GHOST® SDK

Programmer's Guide

Version 4.0

SensAble Technologies, Inc.®

Rev. 1.5 4/23/02

STI License Number: 83111120

GHOST® SDK Development License Agreement

SensAble Technologies, Inc.
15 Constitution Way
Woburn, MA 01801
United States

READ THE TERMS AND CONDITIONS OF THIS SOFTWARE LICENSE ("LICENSE") CAREFULLY BEFORE USING THIS SOFTWARE. BY INSTALLING AND/OR USING THE ENCLOSED GHOST Software Developer's Toolkit (the "Software"), YOU (the "Authorized User") REPRESENT THAT YOU ARE AUTHORIZED TO BIND YOUR EMPLOYER ("Licensed User") IN CONTRACT AND THAT THE LICENSED USER ACCEPTS THE FOLLOWING TERMS AND CONDITIONS.

IF THE LICENSED USER DOES NOT AGREE TO THESE TERMS AND CONDITIONS, THE UNUSED SOFTWARE MUST BE RETURNED WITHIN FIFTEEN (15) DAYS OF PURCHASE FOR A REFUND.

THE LICENSED USER MAY NOT USE, COPY, MODIFY, DISTRIBUTE OR TRANSFER THE SOFTWARE, EXCEPT AS EXPRESSLY PROVIDED IN THE LICENSE.

1. **License to Use.** The software is licensed to the Licensed User, not sold, under the terms of this License. SensAble Technologies, Incorporated ("STI") grants the Licensed User a non-exclusive and non-transferable right to: (1) install one copy of the Software, in machine readable form on the one computer for which the corresponding fee has been paid; and (2) use the Software, together with the accompanying Documentation, to develop software applications for the Licensed User's internal use with hardware which implements the PHANTOM™ haptic interface (including the modification of any examples provided in the Software in the course of developing such software applications).

2. **Limitations on Use**. Except as expressly permitted, THE LICENSED USER MAY NOT:
   - Develop, reproduce, or distribute to a third party any software application which uses, incorporates and/or requires the Software, except pursuant to a separate Object Code Distribution License or License for Academic Use from STI, which will provide the appropriate licenses for such development and for the distribution of the resulting software application subject to, among other requirements, applicable royalties and fees (if any), product support, and branding requirements.
   - Make copies of the Software or Documentation, except for one (1) copy for backup or archival purposes.
   - Lease, rent, sell, sub-license, transfer, distribute, modify, translate, reverse engineer, decompile or disassemble the Software or Documentation.
   - Remove any proprietary notices, labels or trademarks of STI or its licensors on the Software.
   - Use the Software in the design or development of applications intended for use in hazardous environments such as the operation of nuclear facilities, aircraft navigation or control, or direct life support machines.

3. **Ownership.** The Software is owned by STI or its licensors and is protected by US and international patent and copyright laws and international treaties. Rights not expressly granted are reserved to STI and its licensors. Title, ownership rights, and intellectual property rights in and to the Software and all copies thereof shall remain in STI or its licensors. No rights in and to the Software, by implication or estoppel, are granted and/or transferred to the Licensed User except as explicitly provided for in this License. Unauthorized use, copying or distribution of the Software, or failure to comply with this License, will result in automatic termination of the License and will make other legal remedies available to STI.

4. **Limited Warranty.** STI warrants that the media on which the Software is furnished will be free of defects in materials and workmanship under normal use for a period of ninety (90) days from the date of purchase, as evidenced by a copy of the Licensed User's receipt. Otherwise, the Software is provided "AS IS" without warranty of any kind. This warranty extends only to the Licensed User as the original licensee.

5. **Limited Technical Support.** For a period of 90 days after the shipment date of the Software, STI agrees to provide telephone support services expressly limited to installation of the Software and correction of reproducible errors in the Software.

6. **Customer Remedies.** STI's entire liability and the Licensed User's exclusive remedy under this warranty will be the correction of defects in media or the replacement of media, or, if correction or replacement is not reasonably achievable by STI, the refund to the Licensed User of the license fee paid, upon return of the Software and documentation.

7. **Disclaimer of Other Warranties.** EXCEPT AS SPECIFIED IN THIS LICENSE, ALL EXPRESS OR IMPLIED REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED TO THE EXTENT ALLOWED BY APPLICABLE LAW. STI DOES NOT WARRANT THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR FREE. THIS LIMITED WARRANTY GIVES THE LICENSED USER SPECIFIC RIGHTS. THE LICENSED USER MAY HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

8. **Limitation of Damages.** IN NO EVENT WILL STI BE LIABLE FOR ANY LOST REVENUE, PROFIT, OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL, OR PUNITIVE DAMAGES HOWEVER CAUSED AND REGARDLESS OF THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF STI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. The foregoing limitation of liability and exclusion of certain damages shall apply regardless of the success or effectiveness of other remedies. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages so the above limitation or exclusion may not apply to the Licensed User.

9. **Confidentiality.** The Software is the confidential and proprietary information of STI and its licensors. The Licensed User agrees to take adequate steps to protect the Software from unauthorized disclosure or use.

10. **Termination.** This License is effective until terminated. The Licensed User may terminate this Software License at any time by destroying all copies of the Software and documentation. Upon termination, the Licensed User must destroy all copies of Software and documentation. The Licensed User's license to use the Software will terminate immediately without notice from STI if the Licensed User fails to comply with any provision of this license.

11. **Export.** This Software is licensed for use in the US and its territories, or in any other country to which it is legally exported. The Licensed User agrees not to export or re-export, directly or indirectly, the Software, any technical data in the Software, or any direct or derivative product of the Software, except upon compliance with applicable US laws governing export, destination, ultimate end user, and other matters.

12. **US Government Restricted Rights.** The Software is licensed to the Licensed User with restricted rights. The licensed Software constitutes "commercial terms"-as that term is defined in 48 C.F.R.>2.101 (October 1995) consisting of "commercial computer software" and "commercial computer software documentation" as such terms are used in 48 C.F.R. 12.212 (September 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1, 227.7202-3 and 227.7202-4 (June 1995), if the Licensee hereunder is the U.S. Government or any agency or department thereof, the

Licensed Software is licensed hereunder (i) only as a commercial item, and (ii) with only those rights as are granted to all other Customers pursuant to the terms and conditions of this Agreement.

13. **No Franchise Rights.** The relationship of the parties established by this License shall be that of independent contractors, and nothing contained in this License shall be construed as creating any relationship of agency, partnership, franchise, joint venturers, employment or similar relationship between the parties.

14. **No Assignment.** The Licensed User shall not directly or indirectly sell, transfer, assign, or delegate in whole or in part this License, or any rights, duties, obligations or liabilities under this License, to any third party, including to any affiliated entity, without the prior written consent of STI, which consent may be withheld in the absolute discretion of STI.

15. **Compliance with Laws.** The Licensed User will comply, at the Licensed User's own expense, with all statutes, regulations, rules and ordinances of any governmental body, department or agency which apply to or result from the Customer's obligations under this License. The Licensed User agrees not to export the Software(s) directly or indirectly, separately or as part of a system, without first obtaining proper authority to do so from the appropriate governmental agencies or entities, as may be required by law.

