Scene Graph: Visualization of Coordinate Systems in the Image Guided Surgical Toolkit

Janakiram Dandibhotla
Division of Computing Studies
Arizona State University at the Polytechnic Campus
jdandibh@asu.edu

Kevin A. Gary
http://lead4.poly.asu.edu/
Division of Computing Studies
Arizona State University at the Polytechnic Campus
kgary@asu.edu

Abstract

In a surgical environment, there are many tools and objects (including the patient) being used. In a computer-aided surgery, the position of each of these tools in the operating room is very critical and generally will be tracked in their own coordinate systems. To show the surgeon a real time picture of the operating environment on a computer monitor, we need to know the position of each object with respect to one common coordinate system, which in turn can be achieved by knowing each coordinate system and the transforms between them. Image-Guided Surgical Toolkit is a software toolkit designed to enable biomedical researchers to rapidly prototype and create new applications for image-guided surgery. In IGSTK, in which the coordinate systems and the transforms are being successfully used, there is no central data structure or repository, which will hold all the coordinate systems and the transforms between them. Such a data structure could help the IGSTK software developers to have more confidence in the code they have written. This project develops a tool, which will create such a data structure and dynamically show the changes to it, to help such software developers to write better code.

1 Problem Statement

The current IGSTK software developers do not have a way to troubleshoot and verify whether the coordinate systems and transforms, which they have created for the IGSTK application, are being properly constructed and applied. To do this troubleshooting effectively, a data structure, which can hold references to all the coordinate systems being used and the transformations between them is needed. This data structure can be called a "Scene Graph" since this describes the scene of the operating theater at any given point. Without the scene graph, keeping track of the coordinate systems and the relations between them is tough. There is no way to observe the details when a coordinate system is attached to another one or when it is detached. There are also plenty of advantages by creating a Scene Graph. A snapshot of all the coordinate systems and their relationships can be had at any given point of time. The change in relationships between the coordinate systems can be notified to the user. The transforms currently being used can be highlighted.

2 Introduction

2.1 What is IGSTK?

The Image-Guided Surgery Toolkit (IGSTK) is an open-source C++ software library that provides the basic components needed to develop image-guided surgery applications [1]. IGSTK is built on top of several open source software packages, listed below, and FLTK (Fast Light Tool Kit), one among the GUI toolkits, is used to create the dynamic scene graph window.

- The Insight Segmentation and Registration Toolkit (ITK)
- The Visualization Toolkit (VTK)
- GUI toolkits such as FLTK and Qt

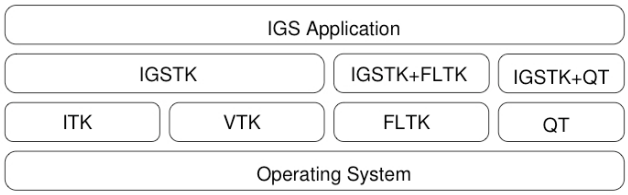

Figure 2.1 IGSTK Layered Architecture.

IGSTK follows state machine architecture. Every component in the operation theater is represented in its own state machine. The communication between the state machines is also isolated. If one of the state machines wants to perform an operation on any other state machine, it has to request the state machine. If the requested state machine can process the request, it will process, or it doesn't. In either case the requested state machine sends an event back to the requesting state machine. This has been dealt in depth in section 2.3.

IGSTK has basically four types of components, which have co-ordinate systems associated with them. They are View, Tracker, TrackerTool and SpatialObject. The functionality of the View class is to aggregate all the graphical representations of spatial objects into one scene. The Tracker class presents a generic interface for tracking the positions of objects in IGSTK. The TrackerTool class provides a generic implementation of a tool of a tracker. This may contain hardware specific details of the tool, along with the fields for position, orientation and error associated with the measurement used. And the SpatialObject class is intended to describe objects in the surgical scenario [2]. Each of these types of classes have a coordinate system object (igstk::CoordinateSystem) associated with them. Any other classes derived from these basic classes also have the access to the coordinate system object.

