Introduction to Programming 3D Touch with the GHOST SDK

SensAble TECHNOLOGIES

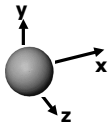
Topics

- Hello GHOST
- Building a Simple Scene
- Grouping and Transforming Objects
- Adding graphics with GhostGL
- Haptics and Graphics Threads

SensAble TECHNOLOGIES

This course aims to give a basic overview of what is involved in writing GHOST SDK programs. We will begin with a simple example - Hello GHOST - that shows the steps required in any GHOST program. We will then build a more complicated scene in the robot example that will introduce the basic building blocks of a scene graph in GHOST SDK. We will then extend this example with OpenGL graphics using the GhostGL library. Along the way, we'll also give an overview of how the haptic and graphic processes cooperate in a haptically enabled application framework.

Hello GHOST



A Simple Program That Renders a Sphere:

- Create a scene object

```
gstScene myScene;
gstSeparator *rootSep = new gstSeparator();
myScene.setRoot(rootSep);
```

- Add a PHANTOM node to the scene

```
gstPHANToM *myPhantom = new gstPHANToM("Default PHANToM");
if (!myPhantom || !myPhantom->getValidConstruction())
   exit(-1);
rootSep->addChild(myPhantom);
```

- Add geometry to the scene

```
gstSphere *mySphere = new gstSphere();
mySphere->setRadius(50);
rootSep->addChild(mySphere);
```

- Start the simulation

```
myScene.startServoLoop();
```

SensAble TECHNOLOGIES

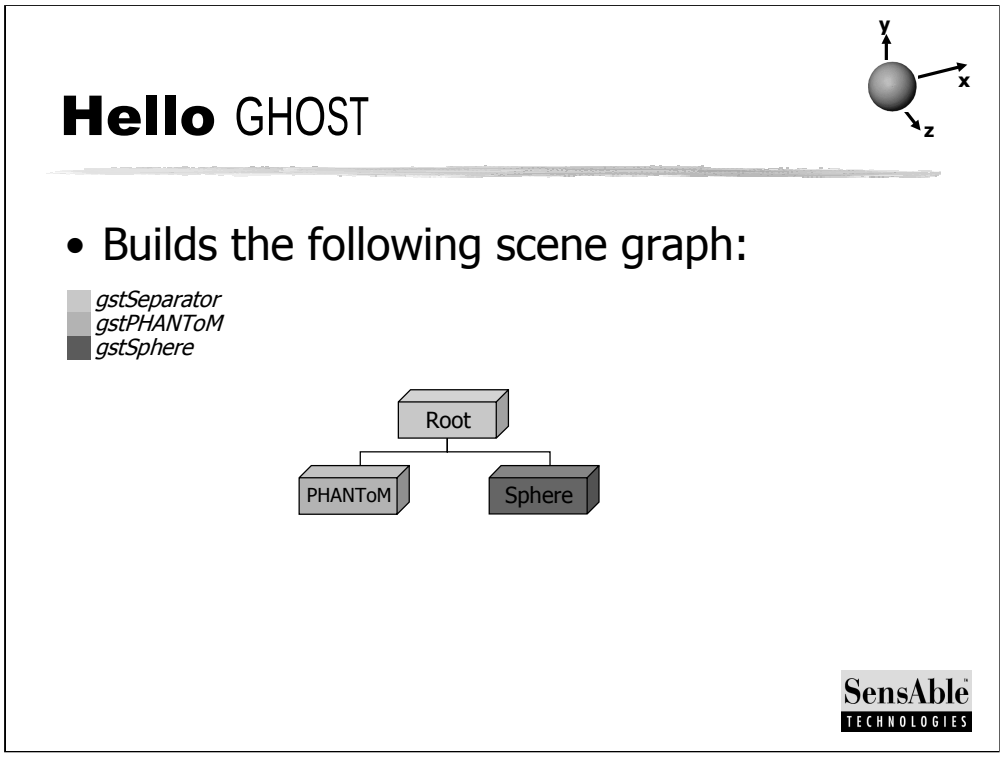
We'll start out with a very simple program that haptically renders a sphere centered at the origin. The steps above show all that is necessary to render a simple object with the GHOST SDK.

First we create a scene object. The scene is a just container for all of the objects in our simulation. We create a root node for our scene graph and it to the scene.

In order to be able to feel the scene we need to add a gstPHANToM node. It is important to call getValidConstruction() - this will return false if the GHOST SDK is unable to communicate with the device.

Next we add geometry nodes to the scene. In this example we create a gstSphere and add it to the scene.

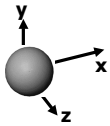
Finally we call startServoLoop() to begin the haptic simulation



The GHOST SDK uses a scene graph to represent all of the geometry and interaction devices in the haptic simulation. Here we see the three nodes that make up the scene graph for Hello GHOST. The sphere and the PHANTOM are both represented by nodes in the scene graph. The root simply serves to group the other two together.

The GHOST SDK calculates any collisions between geometry nodes in the scene and PHANTOM nodes in the scene, calculates the appropriate reaction forces and sends them to the PHANTOM device(s).

Hello GHOST



- Source code
  - link with
    - on Windows NT: ghost.lib
    - on Irix: -lghost*
  - versioned libraries
    - new to Ghost v3.0

*may append _o32, _n32mips3, _n32mips4 or -_64 to GHOST and supporting libraries under Irix depending on binary type chosen in compiler.