16. **Governing Law.** This License is made under and will be governed by the substantive laws of the United States and the Commonwealth of Massachusetts, excluding any of its choice of law provisions. The original of this License has been written in English. The parties hereto waive any statute, law, or regulation that might provide an alternative law or forum or to have this License written in any language other than English. The parties hereto exclude the United Nations Convention on Contracts for the International Sale of Goods from this License and any transaction between them that may be implemented in connection with this License.

17. **Venue and Jurisdiction**. The state and federal courts in the Commonwealth of Massachusetts shall have exclusive venue and jurisdiction for any disputes, and the parties hereby submit to personal jurisdiction in such courts.

18. **Entire Agreement.** This License is the entire agreement between the Licensed User and STI relating in any way to the Software. This License supersedes any proposal or prior agreement, oral or written, and any other communication relating to the subject matter of this License. No variation of the terms of this License or any different terms will be enforceable against STI unless STI gives its express consent, including an express waiver of the terms of this License, signed in writing by an officer of STI.

19. **Severability.** If any of the above provisions are found to be in violation of applicable law, void, or unenforceable in any jurisdiction, then such provisions are waived to the extent necessary for this License to be otherwise enforceable in such jurisdiction.

GHOST® SDK

Programmer's Guide

Version 4.0

SensAble Technologies, Inc.®

Rev. 1.5 4/23/02

Acknowledgments

We'd like to thank our customers for their invaluable feedback and their confidence in SensAble Technologies. We'd also like to thank the many people at SensAble Technologies who make our products a reality – software development, testing, documentation, manufacturing, marketing, sales and logistics. Finally, a special thanks to the GHOST SDK development team: Brandon Itkowitz, Divya Mehra, Philip Winston, Chris Tarr and Tony Tomc for their contributions to the GHOST SDK and PHANTOM Device Drivers.

We hope you find the product reliable and useful in your application. If you have any feedback, please don't hesitate to contact us at 1-888-SENSABL or support@sensable.com.

Manual Updates

Please check the SensAble (www.sensable.com) web site for updated information on the GHOST SDK.

Contents

Preface

The SensAble Technologies Inc. (STI) team would like to thank you for purchasing the GHOST (General Haptics Open Software Toolkit) Software Developer's Toolkit (SDK). We believe that the *GHOST SDK* will significantly improve the way application developers and researchers incorporate haptic interaction technology into their systems, and we hope you'll find the *GHOST SDK* beneficial to your work as you use the product.

About This Guide

The *GHOST SDK* Programmer's Guide introduces application developers to the *GHOST SDK*, an object-oriented 3D haptic toolkit used with SensAble Technologies PHANTOM® haptic interfaces. The *GHOST SDK* is a C++ library of objects and methods used for developing interactive, three-dimensional, touch-enabled environments. This guide describes the general features of the *GHOST SDK*, how to write applications using the toolkit, and how to extend the *GHOST SDK*.

How This Guide Is Organized

The *GHOST SDK Programmer's Guide* is divided as follows:

- ***Chapter 1: Installing the GHOST SDK*** describes how to install the software.
- ***Chapter 2: The GHOST API*** describes the overall *GHOST SDK* programming environment and the main object classes available to the application developer.
- ***Chapter 3*: *Using the GHOST SDK*** describes how to use *GHOST SDK* components to create a haptic environment.
- ***Chapter 4*: *Adding Graphics with GHOSTGL*** describes the optional GHOSTGL library that allows you to easily add OpenGL rendering to a GHOST SDK program.
- ***Chapter 5*: *The HapticView Framework*** describes how to integrate GHOST SDK and the PHANTOM Mouse Driver with the MFC window framework.
- ***Chapter 6: Extending the GHOST SDK*** describes how to add new capabilities to the *GHOST SDK.*
- ***Chapter 7: Demonstration and Example Applications*** describes the demonstration programs provided with the *GHOST SDK*. Source code is provided for most of these programs and can be used to understand the *GHOST SDK's* capabilities.
- ***Appendix A: Compiling GHOST SDK Programs in Windows NT*** explains how to set up Microsoft Visual C++ projects to use with *GHOST SDK*.
- ***Appendix B: PHANTOM Configurations*** explains the PHANTOM Configuration Control Panel, as well as the meanings of PHANTOM registry entries,
- ***Appendix C: GHOST Real-Time Issues*** discusses special issues you *must* consider that relate to how the *GHOST SDK* creates a second real-time, process sharing application.
- ***Appendix D: Autocalibration for the PHANTOM Desktop*** discusses adding on the fly autocalibration to a GHOST SDK application.
- ***Appendix E: GHOST Error Codes*** provides a tabular list of the *GHOST SDK* error codes.

How to Use This Guide

The *GHOST SDK Programmer's Guide* uses the following conventions:

Typographical

Italic indicates terms, concepts, or variables, and shows emphasis. It also indicates cross-references to related publications or chapters and sections in this guide.

Bold is used in code samples to highlight places where you can specify your own classes or methods. Text describing classes and methods appears as shown: **gstPHANToM Dynamic**. This font is also used to show code references within text that specifies fields and structures.

Boolean or logical responses to methods appear as **TRUE** or **FALSE.**

`Monospace` identifies code samples. It may also appear to indicate text you can or must enter to perform a particular action.

Special fonts are used to indicate SensAble Technologies products: *GHOST* and PHANTOM.

A **NOTE:** is used to highlight important additional information.

Units

The following dimensions are used throughout the GHOST API:

- ♦ All distances are in millimeters.
- ♦ All masses are in kilograms.
- ♦ Time is in seconds.
- ♦ The default coordinate reference frame is a right-handed reference frame based upon the most commonly used graphics reference frame. The X-axis is horizontal and increases to the right, the Y-axis is vertical and increases upward, and the Z-axis points towards you.
- ♦ All angular measurements are in radians.
- ♦ All forces are in Newtons.

Related Information

This section lists related publications that can provide more information about the *GHOST SDK*, its API reference document, or documented C++ code examples.

- The GHOST *API Reference Guide* provides detailed information on the *GHOST SDK* C++ classes and methods.
- Please visit the SensAble Technologies web page at http://www.sensable.com for news, updates, and support.

Chapter 1: Installing the GHOST SDK

System Requirements

The following system requirements apply to use the GHOST SDK:

- The GHOST SDK runs on Intel processor PCs under Windows NT 4.0 (with Service Pack 6), Windows 2000 (with Service Pack 1), and Windows XP Professional. It can run on the same OSes on AMD processors using a PCI-based parallel port add-in card.
- The GHOST SDK has been tested under Windows NT/Windows 2000/Windows XP using the Microsoft Visual C++ compiler version 6.0. We recommended that this compiler be used with the GHOST SDK.
- For Windows NT/2000/XP a 300 MHz Pentium processor or better is recommended.
- The GHOST SDK distribution requires approximately 50 MB of disk space. The GHOST SDK *does not* specify memory requirements; however, a minimum of 32 MB is recommended for overall system performance.
- The GHOST SDK is distributed on CD-ROM. Distributions are available for both Windows and, soon, for Linux.

Installing on Windows NT/2000/XP

To install the **GHOST SDK** on a Windows NT/2000/XP platform:

1. You must have the **PHANTOM Device Drivers v4.0** installed on your PC in order for your PHANTOM device to be functional. If you have previous versions of PHANTOM Device Drivers installed, please uninstall them before proceeding to the next step. If you have not installed the device drivers, please install **PHANTOM Device Drivers v4.0** now.
2. Login to your computer as administrator (or as a user with administrative privileges).
3. Insert the CD into the CD-ROM drive.

