates the current number of single leds that have been loaded as tools.

III-E. cisTrackerFactory

cisTrackerFactory is a class created by Cyrill von Tiesenhausen of Brigham & Women's Hospital that allows the writing of applications that contain no tracker specific code. This allows an application to be run with any given tracking device without the need for any modifications within the code.

The `createTracker` method of this class takes in the name of a tracker general configuration file as its parameter. This allows the method to read the file and determine the type of tracker specified. The method then creates the appropriate derived class object and returns a pointer to that object.

III-F. Utilization

III-F-1. Usage

1. The user calls the constructor of a derived Tracker class, which executes three main cisTracker methods, in the following order:
   1. `LoadConfiguration` (Base class method)
   2. `Initialize` (Derived class method)
   3. `InitializeTools` (Base class method)

   The first method, `LoadConfiguration`, reads in and stores the names of the tracker hardware configuration file and tool configuration files. If this method executes successfully, a value of `true` is returned, which allows the constructor to next execute the `Initialize` method.

   The `Initialize` method opens the tracker hardware configuration file, reads in tracker specific parameter values, then sends the appropriate initialization commands to the actual tracking device. If this method executes successfully, a value of `true` is returned, allowing the constructor to next execute the `InitializeTools` method.

   The `InitializeTools` method essentially runs a loop (with number of iterations equaling the size of the `ToolFileList` vector) that creates and adds cisTrackerTool objects to the `Tools` vector by way of the derived class method `InitializeTool`. This method also sends appropriate commands to the tracking device in order to initialize the actual tracking tool hardware. If this method executes successfully, a value of `true` is returned, and the constructor successfully creates the object. At this point, the `StartTracking` method can be called.
2. The user calls the derived class method `StartTracking`, which sends the appropriate commands to the tracking device to initiate its collection of data.
3. While the tracking device is collecting data, the user calls the derived class method `Refresh`, as many times as desired, in order to update current position and rotation data for the tracking tools represented by objects in the `Tools` vector.
4. The user calls the derived class method `StopTracking`, which sends commands to the tracking device to stop its collection of data.
5. **The user can call the derived class method `Reset` at any time to re-initialize the tracking device and its tools. The `StartTracking` method can be called immediately after a successful `Reset`.

The virtual class methods `Initialize`, `StopTracking` and `StartTracking` serve only to facilitate changes in tracker status; each implementation of cisTracker must re-implement these methods to send the appropriate signals or commands to the tracking device. Each method re-implementation should at some point call the base class method to bring about changes in status.

III-F-2. Test Program

```
int main(int argc, char* argv[])
{
  if (argc != 2) {
    cout << "Usage: " << argv[0] << " <configfile>" << endl;
    return 1;
  }

  cisTrackerFactory trackerFactory;
  cisTracker* TestTracker = trackerFactory.createTracker(argv[1]);

  if (!TestTracker) {
    cout << "Could not create Tracker" << endl;
    return 1;
  }
  if (TestTracker->GetStatus() < cisTracker_VALID_TOOLS) {
    cout << "Error during initialization" << endl;
    return 1;
  }

  cisUInt nbTools, iter;
  cisUInt currentToolIndex;
  cisTrackerTool tool;
  cisVec3 toolPosition;
  cisUnitQuat toolOrientation;
  cisInt toolStatus;
  cisULong toolTimeStamp;
  nbTools = TestTracker->GetNumberOfTools();

  cout << "Number of tools: " << nbTools << endl;

  TestTracker->StartTracking();

  for (iter = 0; iter < 100; iter++) {
    if (!TestTracker->Refresh()) {
      cout << "At least one tool is not visible" << endl;
    } else {
      cout << "All tools are visible" << endl;
    }

    currentToolIndex = 0;

    do {
      TestTracker->GetTool(currentToolIndex, tool);
```

```
      if (tool.GetStatus() < cisTrackerTool_VISIBLE) {
        cout << iter << " Tool " << currentToolIndex
             << ": Not visible " << endl;
      } else {
        if (tool.GetNumberOfMarkers() == 1) {
          cout << iter << " Tool #" << currentToolIndex
               << ":  Position:" << tool.GetPosition()
               << " (single point is tracked)" << endl;
        } else {
        tool.GetFrame(toolPosition, toolOrientation,toolStatus,
                      toolTimeStamp);
        cout << iter << " Tool #" << currentToolIndex
             << ": Position:" << toolPosition << " , "
             << ": Orientation: "<< toolOrientation <<endl
             << " (multiple points are tracked)" << endl;
        }
      }
      currentToolIndex++;
    } while ((currentToolIndex < nbTools)
             && !(tool.GetNumberOfMarkers() == 1
             && tool.GetStatus() < cisTrackerTool_VISIBLE
             && !tool.IsActive()));
  }

  TestTracker->StopTracking();

  return 0;
}
```

IV. Results

- ?? Functional cisTracker class
- ?? Functional Polaris and Optotrak implementations of the tracker class
- ?? Various modifications in the CIS libraries:
  - Addition/modification to cisSer of cisNetwork to make the code OS independent
  - Creation of cisTime class
- ?? Code review of cisTracker
- ?? Doxygen Documentation
- ?? Final Report – serves as reference manual as well as guide for creating new derived classes for currently unimplemented tracking devices

V. Management Summary

Anand – cisTrackerOptotrak, cisTracker (general class), .dll stuff

Cyrus – cisTrackerPolaris, cisTrackerTool

VI. Future Work

?? Work is currently being done on cisMessage to provide an organized system for dealing with system messages or errors that are generated by any objects created from the cis libraries. Once this is complete, modifications should be made to the cisTracker library in order to make use of the new class.

?? Cyrill von Tiesenhausen is currently working on a CORBA implementation of cisTracker, which will allow tracking devices to be used over a distributed architecture.

?? Other implementation classes can be written for tracking devices that are not already a part of the cisTracker library.

VIII. Acknowledgements

?? Dr. Taylor – Mentor

?? Anton Deguet – Immediate Supervisor, Technical Support

?? Staff/Students of ERC – Moral/Technical Support

?? Adam Morris – Polaris information

?? Cyrill von Tiesenhausen – Consultant on cisTracker architecture

"*This will be the Tracker to end all trackers…*"

-Cyrill von Tiesenhausen