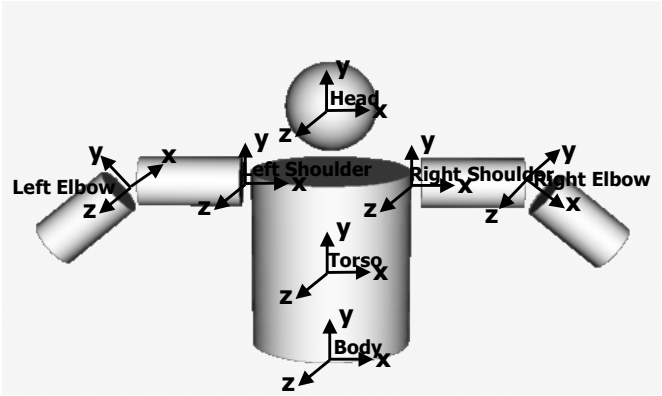
SensAble TECHNOLOGIES

```
//==============================================================================
//     Filename : HelloGhost.cpp
//   Written by : Brandon Itkowitz
//     Contains : Example GHOST Application
// ------------------------------------------------------------------------------
// This program demonstrates how to create a GHOST scene graph, add a primitive
// to it and then feel the scene.
//
// Send your questions, comments or bugs to:
//    support@sensable.com
//    http://www.sensable.com
//
// Copyright (c) 1996-1999 SensAble Technologies, Inc. All rights reserved.
//==============================================================================

#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstSphere.h>
#include <iostream.h>

void main()
{
    // Create a GHOST scene object.
    gstScene myScene;

    // create the root separator and set it as the root of the scene graph
    gstSeparator *rootSep = new gstSeparator();
    myScene.setRoot(rootSep);
```

```
    // prompt the user to place the PHANToM in the reset position
    cout << "Place the PHANToM in its reset position and press <ENTER>." << endl;
    cin.get();

    // create a PHANToM instance and check to make sure it is valid
    gstPHANToM *myPhantom = new gstPHANToM("Default PHANToM");
    if (!myPhantom || !myPhantom->getValidConstruction()) {
        cerr << "Unable to initialize PHANToM device." << endl;
        exit(-1);
    }

    // add the PHANToM object to the scene
    rootSep->addChild(myPhantom);

    // Create a sphere node and add it to the scene
    gstSphere *mySphere = new gstSphere();
    mySphere->setRadius(50);
    rootSep->addChild(mySphere);

    // start the haptic simulation
    myScene.startServoLoop();

    cout << "Press <ENTER> to quit" << endl;
    cin.get();

    // stop the haptic simulation
    myScene.stopServoLoop();
}
```

A More Complicated Scene

- Building a robot



Next we will build a more complicated scene: the upper body of a robot. We will make use of both the gstSphere and gstCylinder primitive shapes, the grouping capability of gstSeparator nodes and the ability to transform (rotate, scale and translate) all of these nodes.

We follow the same basic steps as we did in our simple example:

- Create a scene object
- Add a PHANToM node to the scene
- Add geometry to the scene
- Start the simulation

The only difference is that instead of adding a single sphere to the scene, we will create a more complicated scene by grouping gstSphere and gstCylinder primitives together. This is done in the function createRobotGeometry().

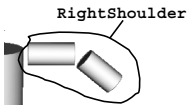
A More Complicated Scene

- Shapes and separators can be transformed

```
rightElbow->setTranslate(30.0,0.0,0.0);
rightElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(-45.0));
```

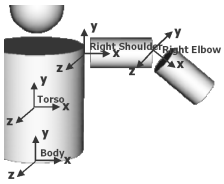
- Use separators to group together nodes

```
rightShoulder->addChild(rightShoulderCylinder);
rightShoulder->addChild(rightElbow);
rightElbow->addChild(rightElbowCylinder);
```



- Transformations accumulate up the graph

Transform of rightElbow is combination of transforms rightShoulder, Torso and Body

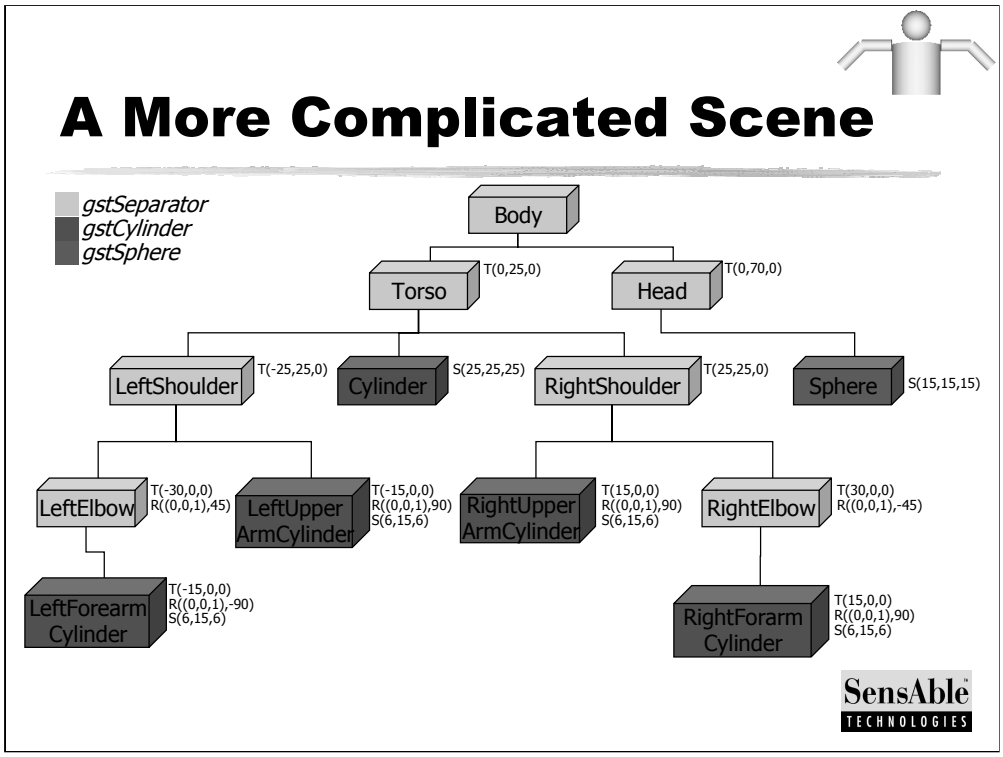


SensAble TECHNOLOGIES

In creating the robot geometry, we take advantage of the fact that gstShape and gstSeparator nodes can be rotated, scaled and translated relative to their parents in the scene graph. For example, the rightShoulder is at (25, 50, 0) and its child the rightElbow is translated (30,0,0) so the right elbow will be at (55, 50, 0) in world coordinates.

Note that total or cumulative transformation of a node is the aggregate of all of the transformations of its antecedents in the scene graph. To calculate the position of the rightElbow in world coordinates you must combine its transformation with the transformations of rightShoulder, Torso and Body in that order.