   Run SETUP.EXE from the GHOST SDK folder on the CD (the installation program will start after a few seconds).

   The default directory for installing the **GHOST SDK v4.0** is **C:\Program Files\SensAble\Ghost\v4.0**. The **GHOST SDK** example application source code is located in the **examples** subdirectory. You can find demo program executables in the **C:\Program Files\SensAble\3D Touch Demos** directory. Demo programs also are available from a **3D Touch Demos** program group, found in the Start Menu.

Chapter 2: The GHOST API

The SensAble Technologies Incorporated (STI) General Haptics Open Software Toolkit (*GHOST SDK*) is a C++ object-oriented toolkit that represents the haptic environment as a hierarchical collection of geometric objects and spatial effects. The *GHOST SDK* provides an abstraction that allows application developers to concentrate on the generation of haptic *scenes*, manipulation of the properties of the scene and objects within the scene, and control of the resulting effects on or by one or more haptic interaction devices. Thus, application developers need not be concerned with low-level force and device issues. In addition, the *GHOST SDK* is highly portable, both in terms of supported computational environments and in terms of different haptic interaction devices. It supports the entire family of STI PHANTOM™ haptic interface devices.

Key Features

The *GHOST* Application Programming Interface (API) enables application developers to interact with haptic interaction devices and create haptic environments at the object or effects level. Using the *GHOST SDK*, developers can specify object geometry and properties, or global haptic effects, using a haptic scene graph.

A scene graph is a hierarchical collection (tree) of nodes. The internal nodes of the tree provide a means for grouping objects, orienting and scaling the subtree relative to the parent node, and adding dynamic properties to their subtrees. The terminal nodes (leaves) of the tree represent actual geometries or interfaces. Leaves also contain an orientation and scale relative to their parent nodes.

The terminus of the haptic interaction device is represented as a point within the scene graph. The *GHOST SDK* automatically computes the interaction forces between this point and objects or effects within the scene, and sends forces to the haptic interaction device for display.

Applications can treat the haptic interaction point using the *GHOST SDK* as either 1) the physical location of the haptic interaction device terminus within its physical workspace, or 2) the computed location of the interaction point constrained to the surface of geometric objects. The latter point position is known as the *Surface Contact Point* (SCP). The SCP is used to generate all *GHOST SDK* surface interaction forces.

The *GHOST SDK does not* generate visual representations of objects within the haptic scene graph. If graphics are desired, the application developer is free to use any computer graphics development environment to create them. The *GHOST SDK* does, however, provide graphic callback mechanisms to facilitate integration between the haptic and graphic domains. To date, the *GHOST SDK* has been successfully used with a variety of graphics packages including OpenGL and Open Inventor.

Key features of the *GHOST SDK* include the ability to:

- Model haptic environments using a hierarchical haptic scene graph.
- Haptically render disparate geometric models within the same scene graph.
- Specify the surface properties (for example, compliance and friction) of the geometric models.
- Use behavioral nodes that can encapsulate either stereotypical behaviors or full freebody dynamics.

The *GHOST SDK* also provides:

- General support for the generation of haptic human-computer interfaces, including haptic manipulators for interacting with objects in the haptic scene using force feedback and spatial effects such as springs, impulses, and vibrations.
- An event callback mechanism to synchronize the haptics and graphics processes.
- The ability to automatically parse and use the static geometry of VRML Version 2.0 geometry files to generate haptic scene graphs.
- Extensibility through the subclassing metaphor. Application developers can extend, modify, or replace all object classes.

Building Blocks

The GHOST API is written in the C++ programming language and consists of a set of C++ classes (see Figure 1). The GHOST API is highly extensible through subclassing.

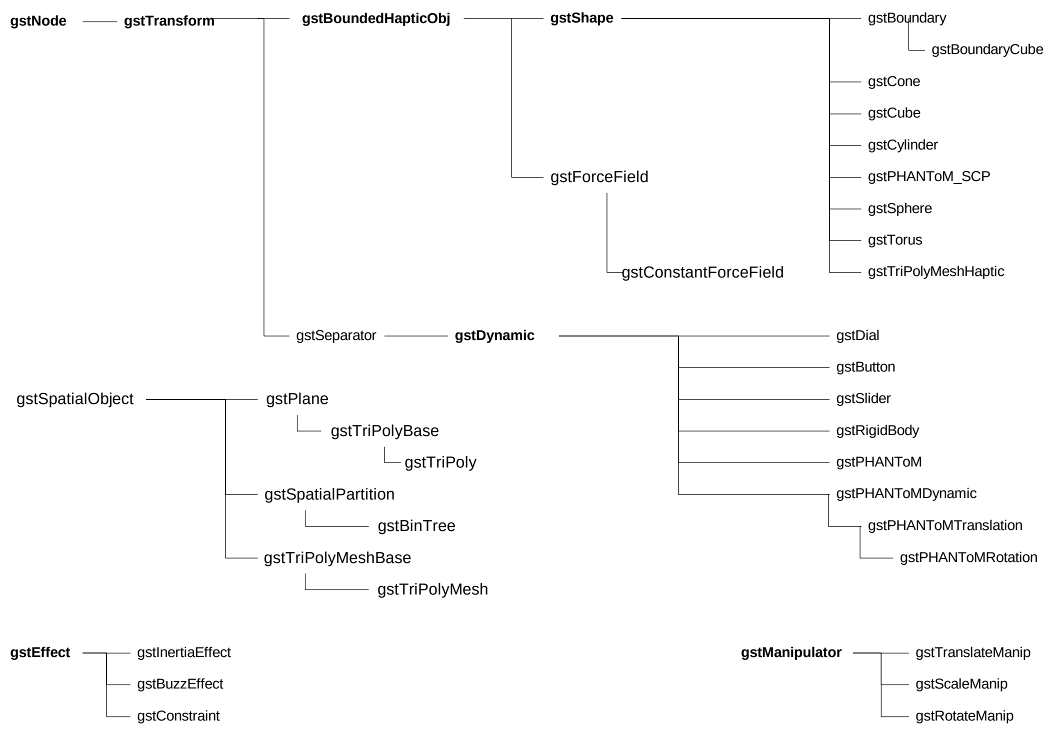


Figure 1. The *GHOST SDK* Class Trees

The *GHOST SDK* classes are organized into nine major class groupings including data types. The remainder of this chapter describes these major class groupings. See the GHOST *API Reference Manual* for a more detailed reference to *GHOST SDK* classes.

Abstract Node Classes

The *GHOST SDK* abstract classes are used as the basis for all other haptic scene graph classes and are *not* part of the haptic scene graph. Abstract classes are listed in the following table.

Abstract Node Classes	Description
gstNode	Abstract base class for all scene graph nodes
gstTransform	Abstract node class that adds 3D transformations and callbacks to nodes
gstBoundedHapticObject	Abstract base class for objects that support a bounding volume
gstShape	Abstract base class for haptic geometry nodes
gstDynamic	Abstract class to add dynamic attributes to a child sub-graph

Hierarchical Node Classes

Hierarchical node classes enable grouping of nodes within the haptic scene graph. Currently, the only hierarchical node class available in the *GHOST SDK* is the Separator class. Application developers can treat nodes below a Separator in the haptic scene graph as a single entity.