2.2 Scene Graph

The developers need a way to verify and troubleshoot the coordinate systems and the transforms created by them. Since coordinate systems form the main part of IGSTK

application development, there should be a way to check if they are created in the desired manner. This can be done, in one way, by having a data structure, which holds all the references of the coordinate systems and the transforms and the developers can check this data structure at any given point of time to verify the functionality.

To show all the components, with respect to each other, on a computer screen requires that all the points to be shown should be transformed to one coordinate system. To achieve this, there should be transformations calculated for each of the coordinate system with respect to at least one of other coordinate system. This is exactly being done in IGSTK. Every coordinate system has a reference to its parent coordinate system and it also has the transform required to translate a point from this coordinate system to the parent coordinate system. So, it should not be astonishing to see almost all of the components requesting the coordinate system objects they hold on to, to set the parent coordinate system and the transform between them.

Since each of the above-mentioned four types of objects has a coordinate system associated with it, there will be many coordinate system objects and transforms in the IGSTK application. It would be nice to have a central repository holding on to these coordinate system objects and the transforms between them. It would also be very useful to have a graph to represent this repository at any point of time and the changes this repository has gone through. There is no such central repository in IGSTK at present. This project attempts to fulfill that purpose, to create a data structure to hold all the coordinate system objects and the relationships between them. This data structure will be used in many ways to help the developer of the IGSTK application and at last the user of that IGSTK application.

The data structure holding all the coordinate systems as discussed above will be called "SceneGraph" and the details of creating this data structure will be discussed in the coming sections.

2.3 Event–driven architecture

The communication between different components in IGSTK follows an event-driven architecture. Each and every component of IGSTK is a state machine in itself. So, if one of the components wants to change the state of other component, it has to request that component. When the component being requested processes the request, an event is generated as a result of the process and this event is sent to all the observers of the event. The event generally is associated with a payload and the data required by observers is packed into the payload of the event.

In this project, the scene graph should be notified of any new parent-child relationships being created among coordinate systems to keep track of them. For this purpose a new event has been created which will be invoked when ever a new parent-child relationship has been created. The payload of this event has the parent coordinate system, the child coordinate system and the transform between them. The coordinate system object invokes the event whenever a parent and the corresponding transform are being set. The scene graph is registered as an observer for the event and thus the event is sent to it.

2.4 What is FLTK?

FLTK is the software used to show the scene graph dynamically. This section will try to address what is FLTK and the use of FLTK in this project will be discussed later.

Fast Light Tool Kit (FLTK) is cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL® and its built-in GLUT emulation [5].

FLTK is one of the GUI tools supported by IGSTK. It is a very convenient, simple, cross platform and fast to learn GUI tool kit. FLTK also includes an excellent UI builder called FLUID that can be used to create applications in minutes [5]. FLTK has many different types of components to build UI like windows, buttons, dialogue boxes etc. Some of these have been used to show the scene graph dynamically in this project. It is also very easy to bring up a window in any platform. Just two lines of code:

```
Fl_Window * window = new Fl_Window(200,300,"Test");
window->show();
```

will create a new window of width 200ox, height 300px and name as "Test".

The scene graph can be changed by adding a relationship between coordinate systems or by detaching the child from the parent. These changes to the scene graph will be nice to see if they happen dynamically on a screen, which draws the latest version of scene graph. It would also be interesting to a developer of IGSTK application to look at the transforms currently being used as a highlighted path in the scene graph. The FLTK window being developed to show this dynamic nature incorporates both of these facilities an IGSTK developer would want.

3 Requirements

Requirements of the project are listed below.

3.1 Creating a Scene Graph

The scene graph is a structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene [7]. A scene graph in the context of this project is a data structure holding the relationships between the coordinate systems at any given point of time. The data structure used here is essentially a tree structure with coordinate systems at its nodes and the transforms representing the edges. The scene graph in this context can also have multiple root nodes representing coordinate systems having no parent set.