The advantage of grouping nodes together using separators is that it is easy to move whole parts of the scene graph together. For example, the entire right arm can be moved simply by transforming the rightShoulder separator.



Here is the complete scene graph for the robot example with the transformations marked.

A More Complicated Scene

- Source code
  - link with
    - on Windows NT: ghost.lib
    - on Irix: -lghost*

*may append _o32, _n32mips3, _n32mips4 or -_64 to GHOST and supporting libraries under Irix depending on binary type chosen in compiler.

SensAble TECHNOLOGIES

```
//==============================================================================
//     Filename : Robot.cpp
//   Written by : Brandon Itkowitz
//     Contains : Example Ghost Application w/ Transforms
// ------------------------------------------------------------------------------
// This program haptically renders the upper body of a robot.
// The robot is built using gstShape primitives and gstSeparator nodes.
//
// Send your questions, comments or bugs to:
//    support@sensable.com
//    http://www.sensable.com
//
// Copyright (c) 1996-1999 SensAble Technologies, Inc. All rights reserved.
//==============================================================================

#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>
#include <iostream.h>

inline double degToRad(double degrees)
{
    return (degrees * (M_PI / 180.0));
}
```

```
//==============================================================================
// Function : createRobotGeometry
// ------------------------------
// Generates a collection of Ghost primitives that are constructed to form
// a robot. The body separator gets returned.
//==============================================================================
gstSeparator *createRobotGeometry()
{
    // Create body for robot
    gstSeparator *body = new gstSeparator;

    // Head
    gstSeparator *head = new gstSeparator;
    head->setTranslate(0.0,70.0,0.0);
    body->addChild(head);

    gstSphere *headSphere = new gstSphere;
    headSphere->setRadius(15.0);
    head->addChild(headSphere);

    // Torso
    gstSeparator *torso = new gstSeparator;
    torso->setTranslate(0.0,25.0,0.0);
    body->addChild(torso);

    gstCylinder *torsoCylinder = new gstCylinder;
    torsoCylinder->setRadius(25.0);
    torsoCylinder->setHeight(50.0);
    torso->addChild(torsoCylinder);

    // Right Shoulder
    gstSeparator *rightShoulder = new gstSeparator;
    rightShoulder->setTranslate(25.0,25.0,0.0);
    torso->addChild(rightShoulder);

    gstCylinder *rightShoulderCylinder = new gstCylinder;
    rightShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightShoulderCylinder->setTranslate(15.0,0.0,0.0);
    rightShoulderCylinder->setHeight(30.0);
    rightShoulderCylinder->setRadius(6.0);
    rightShoulder->addChild(rightShoulderCylinder);

    // Right Elbow
    gstSeparator *rightElbow = new gstSeparator;
    rightElbow->setTranslate(30.0,0.0,0.0);
    rightElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(-45.0));
    rightShoulder->addChild(rightElbow);

    gstCylinder *rightElbowCylinder = new gstCylinder;
    rightElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightElbowCylinder->setTranslate(15.0,0.0,0.0);
    rightElbowCylinder->setHeight(30.0);
    rightElbowCylinder->setRadius(6.0);
    rightElbow->addChild(rightElbowCylinder);
```

```
    // Left Shoulder
    gstSeparator *leftShoulder = new gstSeparator;
    torso->addChild(leftShoulder);
    leftShoulder->setTranslate(-25.0,25.0,0.0);

    gstCylinder *leftShoulderCylinder = new gstCylinder;
    leftShoulder->addChild(leftShoulderCylinder);
    leftShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    leftShoulderCylinder->setTranslate(-15.0,0.0,0.0);
    leftShoulderCylinder->setHeight(30.0);
    leftShoulderCylinder->setRadius(6.0);

    // Left Elbow
    gstSeparator *leftElbow = new gstSeparator;
    leftShoulder->addChild(leftElbow);
    leftElbow->setTranslate(-30.0,0.0,0.0);
    leftElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(45.0));

    gstCylinder *leftElbowCylinder = new gstCylinder;
    leftElbow->addChild(leftElbowCylinder);
    leftElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(-90.0));
    leftElbowCylinder->setTranslate(-15.0,0.0,0.0);
    leftElbowCylinder->setHeight(30.0);
    leftElbowCylinder->setRadius(6.0);

    return body;
}
```

```
void main()
{
    // Create a GHOST scene object.
    gstScene myScene;

    // Create root separator of scene graph for robot
    gstSeparator *rootSep = new gstSeparator;
    myScene.setRoot(rootSep);

    // prompt the user to place the PHANToM in the reset position
    cout << "Place the PHANToM in its reset position and press <ENTER>." << endl;
    cin.get();

    // Add PHANTOM node to scene
    gstPHANToM *myPhantom = new gstPHANToM("Default PHANToM");
    if (!myPhantom || !myPhantom->getValidConstruction()) {
        cerr << "Unable to initialize PHANToM device." << endl;
        exit(-1);
    }

    // add the PHANToM object to the scene
    rootSep->addChild(myPhantom);

    // Add geometry to scene
    gstSeparator *robotSep = createRobotGeometry();
    rootSep->addChild(robotSep);

    // start the haptic simulation
    myScene.startServoLoop();

    cout << "Press <ENTER> to quit" << endl;
    cin.get();

    // stop the haptic simulation
    myScene.stopServoLoop();
}
```

Basic Scene Building Blocks

- Shapes
  - sphere, cylinder, cone, cube, torus, triPolyMesh
- Devices
  - PHANToM
- Separators