Hierarchical Node Class	Description
gstSeparator	Hierarchical node class that allows grouping of nodes under it into a subtree

Geometry Node Classes

The *GHOST SDK* allows objects that use disparate geometry models to be used within the same haptic scene graph. Currently, two major types of geometric representations are supported:

- Geometric primitive objects, such as spheres, cubes, cones, cylinders, and tori
- Objects built using polygonal tri-meshes.

All geometry classes subclass the **gstShape** abstract class and are the *only* node classes that allow non-uniform scaling. Geometry node classes are listed in the following table.

Geometry Node Classes	Description
gstCube	Cube shape class
gstCone	Cone shape class.
gstCylinder	Cylinder shape class
gstSphere	Sphere shape class
gstTorus	Torus shape class
gstTriPolyMeshHaptic	Haptic triangle mesh class

Dynamic Property Node Classes

Dynamic property nodes can be used in the haptic scene graph to add dynamic behaviors to any object geometry. Dynamic properties allow objects to move and reorient in response to external forces. Dynamic property classes are listed in the following table.

Dynamic Property Node Classes	Description
gstButton	Dynamic button class
gstDial	Dial dynamic class
gstSlider	Slider dynamic class
gstRigidBody	Rigid body dynamics class

Haptic Interface Device (PHANTOM) Node Classes

The haptic interface device classes support two nodes for all physical haptic interface devices the *GHOST SDK* handles. Using PHANTOM device classes, an application developer can associate a node containing information about the device's actual state and position; or a node containing spatial information about the device, including its surface contact point (SCP). The SCP is the projection of the actual PHANTOM device end-point position onto the surface of a haptic geometry object. Haptic interface device classes are listed in the following table.

Haptic Interface Device Node Classes	Description
gstPHANToM	Represents the PHANTOM device interface in the scene graph
gstPHANToM_SCP	Represents the surface contact point (SCP) in the scene graph. This node is dependent on its associated gstPHANToM node

The following classes are provided to support generation of boundaries (bounding volumes) for the PHANTOM device workspace.

Additional Haptic Interface Device Node Classes	Description
gstBoundary	Abstract base class for PHANTOM boundary
gstBoundaryCube	Sub-class of gstBoundary for creating cube-like bounding volumes

Effects Node Classes

The *GHOST SDK* includes a series of classes that support the generation of spatial haptic effects for the development of haptic human-computer interfaces. This allows haptic stimuli *not* directly associated with object geometry. Effect classes that define spatial effects are listed in the following table.

Effect Node Classes	Description
gstEffect	Abstract base class for PHANTOM device spatial events
gstBuzzEffect	Creates buzzing (e.g. vibration) effects for the PHANTOM device end-point
gstConstraintEffect	Constraints the PHANTOM device end-point position to a point, line, or plane using a spring-damper connection
gstInertiaEffect	Adds inertia and/or viscosity to the PHANTOM device end-point

Manipulator Node Classes

The *GHOST SDK* manipulator node classes let you transform (i.e., translate, rotate and scale) geometric objects with the PHANTOM device and generate haptics effects during the transformation. Manipulator node classes are listed in the following table.

Manipulator Node Classes	Description
gstManipulator	Abstract base class for node manipulators
gstTranslateManip	Translation node manipulator
gstRotateManip	Rotation node manipulator
gstScaleManip	Scaling node manipulator

The PHANTOM Dynamic Node Classes

The PHANTOM dynamic classes use an associated PHANTOM device to control the position and orientation of the children of the PHANTOM dynamic node. Other PHANTOM nodes in the scene may interact with the children of the PHANTOM dynamic node, and reaction forces from this interaction are applied to the PHANTOM dynamic node. PHANTOM dynamic node classes are listed in the following table.

PHANTOM Node Classes	Description
gstPHANToMDynamic	Abstract base class for PHANTOM Dynamic nodes
gstTranslateDynamic	PHANTOM dynamic that uses PHANTOM to control the position of its subtree in the scene graph
gstRotateDynamic	PHANTOM dynamic that uses PHANTOM to control the position and orientation of its subtree in the scene graph

Scene Classes

The Scene class provides the basis for the *GHOST SDK* process, or simulation loop. A haptic scene graph is enabled for interaction when it is attached to a scene object and when the **startServoLoop()** method is called. The gstDeviceIO class provides for low-level access to the PHANTOM, as well as the ability to develop your own servo loop.

Scene Class	Description
gstScene	Manages the haptic scene graph and simulation
gstDeviceIO	Low-level access to the PHANTOM

Data Types

A variety of data types are defined and used in the GHOST API. Data types are listed in the following table.

Data Types	Description
gstNodeName	String class for storing character name of node
gstPlane	Plane class
gstPoint	Cartesian three dimensional point class
gstTransformMatrix	Homogeneous 4x4 transformation matrix
gstVertex	Triangular mesh vertex class
gstEdge	Triangular mesh edge class
gstTriPoly	Triangular mesh polygon class
gstTriPolyMesh	Triangular mesh class
gstRay	Ray class
gstLine	Line class
gstVector	Cartesian three dimensional vector class

Chapter 3: Using the *GHOST SDK*

This chapter describes the structure of a typical *GHOST SDK* application and key *GHOST SDK* programming concepts. These concepts are described through simple examples and figures supported by detailed explanations.

Prototypical Application

As shown in Figure 2, a typical application using the *GHOST SDK must*:

- Create a haptic environment through the specification of a haptic scene graph
- Start the haptic simulation process (the servo control loop)
- Perform application-specific (core) functions that include the generation and use of computer graphics
- Communicate with the haptic simulation process as needed
- Perform clean-up operations when the application ends

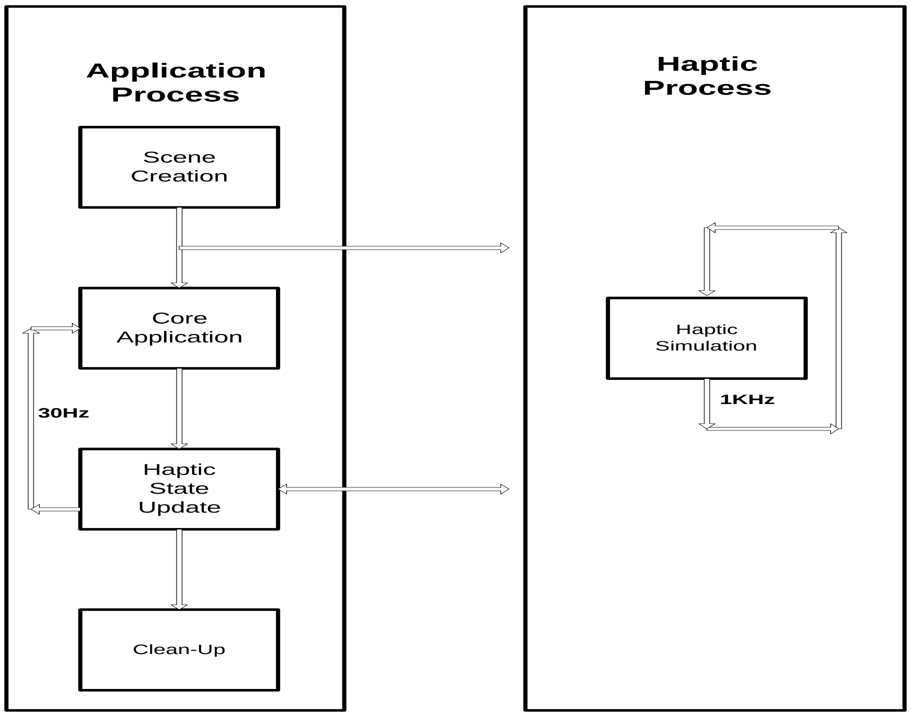


Figure 2: Typical Application Using the *GHOST SDK*

NOTE: A typical *GHOST SDK* application consists of at least two processes: a real-time *GHOST SDK* process that handles the haptic simulation, and the application-level process. The *GHOST SDK* automatically sets up and manages these processes.