3.2 Export a snapshot of the scene graph

The scene graph created in requirement 3.1 should be exportable with a button click to some format that can be used to view it as an image. The file format chosen in this project is .dot format. The dot tool from GraphViz can be used to convert this .dot file to a .jpeg image with the graph structure shown. The export should be allowed at any point to get a snapshot of scene graph at that point.

3.3 Visual annotation of the graph

The graphical image created using DOT tool should be able to clearly distinguish between different types of components in the graph and the current path of transformations being used. So, this means the graph should be visually annotated using different colors and shapes to show the different types of objects holding the reference to coordinate systems.

3.4 Dynamic display of scene graph

The scene graph changes in time by adding or deleting some of the relationships among the coordinate systems. It would be a nice idea to show the changes being made to the scene graph dynamically on the GUI (a separate window). The IGSTK application developer can check if those were the changes he/she really intended to do.

3.5 Highlight a path used to compute transform

The dynamic scene graph tree as created in requirement 3.4 should be able to highlight the path used to compute the particular transform in the application. There is only one parent and an associated transform for any coordinate system. To calculate transform from a coordinate system to another coordinate system, which is not the parent, a path has to be calculated so that we can reach the particular coordinate system and calculate transform accordingly in the path. It would be a very useful feature to have this path highlighted in the dynamic scene graph window.

4 Design Approach – Scene Graph

4.1 The Scene Graph Design

A scene graph, in the context of this project, is a data structure holding reference to all the coordinate systems and the relationships, which exists between them. Since each coordinate system can contain only one parent and the corresponding transform, a "TREE structure" will be very suitable in this scenario. A tree structure generally has nodes and edges associated with it. The edges will be directed from the child to the parent.

The nodes of the tree structure discussed above should be a representation of a coordinate system and the edges will the relationship between the different nodes, which is the transform. We can also incorporate the functionality of edges into the nodes, since one child has only one parent. The transform, which ought to be represented by edges, can be integrated with the child node. So, the edges will now just mean the direction of the relationship and will not have any payload or data associated with them.

The node is also used for drawing the scene graph. So, the details regarding the position of the node in the drawing should also be incorporated in the node. To show the current path being used to calculate the transform, we can also add another flag that can be used to highlight the path accordingly.

With all the details discussed above, the node (igstk::SceneGraphNode class) should have these properties:

i) Name of the node.
ii) Type of the coordinate system.
(One of the four types: SpatialObject, Tracker, TrackerTool or View).
iii) Pointer to parent node.
iv) List of pointers to the child nodes.
v) X coordinate and Y coordinate.
(The x and y coordinates of the corner of the node to be drawn on the FLTK scene graph window).
vi) Flag to set if this node is in the current path.
vii) Transform to the parent.

The scene graph consists of a list of root node pointers. There might be the case of multiple root nodes because of the design of the child-parent coordinate system object relations. So accommodate this, the scene graph has a "list of root node pointers".

For example, a snapshot of the scene graph can look as below:

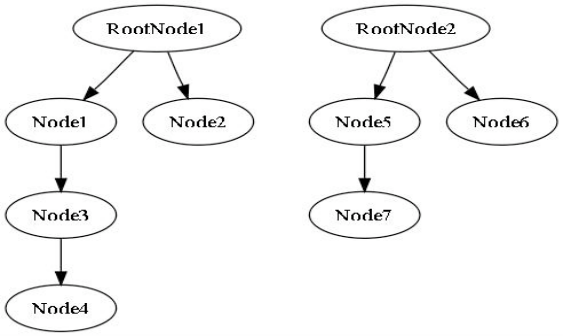

Figure 4.1 Example scene graph.

The example scene graph as shown above has two root nodes and each of them has two child nodes. Some of the child nodes have children and some do not. This diagram implies that the RootNode1 coordinate system is the parent of Node1 coordinate system. The transform between these coordinate systems is stored in the child node, in this case Node1. Each of the nodes has the pointer to the coordinate system objects (igstk::CoordinateSystem) used in IGSTK.