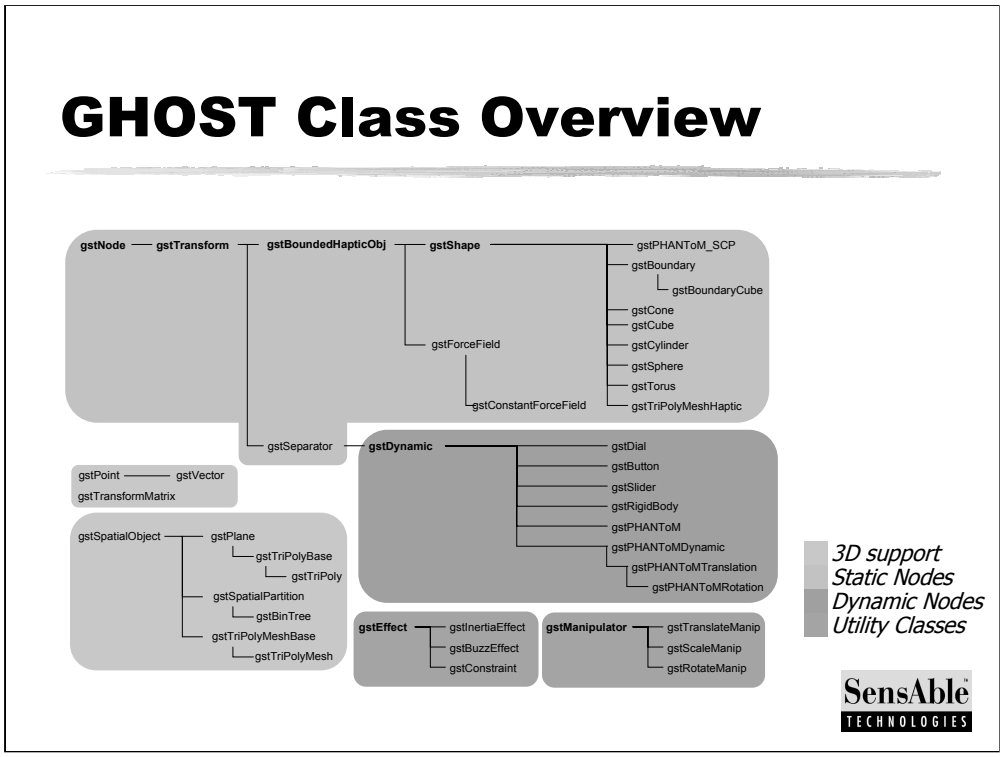
SensAble TECHNOLOGIES

The previous examples showed the basic building blocks of a GHOST scene.

Shape nodes represent solids that can be felt with the PHANToM device. GHOST SDK has shape nodes representing spheres, cubes, cylinders, cones and torii as well as the triPolyMesh class that lets the user create their own solid or surface from a set of triangles that they define.

Device nodes represent user controlled I/O devices. Currently, the GHOST SDK supports only the PHANToM device.

Separator nodes are used to group and separate shape and device nodes.



This diagram shows the complete class hierarchy for GHOST SDK.

Here we can see that gstShape, gstSeparator and gstPHANToM all inherit from both gstNode and gstTransform. gstNode provides some basic capabilities common to any node in the scene graph such as node naming. gstTransform provides the capability to rotate, scale and translate transforms, which are composed in RST order.

GHOST SDK provides a number of other classes in addition to the shape, device and separator classes:

Dynamic object classes which represent objects that move in response to interaction with the PHANToM.

Force fields which represent force interactions in a local area

Effects which are like force fields but operate globally.

Manipulators which allow the PHANToM to interactively affect other nodes in the scene.

Adding Graphics

- Using GhostGL with GLUT
  - Create an instance of ghostGLUTManager

```
ghostGLUTManager *glutManager =
  ghostGLUTManager::CreateInstance(argc, argv, "Robot GhostGL");
```

  - Load scene into ghostGLUTManager

```
glutManager->loadScene(&myScene);
```

  - Start GhostGL main loop

```
glutManager->startMainloop();
```

SensAble TECHNOLOGIES

Now we will add graphics to the robot example using the GhostGL library and the freely available GLUT toolkit. GhostGL is a library that renders the GHOST scene using OpenGL. The library and full source code is provided with GHOST SDK v3.0. GLUT provides a simple platform independent way of creating a window that will accept OpenGL commands. GhostGL can also be used independently of GLUT. The GHOST SDK comes with a Windows MFC framework or an X Windows / Motif framework that both work very well with GhostGL.

Adding GhostGL to the robot example is accomplished with the above three lines of code. First we create an instance of a ghostGLUTManager. This class is the glue between the GLUT window and GhostGL. We pass the command line arguments from our program and the window title into the ghostGLUTManager which will pass them directly onto GLUT. Second, we pass our GHOST scene to GhostGL, so it can traverse it and create OpenGL display list entries for every node in the scene. Finally we start the ghostGLUTManager main loop which will create an OpenGL window and begin rendering into it.

Adding Graphics

- RobotGhostGL source code
  - link with
    - on Windows NT: ghost.lib, ghostGLUTManager.lib, opengl32.lib, glu32.lib, glut32.lib
    - on Irix: -lghostGLUTManager* -lghost* -lglut* -lGLw -lGL -lGLU -lXmu -lXm -lXt -lX11 -lm

*may append _o32, _n32mips3, _n32mips4 or -_64 to GHOST and supporting libraries under Irix depending on binary type chosen in compiler.