Hello World

The following program is provided as a simple example of a *GHOST SDK* application. The application creates a sphere with default orientation (the object is centered at the origin) and surface properties. The radius of the sphere is set to 20 millimeters. The application runs until the servo loop is killed using the hardware enable switch. No graphics are created.

```
#include <stdlib.h>
#include <gstBasic.h>
#include <gstSphere.h>
#include <gstPHANToM.h>
#include <gstSeparator.h>
#include <gstScene.h>

main() {
  gstScene *scene = new gstScene;
  gstSeparator *root = new gstSeparator;
  gstSphere *sphere = new gstSphere;
  sphere->setRadius(20);
  gstPHANToM *phantom = new gstPHANToM("PHANTOM name");
  root->addChild(phantom);
  root->addChild(sphere);
  scene->setRoot(root);
  scene->startServoLoop();
  while (!scene->getDoneServoLoop()) {
      // Do application loop here.
      // Could poll for keyboard input
      // and when user inputs to quit
      // application call
      // scene->stopServoLoop();
  }
}
```

The remainder of this chapter describes a more complicated *GHOST SDK* application and related concepts in more detail.

NOTE: See **Appendix C** for an explanation of the "**PHANTOM name"** argument**.**

Nodes

Nodes are the basic building blocks used in *GHOST SDK* to create haptic environments. The *GHOST SDK* class library defines many types of nodes that can be organized into several categories. The two essential node types are geometry and PHANTOM nodes. These nodes represent the physical geometry of objects in the haptic scene (for example. cube, cylinder, etc.) and the PHANTOM haptic interface respectively. In addition, there are grouping nodes, referred to as separators, which maintain a list of pointers to one or more other nodes. The nodes in this list are considered children nodes. Conversely, the grouping node is considered the parent of its children. There are also dynamic nodes that group nodes and in addition, add dynamic state to the nodes grouped below it. All nodes in the *GHOST SDK* also maintain a position, orientation and scale.

Node Graphs

A node graph is a collection of nodes forming a hierarchical data structure or tree. Thus, one node forms the root of the tree and all other nodes in the node graph are descendents of the root node. Fig 3A and Fig 3B show an example of a robot made of geometric primitives and its associated *GHOST SDK* node graph.

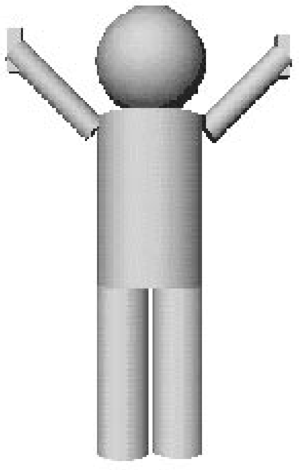


Figure 3A: Robot Node Graph Example

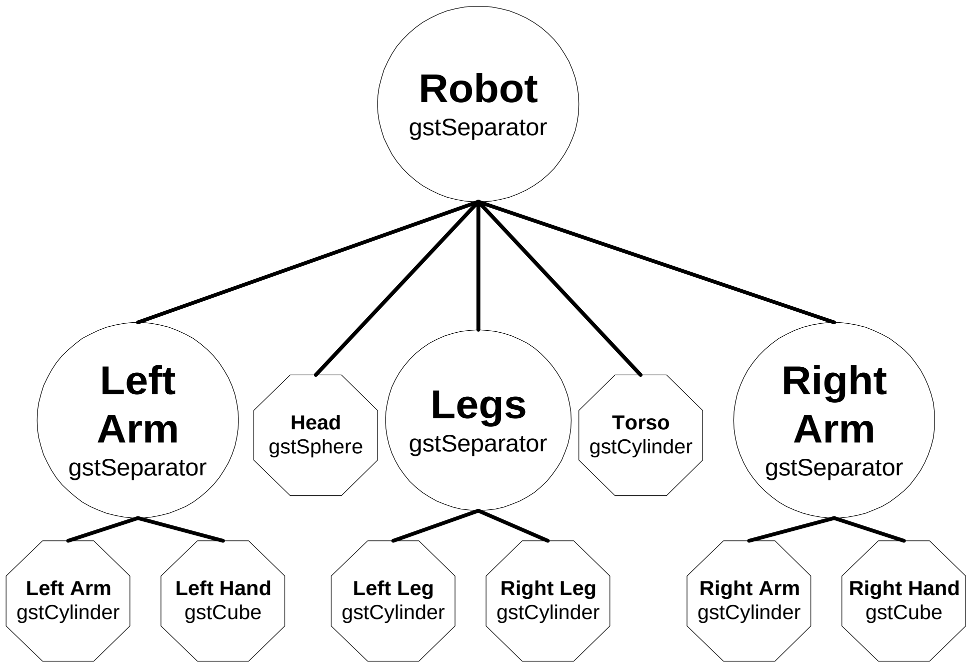


Figure 3B: Robot Node Graph Example

The internal nodes of a tree provide a means to group objects and position, orient, and scale the subtree relative to the parent node. Any dynamic internal nodes also add dynamic properties to the subtrees below them. The terminal nodes (leaves) of the tree represent the actual geometry and PHANTOM haptic interfaces in the scene. Leaves also contain a position, orientation, and scale relative to their parent nodes.

The geometric transformations available at each node create a separate reference frame for each node in the node graph. The result is that a child node's transformation is given relative to its parent node. The root of the node graph represents the global (world) reference frame to which all nodes in the node graph are referenced. The position, orientation and scale of a node in the tree relative to the world reference frame is thus the accumulation of the transforms of the node's ancestors, from the root node to the parent node, with the transform of the node in question. By using a tree data structure and by exploiting the fact that transformations are relative to child and parent, objects can be grouped logically into a subtree, and the entire group can be transformed by transforming the root of the subtree.

For example, in Figure 3A, a robot is comprised of arms, hands, legs, torso, and a head. Nodes representing the geometry of a hand and arm are organized into a subtree and treated as a complete arm. To move this robot arm up or down, the application developer needs *only* to rotate the root of the arm subtree. This operation will rotate all the arm subobjects within the subtree while maintaining the orientation and placement of the arm pieces relative to each other, thus simulating the effect of movement.