4.2 Coordinate systems in IGSTK

Coordinate systems are very important in IGSTK. IGSTK has several types of components having coordinate systems associated with them as discussed in section 2.1. Coordinate system object is private and to access that, we need to go through Coordinate system delegator class (igstk::CoordinateSystemDelegator). Each one of the four types of components (View, Tracker Tool, Tracker or Spatial Object) has a coordinate system delegator associated with them, which in turn can access the CoordinateSystem class associated with it. The above-mentioned four types of components are generic types and the real functionality for different kinds of tools and instruments is incorporated in the classes derived from them. So, each one these will be having a coordinate system associated with them.

The coordinate system object is also a state machine in IGSTK. This makes it easier to observe the coordinate system, because observing the coordinate system state machine and the events generated by it will be simple. A coordinate system in IGSTK has a parent and a transform with respect to the parent. Whenever a request is made to set the parent and the corresponding transform of a coordinate system, an event can be generated which will be used to update the scene graph. The details needed by the scene graph from the event are:

i) Parent coordinate system.
ii) Child coordinate system.
iii) Transform from child to parent coordinate system or vice-versa.

So an event has been created to exactly accommodate this and with the payload of the parent and child coordinate systems and the transform. The header file is (igstkCoordinateSystemSetTransformResult.h) and the class is (igstk::CoordinateSystemSetTransformResult).

A SceneGraph class (igstk::SceneGraph) has been created to hold the scene graph tree data structures. This class also has all the logic to add and delete any relationships between the coordinate systems. There is also a function in this class that calls the FLTK GUI class, written separately as igstk::SceneGraphUI class, to be shown and hidden accordingly. All the logic to export the scene graph to .dot format and to print the details of the scene graph to console is also incorporated in this class.

4.3 Algorithms – SceneGraph

The algorithms to add and delete nodes in a scene graph are pretty straightforward. The algorithm to add the nodes to the scene graph is as follows:

An event is invoked whenever a parent is being set for a coordinate system. The payload of the event gives all the necessary details required to add it to the scene graph. The payload consists of the parent coordinate system pointer, pointer to child coordinate system and the transform between them. When this event is received,

i) Get the parent coordinate system and check if a node with the same name already exists.
ii) If the parent node already exists, then create a new child node with a reference to child coordinate system and set the parent field accordingly. Also add this child node to the list of children in the parent node.
iii) If the parent node doesn't exist, then create a new parent node with a reference to parent coordinate system and also create a child node with a reference to child coordinate system. Then set the parent in the child coordinate system to be the parent just created and add the child in the list of children in the parent node. Then add the parent node to the list of root nodes in the scene graph object.

In this algorithm, there is a need for optimization. Consider the scenario as shown in the figure 4.1.

Now, the event got by the scene graph has the (parent, child) pair to be (RootNode1, RootNode2). Then according to this algorithm, the parent is checked for and RootNode1 is will be found. Then the RootNode2 will be added as child to RootNode1. But, this will be duplication as shown in the following figure.

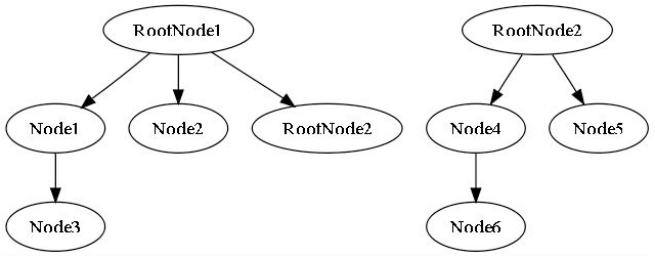

Figure 4.2 Duplication of RootNode2.

So, to eliminate this scenario, we need to optimize the scene graph whenever a node is added or deleted.

The algorithm for optimization will be as follows:

i) Check if any of the root nodes is already a child node of some other node.
ii) If this is the case, then attach this root node as a child node to the node found to be its parent and update the list of children in the parent node.
iii) If nothing is found in step 1, do nothing.