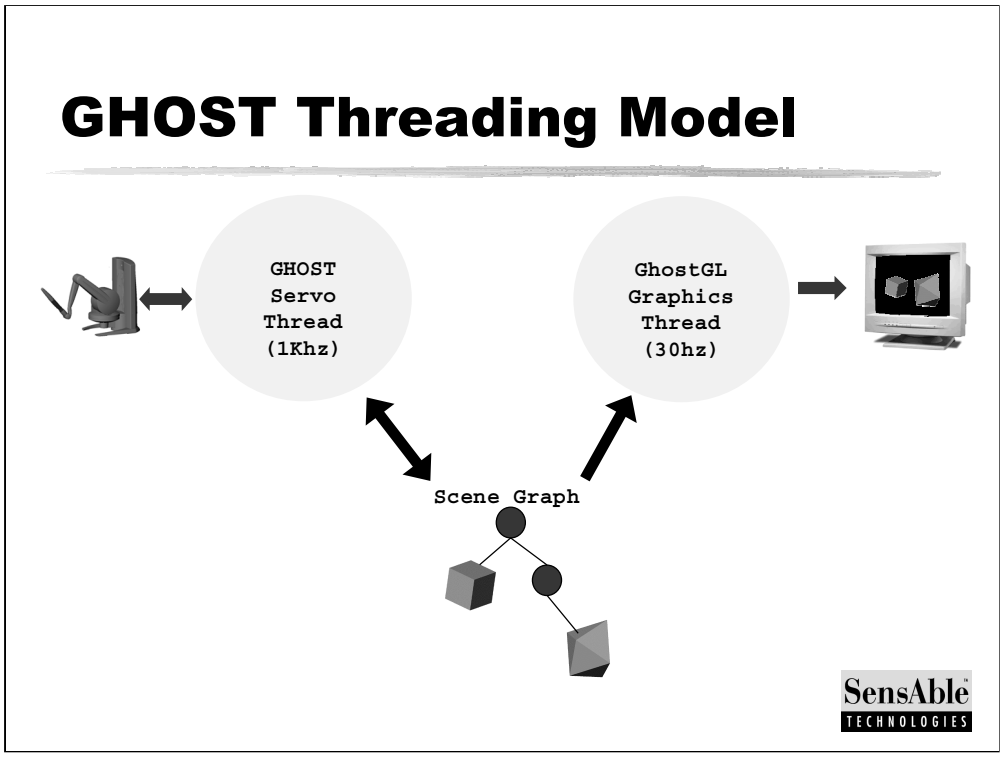
SensAble TECHNOLOGIES

```
//==============================================================================
//     Filename : RobotGhostGL.cpp
//   Written by : Brandon Itkowitz
//     Contains : Example GhostGL Application w/ Transforms
// ------------------------------------------------------------------------------
// This program haptically and graphically renders the upper body of a robot.
// The robot is built using gstShape primitives and gstSeparator nodes.
// The graphics are managed by GLUT and GhostGL.
//
// Send your questions, comments or bugs to:
//    support@sensable.com
//    http://www.sensable.com
//
// Copyright (c) 1996-1999 SensAble Technologies, Inc. All rights reserved.
//==============================================================================

#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>
#include <iostream.h>

#include <ghostGLUTManager.h>

inline double degToRad(double degrees)
{
    return (degrees * (M_PI / 180.0));
}
```

```
//==============================================================================
// Function : createRobotGeometry
// ------------------------------
// Generates a collection of Ghost primitives that are constructed to form
// a robot. The body separator gets returned.
//==============================================================================
gstSeparator *createRobotGeometry()
{
    // create body for robot
    gstSeparator *body = new gstSeparator;

    // Head
    gstSeparator *head = new gstSeparator;
    head->setTranslate(0.0,70.0,0.0);
    body->addChild(head);
    gstSphere *headSphere = new gstSphere;
    headSphere->setRadius(15.0);
    head->addChild(headSphere);

    // Torso
    gstSeparator *torso = new gstSeparator;
    torso->setTranslate(0.0,25.0,0.0);
    body->addChild(torso);

    gstCylinder *torsoCylinder = new gstCylinder;
    torsoCylinder->setRadius(25.0);
    torsoCylinder->setHeight(50.0);
    torso->addChild(torsoCylinder);

    // Right Shoulder
    gstSeparator *rightShoulder = new gstSeparator;
    rightShoulder->setTranslate(25.0,25.0,0.0);
    torso->addChild(rightShoulder);

    gstCylinder *rightShoulderCylinder = new gstCylinder;
    rightShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightShoulderCylinder->setTranslate(15.0,0.0,0.0);
    rightShoulderCylinder->setHeight(30.0);
    rightShoulderCylinder->setRadius(6.0);
    rightShoulder->addChild(rightShoulderCylinder);

    // Right Elbow
    gstSeparator *rightElbow = new gstSeparator;
    rightElbow->setTranslate(30.0,0.0,0.0);
    rightElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(-45.0));
    rightShoulder->addChild(rightElbow);

    gstCylinder *rightElbowCylinder = new gstCylinder;
    rightElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightElbowCylinder->setTranslate(15.0,0.0,0.0);
    rightElbowCylinder->setHeight(30.0);
    rightElbowCylinder->setRadius(6.0);
    rightElbow->addChild(rightElbowCylinder);
```

```
    // Left Shoulder
    gstSeparator *leftShoulder = new gstSeparator;
    torso->addChild(leftShoulder);
    leftShoulder->setTranslate(-25.0,25.0,0.0);

    gstCylinder *leftShoulderCylinder = new gstCylinder;
    leftShoulder->addChild(leftShoulderCylinder);
    leftShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    leftShoulderCylinder->setTranslate(-15.0,0.0,0.0);
    leftShoulderCylinder->setHeight(30.0);
    leftShoulderCylinder->setRadius(6.0);

    // Left Elbow
    gstSeparator *leftElbow = new gstSeparator;
    leftShoulder->addChild(leftElbow);
    leftElbow->setTranslate(-30.0,0.0,0.0);
    leftElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(45.0));

    gstCylinder *leftElbowCylinder = new gstCylinder;
    leftElbow->addChild(leftElbowCylinder);
    leftElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(-90.0));
    leftElbowCylinder->setTranslate(-15.0,0.0,0.0);
    leftElbowCylinder->setHeight(30.0);
    leftElbowCylinder->setRadius(6.0);

    return body;
}
```

```
void main(int argc, char *argv[])
{
    // create a GHOST scene object
    gstScene myScene;

    // create the root separator and set it as the root of the scene graph
    gstSeparator *rootSep = new gstSeparator();
    myScene.setRoot(rootSep);

    // prompt the user to place the PHANToM in the reset position
    cout << "Place the PHANToM in its reset position and press <ENTER>." << endl;
    cin.get();

    // create a PHANToM instance and check to make sure it is valid
    gstPHANToM *myPhantom = new gstPHANToM("Default PHANToM");
    if (!myPhantom || !myPhantom->getValidConstruction()) {
        cerr << "Unable to initialize PHANToM device." << endl;
        exit(-1);
    }

    // add the PHANToM object to the scene
    rootSep->addChild(myPhantom);

    // add robot geometry to scene
    gstSeparator *robotSep = createRobotGeometry();
    rootSep->addChild(robotSep);

    // start the haptic simulation
    myScene.startServoLoop();

    // create an instance of the GLUT OpenGL Manager
    ghostGLUTManager *glutManager = ghostGLUTManager::CreateInstance(argc, argv,
                                                                     "Robot GhostGL");

    // load the scene graph into the ghostGLUTManager instance
    glutManager->loadScene(&myScene);

    // start the display of graphics
    glutManager->startMainloop();

    // stop the haptic simulation
    myScene.stopServoLoop();
}
```



In our example program, the haptics rendering and the graphics rendering are running in two different threads. Both threads access the GHOST scene graph via shared memory (same address space).