The node graph may contain dynamic nodes. Dynamic nodes are the only nodes in the scene graph whose geometric transformations are dynamically updated by the haptic simulation process instead of by the user. These nodes allow the geometries of the node graph to take on realistic dynamic properties affecting the accumulated transforms of all nodes in the subtree or subtrees (each child of dynamic forms a subtree) below the dynamic node. For example, the child nodes of a **gstDial** dynamic node could represent the geometry of a knob. The **gstDial** dynamic node gives the knob the dynamic properties of a dial. In addition, the dynamic nodes allow PHANTOM haptic interfaces to be used as input/output devices in the scene.

NOTE: **The *GHOST SDK's* node graph paradigm is similar to the representation used by many computer graphics applications, but there is an important difference: subtrees cannot be attached to multiple nodes in the tree. This implies that subtrees cannot be reused, and thus each node should be reachable by only one unique traversal path from the root. In addition, accumulation of node transformations in *GHOST SDK* is strictly vertical in nature, traversing from parent to child starting at the root. Traversal state does not accumulate horizontally as it does in some graphics packages.**

The Haptic Scene Graph

The gstScene class maintains a pointer to the root of a node graph that becomes the scene database used for haptic simulation. The node graph pointed to by the gstScene object is referred to as the haptic scene graph or scene graph. Only objects within the scene graph become part of the *GHOST SDK* haptic simulation.

The Haptic Simulation Process

Once the haptic scene graph is built, a **gstScene** class object manages the tree and the haptic simulation process. Every *GHOST SDK* application should contain exactly one instance of a **gstScene** object. The **gstScene** instance maintains a pointer to the haptic scene graph and performs the haptic simulation by repeating the servo loop at a rate of 1kHz. The servo loop then does the following:

- Updates the state of the **gstPHANToM** nodes in the scene graph representing the PHANTOM
- Updates the dynamic state of all **gstDynamic** objects
- Detects collisions of **gstPHANToM** nodes with geometry nodes in the scene graph
- Sends resultant forces back to all **gstPHANToM** nodes in the scene graph.
- If application requests graphics and/or event callbacks then data is prepared for callbacks and application process is then allowed to execute callbacks of objects with new state.

The Application Loop

The call to **gstScene::startServoLoop** creates the real-time haptic simulation process and immediately returns control to the application. After returning from the **gstScene::startServoLoop** method, the application must begin its own application loop. It can then perform its own actions and interact with the haptic process. The haptic simulation process continues to run until it is explicitly stopped using the **gstScene::stopServoLoop** method, an error occurs in the servo loop, or the application exits. The interaction between the application and the haptic simulation process may be accomplished using either *GHOST SDK* methods to directly set and query the state of the haptic scene graph and the haptic simulation, or using *GHOST SDK* callback mechanisms.

Callback mechanisms, described in later sections of this chapter, provide an efficient, asynchronous mechanism for updating the state of the application process when areas of interest in the haptic scene graph change (i.e., because of human or PHANTOM haptic interface device interactions, or because of *GHOST SDK* object dynamic properties). The most common use and need for callback mechanisms is to synchronize an application's visual representation or scene graph with that of the haptics scene graph.

Required Post Application Clean-Up

When the application is ready to exit, the haptic simulation must be stopped using **gstScene::stopServoLoop** and any application-specific cleanup required is performed by the application.

Creating Haptic Environments

Using the node graph paradigm, application developers assemble the individual parts of *GHOST SDK* to create a meaningful haptic environment. The process of building an environment involves two basic steps:

- Building a scene graph made up of geometry, separator, and dynamic nodes.
- Adding one or more PHANTOM haptic interfaces to the scene.

Static Haptic Environments

A static haptic environment is one in which objects *do not* move, versus one in which objects move in response to touch or some object-specific dynamic property. Once a static haptic scene graph is created, the objects maintain their relative positions to each other unless the application explicitly modifies the transformation of a node in the scene graph.

The nodes that can comprise a static scene are limited to separator nodes and to geometry nodes. Separator nodes are derived from the base separator class, **gstSeparator**. This class maintains a list of multiple children nodes and is used exclusively as the internal nodes of a static scene graph. The leaf nodes of the static scene graph are similarly limited to geometry nodes (e.g., **gstCube**, **gstCone**, etc.). Geometry nodes are derived from the **gstShape** class to represent the geometry of physical objects in the haptic scene graph. Geometry nodes *cannot* have children.

gstShape and **gstSeparator** classes are derived from the gstBoundedHapticObj, **gstTransform** and **gstNode** classes, and thus share a set of common properties. These properties and the specific methods for each class are described throughout the remainder of this chapter.

Common Haptic Scene Graph Node Properties

All *GHOST SDK* nodes are derived from the **gstNode** and **gstTransform** classes and share a set of common properties and methods.

Properties and Methods Acquired from gstNode

Class Type. The **gstNode** class allows all nodes to query for their class and derived class types using the **isOfType** virtual method. For example, to test if **node1** is derived from the **gstTransform** class, the application developer would use the following:

```
node1->isOfType(gstTransform::getClassTypeId());
```

Or the equivalent action could be performed by:

```
gstTransform::staticIsOfType(node1->getTypeId());
```

Node Name. The **gstNode** class allows the application to name individual nodes and query the scene graph for a node by its name. The following code example gives a **gstSeparator** node a name, places the node into the scene graph, and finds a pointer to this node by querying the root node of the scene graph.

```
gstSeparator *armSeparator = new gstSeparator;
gstSeparator->setName(gstNodeName("armNode"));

anyNodeInSceneGraph->addChild(armNode);

gstSeparator *ptrToArmNode = rootOfSceneGraph->
getByName(gstNodeName("armNode"));
```

Node Existence. The **gstNode** class allows the application to query a node's existence in the haptic scene graph. A node is considered to be in the scene graph if it is a descendent of the root pointed to by the **gstScene** object. The root node is set through the **gstScene::setRoot** method and defines the scene graph. The **gstNode::getInSceneGraph** method returns **TRUE** if the node resides in the current haptic scene graph; otherwise, it returns **FALSE**. The following example shows how a **gstCube** node returns **FALSE** before being put into the scene graph and **TRUE** afterwards:

```
gstScene *scene = new gstScene;
gstSeparator *separator1 = new gstSeparator;
gstCube *cube1 = new gstCube;
separator1->addChild(cube1);

cube1->getInSceneGraph();  // This returns FALSE

// Make node graph with root separator1 scene graph
scene->setRoot(separator1);

cube1->getInSceneGraph();  // This now returns TRUE
```

Properties and Methods Acquired from gstTransform

Geometric Transformation. As its name implies, the **gstTransform** class adds geometric transformation properties to *GHOST SDK* scene graph nodes. These properties are used to generate a local reference frame for the node. In particular, each node is given the composite transform of a scale, rotation, and translation applied in that order to each node. Nodes that are also derived from the **gstSeparator** class have a center of rotation geometric transformation.

NOTE: Non-uniform scaling is permitted *only* for nodes derived from **gstShape**, such as **gstCube**.

Methods are provided to allow a node to be scaled, rotated, and translated (e.g. **translate, rotate, and scale**). These methods are additive, that is, they accumulate the new scale, translate, or rotate actions with existing geometric transformation values for the node.

The methods that are prefixed with *set* (e.g. **setTranslate, setRotate, setScale**) override any previous geometric transformation values. For example, the **setTranslate** method resets the translation value to that of a new translation represented by the **gstPoint** argument passed into **gstTransform::setTranslate**.