According to this algorithm, if we optimize the above-discussed scenario, we get the scene graph to look as below:

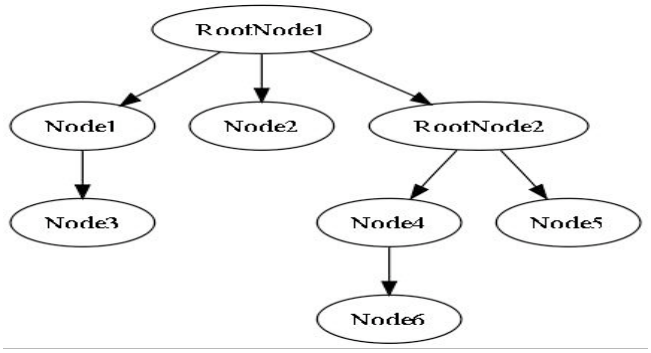

Figure 4.3 Scene Graph after optimization.

If a request to detach the parent from the child coordinate system is made, the scene graph will get the corresponding event and will adjust the graph accordingly, by deleting the link between the corresponding nodes. In this process a new root node might be created too.

5 Functional Design

5.1 Export scene graph

To take a snapshot of the scene graph at a given point of time, the export functionality is provided. The export functionality should facilitate an easy generation of a graphical image of the scene graph. The format chosen in this project is ".dot" format supported by a tool by name "DOT" from GraphViz. In this format, the graphical image, intended to be drawn, is described in a specific text format and the DOT tool can generate an image from the .dot file.

For a small example, consider a graph having only one parent and one child. To draw a graph directed from parent to child, the following lines should be put in a "test.dot" file. Here the PNode denotes the parent node and the CNode denotes the child node.

```
digraph d {
        PNode -> CNode;
}
```

The image is produces as follows.

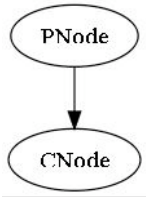

Figure 5.1 Example image created by DOT tool.

There are many other customizations like the color of the node, the size, type and color of the arrow pointing from parent to child etc. The following command uses the DOT tool to convert the file "test.dot" to "test.jpg" image file.

```
$dot  -Tjpg test.dot –o test.jpg
```

So, a snapshot of the scene graph can be very conveniently exported at any time the application is running. For an example of the .dot file created by the export functionality, please see the appendix section.

5.2 Dynamic Scene Graph

The above-explained methodology cannot support dynamic image generation of the changes made to the scene graph. The dynamic nature of scene graph cannot be completely understood by static snapshots of the scene graph at particular intervals. Dynamic graphical image generation in some form can help this dynamic nature of scene graph to be studied in full. Also, all the paths used to calculate transforms from one coordinate system to other are also very difficult to show in the static image. For this purpose, this project uses an FLTK window and other drawing tools provided by FLTK in order to show the scene graph dynamically. FLTK is a very efficient, lightweight drawing tool although it has some drawbacks.

The Fast Light Tool Kit (FLTK) has many GUI widgets used in drawing a scene graph. The basic widgets required to draw a similar image as shown by the DOT tool are "Fl_Box" and "fl_line". The major drawback though is that there is no arrow widget in FLTK although it is not a big constraint. The Fl_Box widget can be used to draw the nodes of the scene graph. The fl_line needs to be used to draw the arrows and links between the nodes of the scene graph. The arrow can be drawn by calculating the end point of a line and using "fl_point" function to draw the arrowhead.

To draw the boxes required to show the nodes, the x and y coordinates of a corner of the box can be calculated and stored in the node object. Since there can be many root nodes associated with a scene graph, the window height and width should be calculated before drawing the scene graph. Then each of the root nodes of the scene graph is grouped into an "Fl_Group" object. Such groups created are then added to the FLTK scene graph window. Each of the nodes also needs its box representation. So, once the x and y coordinates of the corner (top-left) corner of the box are calculated, an Fl_Box is created with those coordinates and of height 20px and width 150px and label being the name of the coordinate system.