The haptics thread, or servo loop, runs as a real-time process at a frequency of 1Khz. In this thread, GHOST SDK updates the position and orientation of all of the PHANToM nodes in the scene graph by querying the associated devices. In addition, it traverses the scene graph and calculates collisions between PHANToM nodes and geometry. The resultant collision forces are fed back to the respective PHANToM devices.

The graphics process runs as a lower priority process at approximately 30hz. In this thread, GhostGL traverses the scene and graphically renders each of the geometry and PHANToM nodes using OpenGL. The application loop can also respond to gstEvents triggered by PHANToM interaction with the scene, receive graphics updates about nodes in the scene that have changed, and respond to user interface events.

Spicing up the Graphics

- GhostGL adds display settings to each node

```
gfxDisplaySettings *displaySettings =
            glutManager->getDisplaySettings((gstTransform *)
                                                headSphere);
```

- Can set pre and post display lists for each node

```
displaySettings->preDisplayList = glGenLists(1);
glNewList(displaySettings->preDisplayList, GL_COMPILE);
        glPushAttrib(GL_LIGHTING_BIT);
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, headColor);
glEndList();
displaySettings->postDisplayList = glGenLists(1);
glNewList(displaySettings->postDisplayList, GL_COMPILE);
        glPopAttrib();
glEndList();
```

SensAble TECHNOLOGIES

In the previous example, the entire robot was drawn in gray. GhostGL does however provide a mechanism for changing the rendering properties of nodes in the scene. When loadScene is called, GhostGL creates a gfxDisplaySettings object for each node in the scene. This object contains OpenGL display lists that are executed before, during and after the geometry is drawn. By customizing these display lists you can set material properties such as colors and textures, as well as change the graphics geometry to be different from the haptics geometry.

In this example, we set the pre and post display lists for the robot's gstSphere head node so that it is drawn in red. The pre display list is called before the sphere is drawn, which saves off the current color and sets the new color to red. The post display list is called after the sphere is drawn to restore the color saved off in the pre display list. If this color was not restored, the entire robot would be drawn in red.

Note: in order to easily acquire a pointer to the headSphere node, we added the following line to createRobotGeometry:

```
headSphere->setName("headsphere");
```

This allows us to retrieve that node by name later on in setHeadColor().

Spicing up the Graphics

- RobotColor source code
  - link with
    - on Windows NT: ghost.lib, ghostGLUTManager.lib, opengl32.lib, glu32.lib, glut32.lib
    - on Irix: -lghostGLUTManager* -lghost* -lglut* -lGLw -lGL -lGLU -lXmu -lXm -lXt -lX11 -lm

*may append _o32, _n32mips3, _n32mips4 or -_64 to GHOST and supporting libraries under Irix depending on binary type chosen in compiler.

SensAble TECHNOLOGIES

```
//==============================================================================
//     Filename : RobotColor.cpp
//   Written by : Brandon Itkowitz
//     Contains : Example GhostGL Application w/ Transforms & Color
// ------------------------------------------------------------------------------
// This program haptically and graphically renders the upper body of a robot.
// The robot is built using gstShape primitives and gstSeparator nodes.
// The graphics are managed by GLUT and GhostGL.
// Additionally, this example shows how to modify OpenGL rendering properties
//
// Send your questions, comments or bugs to:
//    support@sensable.com
//    http://www.sensable.com
//
// Copyright (c) 1996-1999 SensAble Technologies, Inc. All rights reserved.
//==============================================================================

#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>
#include <iostream.h>

#include <ghostGLUTManager.h>

inline double degToRad(double degrees)
{
    return (degrees * (M_PI / 180.0));
}
```