The *local composite* transform (scale, rotation, and translation matrixes, or **SRT**) for each node is stored in a 4x4 homogeneous matrix. This matrix transforms points and vectors between the reference frame of the node and that of its parent. The state of the matrix can be queried and set through the **gstTransform::getTransformMatrix** and **gstTransform::setTransformMatrix** methods. The **gstTransform::fromLocal** and **gstTransform::toLocal** methods are used to transform a **gstPoint** or **gstVector** between the local (nodes') reference frame and the parent's reference frame.

Also, each node stores a *cumulative transform* as a 4x4 matrix that represents the transformation of the node's reference frame to or from the world reference frame. The cumulative transform of a node is the accumulation of the geometric transformations of the ancestor nodes starting from the scene graph root node to the node in question. Its state may be read using the **gstTransform::getCumulativeTransformMatrix** method. The **gstTransform::toWorld** and **gstTransform::fromWorld** methods transform a **gstPoint** or **gstVector** between the local- and world-reference frames. All 4x4 homogeneous transformation matrices in the *GHOST SDK* have the following form:

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}$$

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

where the 3x3 submatrix represents rotation and scale, relative to the parent reference frame or world reference frame in the case of a "cumulative transform". Specifically the rows of the submatrix represent the coordinates in the base frame of vectors along the axes of the transformed reference frame. The following vector represents the translation relative to the parent reference frame.

$$\begin{pmatrix} t_x & t_y & t_z \end{pmatrix}$$

To transform a homogenous column vector, **p1$_{\text{local}}$** from the local reference frame to the world reference frame using **gstTransform::fromLocal**, the following geometric operations are performed:

```
gstSeparator:     p1parent= anySeparatorNode->fromLocal(p1local);
```

$$p1_{Parent} = p1_{local}^{T}\ (CSRTC')$$

```
gstShape:         p1parent= anyShapeNode->fromLocal(p1local);
```

$$p1_{Parent} = p1_{local}^{T}\ (SRT)$$

C is the translation for the center of rotation, and **C'** is the inverse of the center translation. **S**, **R,** and **T** represent the scale, rotation, and translation matrices.

NOTE: Under this convention, the translation matrix has the translation on the bottom row of the matrix. Some conventions place the translation in the last column of the 4x4 homogenous transform matrix. Also note that, *only* nodes derived from **gstShape** are allowed a non-uniform scale.

Event Callbacks. The **gstTransform** class also implements event and graphics callback mechanisms described in *Graphics and Events Callback* section of this guide.

Dynamic Dependents. The **gstTransform::getDynamicDependent** method is explained in *Dynamic Objects in the Scene Graph* section.

Properties and Methods Acquired from gstBoundedHapticObj

The gstBoundedHapticObj allows the developer to specifying a bounding volume for its subclasses (gstShape or gstForceField). The bounding volume is used to minimize the number of PHANTOM-object interactions that are evaluated in each servoloop pass. PHANTOM interactions are only evaluated for objects in which the PHANTOM position is determined to be inside its specified bounding volume.

The gstBoundedHapticObj points to an instance of the gstBoundingVolume class (gstTransformedBoundingSphere or gstTransformedBoundingBox) to represent the bounding volume of the instance. The application can specify the type and parameters of the bounding volume to be used for an instance of a gstBoundedHapticObj through the boundBySphere or boundByBox methods. It is the responsibility of the application to maintain a good bounding volume.

For optimization during evaluation, the bounding volume also maintains its representation in world coordinates. If a boundedHapticObject instance changes its shape or orientation, it should also call the setNeedsUpdate(TRUE) method to ensure that its world coordinate representation is updated before its next evaluation.

Geometry Nodes

Nodes derived from the **gstShape** class represent the physical geometry of an object to be haptically rendered (for example, cube, sphere, and triangular mesh).

Geometric objects simulated with the *GHOST SDK* are currently limited to geometries consisting of rigid surfaces. Thus, when the position of a **gstPHANToM** node (representing the physical position of a PHANTOM haptic interface end-point) passes through the surface of a **gstShape** object, the **gstShape** object does *not* deform – the geometry representing the object. Instead, a Surface Contact Point (SCP) is maintained for each **gstPHANToM** node; the SCP is forced to a point on the surface of any object intersected by that **gstPHANToM**. The SCP, therefore, never penetrates the surface of an object, even if the position of the associated **gstPHANToM** node does penetrate the logical boundaries of the surface.

When computing and sending a force representative of the **gstPHANToM** object touching an intersected **gstShape** node, the force normal to the **gstShape** surface is calculated using a spring-dashpot (spring-damper) model for the surface. The force is proportional to the difference in the position of the SCP from the actual position maintained in the **gstPHANToM** node. In addition, force components tangential to the intersected surface are computed using a stick-slip friction model. Figure 4 below shows how the SCP is used to track the surface contact point and calculate contact forces for a gstPHANToM node.

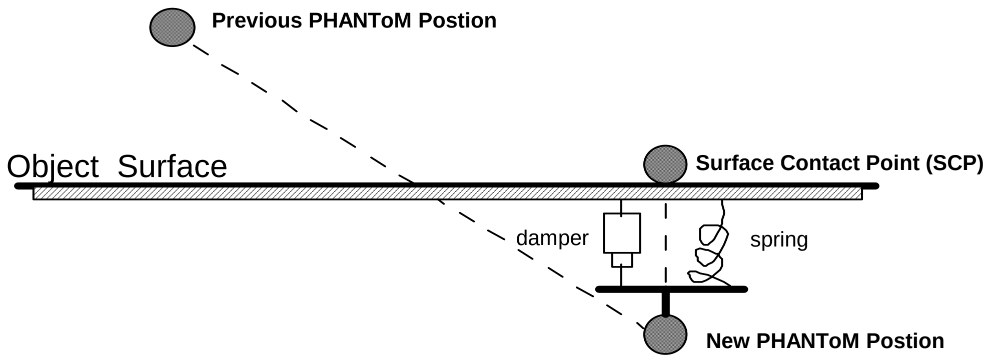


Figure 4**:** The SCP

You can set several properties in the *GHOST SDK* to support this contact force model. These properties are the spring and damping constants used in the spring-dashpot model connecting the SCP to the **gstPHANToM** position, and the coefficients of dynamic and static friction. The properties are stored as **surfaceKspring**, **surfaceKdamping**, **surfaceFdynamic**, and **surfaceFstatic**, and can be queried or set through the corresponding **get** or **set** methods. Each property contains a static default value for the class, which all instances are based upon. These defaults may be queried and set through **getDefault** and **setDefault** methods.

Two other properties handled by **gstShape** are *bounding sphere* and *contact state*. The **gstShape::getBoundingSphereInfo** method returns the center coordinates in the world-reference frame and the radius of the bounding sphere. The contact state of an object is accessible through the **gstShape::isInContact** method. This method returns **TRUE** if any **gstPHANToM** node is touching the **gstShape** node; otherwise, it returns **FALSE**.

Hierarchical Nodes

GHOST nodes derived from the **gstSeparator** class form the internal nodes of the haptic scene graph. Aside from the common properties shared by all nodes in the scene graph, the **gstSeparator** class can be used to add, query, and remove children. These operations expect pointers to the instances of the nodes. The methods **gstSeparator::addChild**, **gstSeparator::removeChild**, and **gstSeparator::getChild** receive and return pointers to the instance of the child node in question.