5.2.1 Drawing the Scene Graph - Logic

Each group, representing one of the trees, will be drawn from top to bottom. The height and width is calculated and set. To calculate the height of each group, the depth of the tree is calculated and multiplied with (height of each box + height of the cushion space) to get the height in pixels. To calculate the width, the maximum number of nodes at each depth is calculated and this maximum number is multiplied by (width of each box + cushion space). Once the height and width is calculated, the x and y coordinates of each node are calculated and stored in the node. The x and y coordinates are calculated using the depth at which the current node is and the number of boxes in that particular depth in the current group.

Thus the x and y coordinates of each box are calculated. Each box is then created with those coordinates and added to the group, which finally will be added to the window. The figure below shows an example of how a tree looks like.

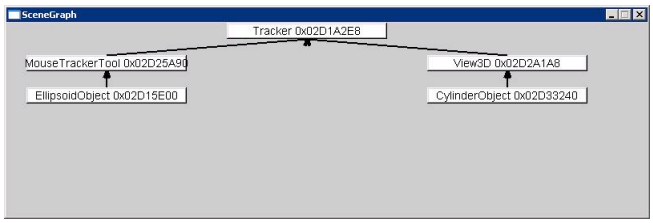

Figure 5.2 Dynamic Scene Graph window.

In the above figure, there is only one tree and five nodes. The x and y coordinates each node was calculated. Then the boxes were created and added to the group. The group was then added to the window. The lines have been created using the fl_line function and the end points were given to this function by getting the stored values of the x and y in the boxes. Once the lines have been drawn, arrows were drawn by filling out a small triangle at one end of the line.

5.2.2 Showing the transform path

To get a point value in one coordinate system from another coordinate system, we need to transform the point by rotation and translation. But, since in IGSTK, there is only parent and transform being maintained per coordinate system, we might need to transform the point to many intermediate coordinate systems before we get the value in required coordinate system. For example, consider a point in CylinderObject coordinate system from Fig 5.2.1.1. The application needs the point in EllipsoidObject coordinate system. So, we can see from the graph that first, the point needs to be transformed into many of the coordinate systems in between (CylinderObject -> View3D -> Tracker -> MouseTrackerTool ->EllipsoidObject) before converting into the required coordinate system. It will be very useful to the IGSTK application developer to see this path in the scene graph. For example, in the figure shown below, the path being used is highlighted in red.

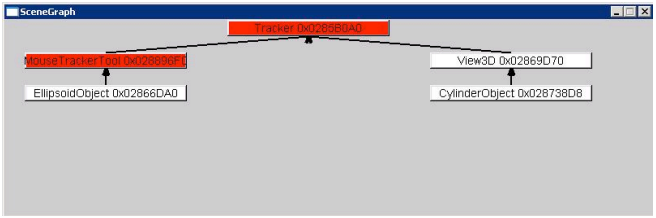

Figure 5.3 Scene Graph showing the transformation path.

To accommodate this behavior, a flag has been added into each of the nodes to represent if it is in the current path being shown. If the variable is flagged, the node is colored in red, thus showing the path.

6 Results

6.1 Export scene graph

The export scene graph functionality as explained above should create a .dot file. The .dot file will be used to create the graphical image using DOT program from GraphViz. One of the snapshots of the image created using the "Hello World" program of IGSTK is as shown below.

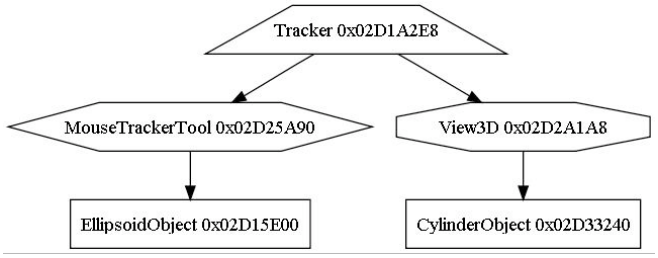

Figure 6.1 Scene Graph image generated by DOT tool.