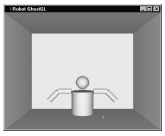
```
//==============================================================================
// Function : createRobotGeometry
// ------------------------------
// Generates a collection of Ghost primitives that are constructed to form
// a robot. The body separator gets returned.
//==============================================================================
gstSeparator *createRobotGeometry()
{
    // create body for robot
    gstSeparator *body = new gstSeparator;

    // Head
    gstSeparator *head = new gstSeparator;
    head->setTranslate(0.0,70.0,0.0);
    body->addChild(head);

    gstSphere *headSphere = new gstSphere;
    headSphere->setRadius(15.0);
    headSphere->setName(gstNodeName("headsphere"));
    head->addChild(headSphere);

    // Torso
    gstSeparator *torso = new gstSeparator;
    torso->setTranslate(0.0,25.0,0.0);
    body->addChild(torso);

    gstCylinder *torsoCylinder = new gstCylinder;
    torsoCylinder->setRadius(25.0);
    torsoCylinder->setHeight(50.0);
    torso->addChild(torsoCylinder);

    // Right Shoulder
    gstSeparator *rightShoulder = new gstSeparator;
    rightShoulder->setTranslate(25.0,25.0,0.0);
    torso->addChild(rightShoulder);

    gstCylinder *rightShoulderCylinder = new gstCylinder;
    rightShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightShoulderCylinder->setTranslate(15.0,0.0,0.0);
    rightShoulderCylinder->setHeight(30.0);
    rightShoulderCylinder->setRadius(6.0);
    rightShoulder->addChild(rightShoulderCylinder);

    // Right Elbow
    gstSeparator *rightElbow = new gstSeparator;
    rightElbow->setTranslate(30.0,0.0,0.0);
    rightElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(-45.0));
    rightShoulder->addChild(rightElbow);

    gstCylinder *rightElbowCylinder = new gstCylinder;
    rightElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightElbowCylinder->setTranslate(15.0,0.0,0.0);
    rightElbowCylinder->setHeight(30.0);
    rightElbowCylinder->setRadius(6.0);
    rightElbow->addChild(rightElbowCylinder);
```

```
    // Left Shoulder
    gstSeparator *leftShoulder = new gstSeparator;
    torso->addChild(leftShoulder);
    leftShoulder->setTranslate(-25.0,25.0,0.0);

    gstCylinder *leftShoulderCylinder = new gstCylinder;
    leftShoulder->addChild(leftShoulderCylinder);
    leftShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    leftShoulderCylinder->setTranslate(-15.0,0.0,0.0);
    leftShoulderCylinder->setHeight(30.0);
    leftShoulderCylinder->setRadius(6.0);

    // Left Elbow
    gstSeparator *leftElbow = new gstSeparator;
    leftShoulder->addChild(leftElbow);
    leftElbow->setTranslate(-30.0,0.0,0.0);
    leftElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(45.0));

    gstCylinder *leftElbowCylinder = new gstCylinder;
    leftElbow->addChild(leftElbowCylinder);
    leftElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(-90.0));
    leftElbowCylinder->setTranslate(-15.0,0.0,0.0);
    leftElbowCylinder->setHeight(30.0);
    leftElbowCylinder->setRadius(6.0);

    return body;
}
```

```
//==============================================================================
// Function : setHeadColor
// ------------------------
// Uses GhostGL pre and post display lists to set the color of the head
//==============================================================================
void setHeadColor(gstScene *scene, ghostGLUTManager *glutManager)
{
    // Find headSphere node in scene by name
    gstNode *headSphere = scene->getRoot()->getByName("headsphere");

    // Get ghostGLManager display settings for the node
    gfxDisplaySettings *displaySettings =
            glutManager->getDisplaySettings((gstTransform *) headSphere);

    GLfloat headColor[] = {1.0f, 0.0f, 0.0f, 1.0f};

    // Pre display list is called before head is drawn
    // it saves current color and sets color to be red
    displaySettings->preDisplayList = glGenLists(1);
    glNewList(displaySettings->preDisplayList, GL_COMPILE);
        glPushAttrib(GL_LIGHTING_BIT);
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, headColor);
    glEndList();

    // Post display is called after head is drawn
    // it restores the saved color.
    displaySettings->postDisplayList = glGenLists(1);
    glNewList(displaySettings->postDisplayList, GL_COMPILE);
        glPopAttrib();
    glEndList();
}
```

```
void main(int argc, char *argv[])
{
    // create a GHOST scene object
    gstScene myScene;

    // create the root separator and set it as the root of the scene graph
    gstSeparator *rootSep = new gstSeparator();
    myScene.setRoot(rootSep);

    // prompt the user to place the PHANToM in the reset position
    cout << "Place the PHANToM in its reset position and press <ENTER>." << endl;
    cin.get();

    // create a PHANToM instance and check to make sure it is valid
    gstPHANToM *myPhantom = new gstPHANToM("Default PHANToM");
    if (!myPhantom || !myPhantom->getValidConstruction()) {
        cerr << "Unable to initialize PHANToM device." << endl;
        exit(-1);
    }

    // add the PHANToM object to the scene
    rootSep->addChild(myPhantom);

    // add robot geometry to scene
    gstSeparator *robotSep = createRobotGeometry();
    rootSep->addChild(robotSep);

    // start the haptic simulation
    myScene.startServoLoop();

    // create an instance of the GLUT OpenGL Manager
    ghostGLUTManager *glutManager = ghostGLUTManager::CreateInstance(argc, argv,
                                                                     "RobotColor");

    // load the scene graph into the ghostGLUTManager instance
    glutManager->loadScene(&myScene);

    // Change the color of the head
    setHeadColor(&myScene, glutManager);

    // start the display of graphics
    glutManager->startMainloop();

    // stop the haptic simulation
    myScene.stopServoLoop();
}
```

Adding a workspace



- GHOST now provides standard workspace API for determining approximate bounds for a particular PHANToM device
- GhostGL knows how to sync camera with workspace to provide optimal viewing
  - camera to workspace syncing (default)
  - workspace to camera syncing
  - workspace to window syncing (default)

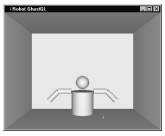
SensAble TECHNOLOGIES

GHOST now provides a standard facility for querying the usable workspace dimensions from a gstPHANToM instance. This workspace represents a rectangular volume in physical space where every point is touchable by the PHANToM. By treating this workspace as a confinement on the PHANToM, you can theoretically place this workspace anywhere in your geometry space and feel whatever lies within it.

GhostGL knows about the GHOST workspace boundary and can use it for determining viewing parameters. The ghostGLSyncCamera offers a few modes of operation:
1) syncing the camera to the workspace guarantees that you can see whatever is presently within the workspace
2) syncing the workspace to the camera guarantees that you can feel whatever the camera is looking at (z depth is still an issue however)
3) syncing the workspace to the window will maximally adjust the workspace dimensions to fit the aspect ratio of the window.

Modes 1 & 2 are mutually exclusive, but mode 3 can be toggled on or off. It is enabled by default.

Adding a workspace



- RobotWorkspace source code
  - link with
    - on Windows NT: ghost.lib, ghostGLUTManager.lib, opengl32.lib, glu32.lib, glut32.lib
    - on Irix: -lghostGLUTManager* -lghost* -lglut* -lGLw -lGL -lGLU -lXmu -lXm -lXt -lX11 -lm