From this image the type of the node can be very easily identified. Each one of the shapes used in the above figure represents one of the four types of objects (SpatialObject, Tracker, TrackerTool or View). The shapes used are:

Rectangle: Spatial Object
Trapezoid: Tracker
Hexagon: Tracker Tool
Octagon: View

The color of the shapes can also be changed and the different types of nodes can clearly be distinguished. See below figure.

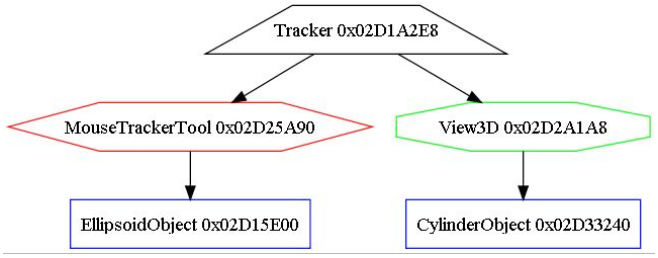

Figure 6.2 Scene Graph image with borders colored.

The nodes can also be filled up with the respective colors and that will be clearer. See the figure below.

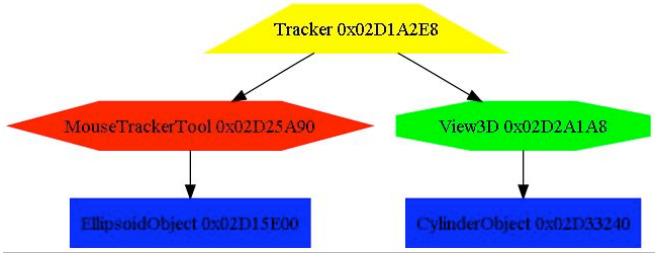

Figure 6.3 Scene Graph with nodes filled with colors.

The colors can be chosen as required and this provides a very clear view of all the different types of components.

6.2 Dynamic scene graph

The dynamic scene graph functionality creates a new FLTK window and shows the current scene graph in the window. Example of a scene graph created by the Hello World program of IGSTK is as seen below:

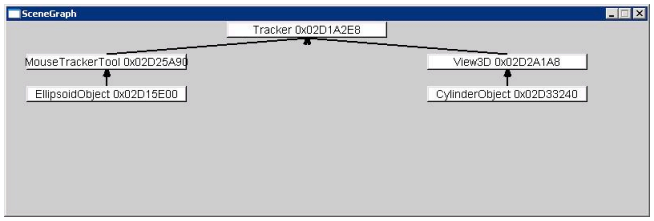

Figure 6.4 Dynamic scene graph FLTK window for Hello World IGSTK application.

The transforms being currently used can be shown on the above window. This helps the developer to verify if the path being used is what he/she is expecting. A screen shot of the dynamic scene graph showing one of the path being used to calculate transforms is as shown below:

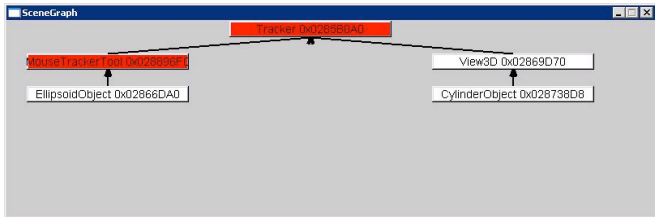

Figure 6.5 Dynamic scene graph showing the transformation path being used in the Hello World example.

7 Advantages for IGSTK application developer

The IGSTK application developer needs to have a way to test and verify the application written. Troubleshooting the coordinate systems and the transforms being used between them is the important part of all the testing. This project provides a tool to the developer to test the coordinate systems and verify if the transforms are being used as intended. The developer has also several advantages as discussed below.

7.1 Re-check the relationships among the coordinate systems

IGSTK applications are developed based on IGSTK framework, which has many components having coordinate systems. There will be many parent-child relationships set among these coordinate systems, but there is no way to check if the coordinate systems have been created as wanted by the developer. The scene graph model helps to achieve this confidence in the application developer. All the relationships between the coordinate systems can be checked and the relationships evaluated by exporting or dynamically viewing the scene graph.