*may append _o32, _n32mips3, _n32mips4 or -_64 to GHOST and supporting libraries under Irix depending on binary type chosen in compiler.

```
//==============================================================================
//     Filename : RobotWorkspace.cpp
//   Written by : Brandon Itkowitz
//     Contains : Example GhostGL Application w/ Transforms & Workspace
// ------------------------------------------------------------------------------
// This program haptically and graphically renders the upper body of a robot.
// The robot is built using gstShape primitives and gstSeparator nodes.
// The graphics are managed by GLUT and GhostGL.
// This example additionally demonstrates the use of a workspace and a
// ghostGLSyncCamera to provide a bounded work environment
//
// Send your questions, comments or bugs to:
//    support@sensable.com
//    http://www.sensable.com
//
// Copyright (c) 1996-1999 SensAble Technologies, Inc. All rights reserved.
//==============================================================================

#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>
#include <gstBoundaryCube.h>
#include <iostream.h>

#include <ghostGLUTManager.h>
#include <ghostGLSyncCamera.h>
```

```
inline double degToRad(double degrees)
{
    return (degrees * (M_PI / 180.0));
}

//==============================================================================
// Function : createRobotGeometry
// ------------------------------
// Generates a collection of Ghost primitives that are constructed to form
// a robot. The body separator gets returned.
//==============================================================================
gstSeparator *createRobotGeometry()
{
    // create body for robot
    gstSeparator *body = new gstSeparator;

    // Head
    gstSeparator *head = new gstSeparator;
    head->setTranslate(0.0,70.0,0.0);
    body->addChild(head);

    gstSphere *headSphere = new gstSphere;
    headSphere->setRadius(15.0);
    head->addChild(headSphere);

    // Torso
    gstSeparator *torso = new gstSeparator;
    torso->setTranslate(0.0,25.0,0.0);
    body->addChild(torso);

    gstCylinder *torsoCylinder = new gstCylinder;
    torsoCylinder->setRadius(25.0);
    torsoCylinder->setHeight(50.0);
    torso->addChild(torsoCylinder);

    // Right Shoulder
    gstSeparator *rightShoulder = new gstSeparator;
    rightShoulder->setTranslate(25.0,25.0,0.0);
    torso->addChild(rightShoulder);

    gstCylinder *rightShoulderCylinder = new gstCylinder;
    rightShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightShoulderCylinder->setTranslate(15.0,0.0,0.0);
    rightShoulderCylinder->setHeight(30.0);
    rightShoulderCylinder->setRadius(6.0);
    rightShoulder->addChild(rightShoulderCylinder);

    // Right Elbow
    gstSeparator *rightElbow = new gstSeparator;
    rightElbow->setTranslate(30.0,0.0,0.0);
    rightElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(-45.0));
    rightShoulder->addChild(rightElbow);
```

```
    gstCylinder *rightElbowCylinder = new gstCylinder;
    rightElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    rightElbowCylinder->setTranslate(15.0,0.0,0.0);
    rightElbowCylinder->setHeight(30.0);
    rightElbowCylinder->setRadius(6.0);
    rightElbow->addChild(rightElbowCylinder);

    // Left Shoulder
    gstSeparator *leftShoulder = new gstSeparator;
    torso->addChild(leftShoulder);
    leftShoulder->setTranslate(-25.0,25.0,0.0);

    gstCylinder *leftShoulderCylinder = new gstCylinder;
    leftShoulder->addChild(leftShoulderCylinder);
    leftShoulderCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(90.0));
    leftShoulderCylinder->setTranslate(-15.0,0.0,0.0);
    leftShoulderCylinder->setHeight(30.0);
    leftShoulderCylinder->setRadius(6.0);

    // Left Elbow
    gstSeparator *leftElbow = new gstSeparator;
    leftShoulder->addChild(leftElbow);
    leftElbow->setTranslate(-30.0,0.0,0.0);
    leftElbow->setRotate(gstVector(0.0,0.0,1.0),degToRad(45.0));

    gstCylinder *leftElbowCylinder = new gstCylinder;
    leftElbow->addChild(leftElbowCylinder);
    leftElbowCylinder->setRotate(gstVector(0.0,0.0,1.0),degToRad(-90.0));
    leftElbowCylinder->setTranslate(-15.0,0.0,0.0);
    leftElbowCylinder->setHeight(30.0);
    leftElbowCylinder->setRadius(6.0);

    return body;
}
```

```
void main(int argc, char *argv[])
{
    // create a GHOST scene object
    gstScene myScene;

    // create the root separator and set it as the root of the scene graph
    gstSeparator *rootSep = new gstSeparator();
    myScene.setRoot(rootSep);

    // prompt the user to place the PHANToM in the reset position
    cout << "Place the PHANToM in its reset position and press <ENTER>." << endl;
    cin.get();

    // create a PHANToM instance and check to make sure it is valid
    gstPHANToM *myPhantom = new gstPHANToM("Default PHANToM");
    if (!myPhantom || !myPhantom->getValidConstruction()) {
        cerr << "Unable to initialize PHANToM device." << endl;
        exit(-1);
    }

    // create a parent for the PHANToM and add both to the scene
    gstSeparator *phantomParent = new gstSeparator();
    rootSep->addChild(phantomParent);
    phantomParent->addChild(myPhantom);

    // attach a workspace boundary (added as a sibling to the PHANToM)
    // this routine requires the PHANToM to have a distinct parent node
    gstBoundaryCube *myBoundary = myPhantom->attachMaximalBoundary();

    // add robot geometry to scene
    gstSeparator *robotSep = createRobotGeometry();
    rootSep->addChild(robotSep);

    // start the haptic simulation
    myScene.startServoLoop();

    // create an instance of the GLUT OpenGL Manager
    ghostGLUTManager *glutManager = ghostGLUTManager::CreateInstance(argc, argv,
                                                                     "Robot Workspace");

    // use a sync camera which by default syncs the camera to the workspace
    // as well as adjusts the workspace to fit in the window
    ghostGLSyncCamera *myCamera = new ghostGLSyncCamera();
    glutManager->setCamera(myCamera);
    myCamera->setWorkspaceBoundary(myBoundary);

    // load the scene graph into the ghostGLUTManager instance
    glutManager->loadScene(&myScene);

    // start the display of graphics
    glutManager->startMainloop();

    // stop the haptic simulation
    myScene.stopServoLoop();
}
```

Summary

- Simple GHOST program
- Basic GHOST SDK classes
- Scene graph
- Graphics and haptics
- Using GhostGL for graphics
- More info
  - GHOST SDK Programmers Guide
  - www.sensable.com/support/ghost_class.htm

SensAble TECHNOLOGIES

Today, we've seen how to build a simple GHOST program to render basic geometry. We've seen how we can extend that scene to create more complicated scenes and use OpenGL graphics. We also saw the basic class structure of GHOST and how GHOST programs are broken down into haptics and graphics threads.

For more information, consult the GHOST SDK Programmers Guide and our complete GHOST SDK Tutorial on our website.