7.2 Check the transformation path

The dynamic window of scene graph, which shows the transformation path, will help the developer to check if that is what he/she is expecting. The current transformation path being used by IGSTK application is shown on the FLTK window for scene graph.

Also, the inclusion of the "export scene graph" and "dynamic scene graph" functionalities in the IGSTK application is very easy. The developer just needs to create a reference to singleton scene graph and call a function to export or pop up the FLTK window.

The IGSTK open source community, namely, Patrick Cheng and Matt Turek, has reviewed this project work and the community approves the work done for this project.

8 Future Work

The work done and the code written for this project will be contributed back to open source project IGSTK and will be part of IGSTK code base.

The project satisfies the initial need for the scene graph and other functionalities, but certainly there are ways for improvements and customizations. For example, the user can be given a property to control which transformation paths to show and what can be skipped. Some proposals for future work include:

i) Export to different formats other than DOT.
ii) Showing the transforms i.e., translations and rotations among the coordinate system in some form on the image.
iii) Extending the dynamic scene graph logic for other UI platforms like Qt.
iv) Allowing the user to control what transformation paths to be highlighted.
v) Allowing users to customize the look and feel like the colors, shapes etc.

9 Acknowledgements

We are grateful to Patrick Cheng and Matt Turek from IGSTK community for investing their time in helping us throughout this project and validating it at the end.

10 References

[1] Kevin Cleary, "IGSTK The Book", pp. 1-200, 2007.
[2] IGSTK Doxygen documentation [Online] http://public.kitware.com/IGSTK/NightlyDoc/ (15Oct2008)
[3] Herbert Schildt, "C++: The Complete Reference, 4th Edition", pp. 50-100, 2002.
[4] What is Scene graph? [Online] http://www.gamedev.net/reference/programming/features/scenegraph/ (20Oct2008)
[5] FLTK Documentation [Online] http://fltk.org/documentation.php/doc-1.1/toc.html (10Nov2008)
[6] Emden Gansner, Eleftherios Koutsofios and Stephen North, "Drawing graphs with dot", pp. 1-28, 2006.
[7] Scene Graph definition – Wikipedia [Online] http://en.wikipedia.org/wiki/Scene_graph (15Nov2008)

11 Appendix

This is the .dot file created when running the hello world example in IGSTK.

```
digraph G {
        "Tracker 0x02D1A2E8" -> "MouseTrackerTool
        0x02D25A90";
        "MouseTrackerTool 0x02D25A90" ->
        "EllipsoidObject 0x02D15E00";
        "Tracker 0x02D1A2E8" -> "View3D
        0x02D2A1A8";
        "View3D 0x02D2A1A8" -> "CylinderObject
        0x02D33240";
        "Tracker 0x02D1A2E8"[shape=polygon, sides=4,
        distortion=-0.7];
        "MouseTrackerTool
        0x02D25A90"[shape=polygon, sides=6];
        "EllipsoidObject 0x02D15E00"[shape=polygon,
        sides=4];
        "View3D 0x02D2A1A8"[shape=polygon, sides=8];
        "CylinderObject 0x02D33240"[shape=polygon,
        sides=4];
}
```

The corresponding image generated by DOT tool is as below:

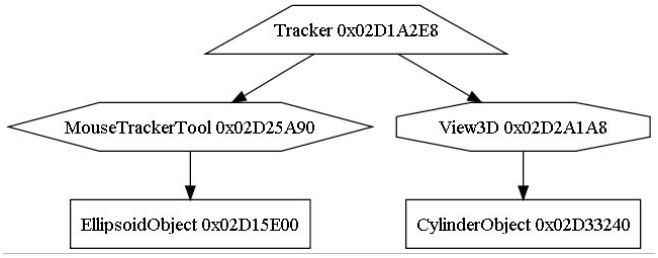

Figure app1 Scene Graph image from the .dot file above.