NOTE: It is invalid to add a node more than once to any **gstSeparator** node in the scene. Some graphics toolkits allow developers to create an object once and use a pointer to the object in a scene graph multiple times to represent separate graphic instances of the object. For example, a robot with identical right and left hands might have the same pointers to the hand subtree for both hands. This operation *cannot* be done with the *GHOST SDK*; since geometries can be touched, they *must* maintain a contact state that is distinct for each object. Using an object in multiple places in the scene graph can result in invalid contact states when the PHANTOM haptic interface is touching an object.

The descendants of a **gstSeparator** have reference frames that are defined with respect to the reference frame of the **gstSeparator** node. If a child of a **gstSeparator** node is translated, the translation is performed along the coordinate axis of the **gstSeparator** node. This is because the translation happens before the rotation specified for the node. Before a node has its rotation applied, the axis of the node corresponds directly with the axis of its parent. Note that, rotations and scales are best thought of as around the axis of the node in question, not the parent. Consider the example shown in Figure 5. This figure presents the top view of a cube sitting on a desk in front of the viewer. The cube is rotated 45 degrees with respect to the front edge of the desk. To move the cube 50 units to the right, the developer should translate the cube down the x-axis of the desk by 50 units. This action should be thought of as moving the cube down the x-axis of the desk, *not* down the x-axis of the cube itself.

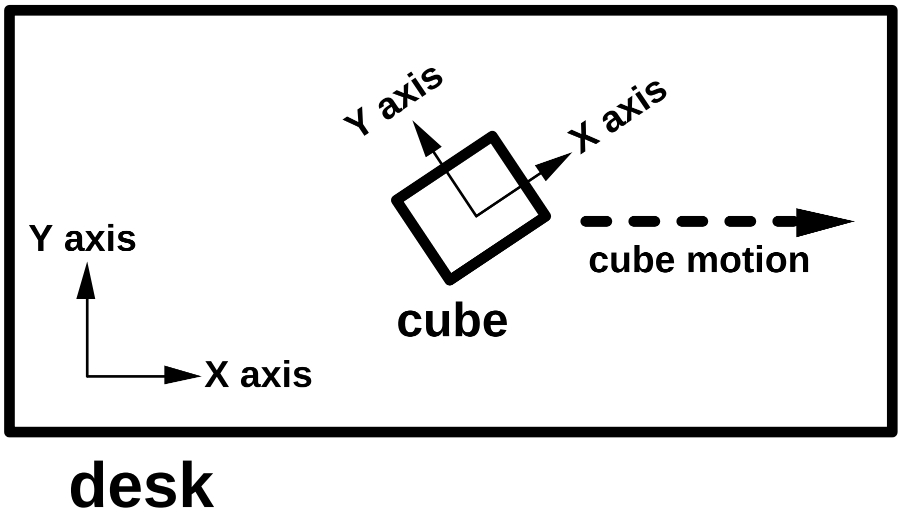


Figure 5: Reference Frame Example

The following code implements this example:

```
gstSeparator *desk = new gstSeparator;
gstCube *cube = new gstCube;

desk->addChild(cube);

cube->rotate(gstVector(0.0,1.0,0.0), PI/2);
cube->translate(50.0,0.0,0.0);
```

Thus, **cube->translate** is a translation with respect to the parent reference frame. In the cube example, the parent reference frame is the world reference frame.

Static Scene Example

The following example program (included in the *GHOST SDK* distributions) builds a robot haptic scene graph to illustrate the proper use of the classes discussed in this chapter. The code creates a simple robot comprising primitive geometries (cylinders, sphere, and cubes) arranged in a hierarchical scene.

```
// Robot static scene example.
// SensAble Technologies, Inc. Copyright 1997
// Written by Chris Tarr

#include <stdlib.h>
#include <gstBasic.h>
#include <gstScene.h>
#include <gstCube.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>

void main ()
{
       // Create root separator of scene graph for robot
       gstSeparator *robot = new gstSeparator;
       robot->setName(gstNodeName("robot"));

       // Create torso
       gstCylinder *torso = new gstCylinder;
       torso->setHeight(60.0);
       torso->setRadius(20.0);
       // Position by default is at origin.  Keep it there
       // Add torso to root
       robot->addChild(torso);

       // Create head
       gstSphere *head = new gstSphere;
       head->setRadius(20.0);
       head->setTranslate(0.0,50.0,0.0);
       // Add head to root
       robot->addChild(head);
```

```
// Create legs
// NOTE: We must create distinct objects
//  for each leg.  Pointers can only be
//  used once in scene graph!
gstCylinder *rightLeg = new gstCylinder;
rightLeg->setRadius(10.0);
rightLeg->setHeight(60.0);
rightLeg->setTranslate(10.0,-60.0,0.0);
robot->addChild(rightLeg);

gstCylinder *leftLeg = new gstCylinder;
leftLeg->setRadius(10.0);
leftLeg->setHeight(60.0);
leftLeg->setTranslate(-10.0,-60.0,0.0);
robot->addChild(leftLeg);

// Create complete arms

// Create right arm
gstCylinder *rightArm = new gstCylinder;
rightArm->setRadius(5.0);
rightArm->setHeight(40.0);
rightArm->setTranslate(20.0,0.0,0.0);
// Rotate 90 degrees around z-axis
rightArm->setRotate(gstVector(0.0,0.0,1.0),M_PI/2.0);

// Create right hand
gstCube *rightHand = new gstCube;
rightHand->setHeight(5.0);
rightHand->setWidth(15.0);
rightHand->setLength(5.0);
// Rotate 45 degrees around z-axis
rightHand->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
rightHand->setTranslate(40.0, 0.0, 0.0);

// Group right arm parts into rightArmSep
gstSeparator *rightArmSep = new gstSeparator;
// Give rightArmSep a name
rightArmSep->setName(gstNodeName("rightArmSep"));
rightArmSep->addChild(rightArm);
rightArmSep->addChild(rightHand);
rightArmSep->setTranslate(20.0,25.0,0.0);
// Rotate 45 degrees on z-axis
rightArmSep->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
// Put rightArmSep into root node
robot->addChild(rightArmSep);

// Create left arm
gstCylinder *leftArm = new gstCylinder;
leftArm->setRadius(5.0);
leftArm->setHeight(40.0);
leftArm->setTranslate(20.0,0.0,0.0);
// Rotate 90 degrees around z-axis
leftArm->setRotate(gstVector(0.0,0.0,1.0),M_PI/2.0);
```

```
// Create left hand
gstCube *leftHand = new gstCube;
leftHand->setHeight(5.0);
leftHand->setWidth(15.0);
leftHand->setLength(5.0);
// Rotate 45 degrees around z-axis
leftHand->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
leftHand->setTranslate(40.0, 0.0, 0.0);

// Group left arm parts into leftArmSep
gstSeparator *leftArmSep = new gstSeparator;
leftArmSep->addChild(leftArm);
leftArmSep->addChild(leftHand);
leftArmSep->setTranslate(-20.0,25.0,0.0);
// Rotate 45 degrees on z-axis
leftArmSep->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
// Make mirror image across y-z plane so
//  do 180 rotation around y axis
// Note: use rotate instead of setRotate so that the
//       rotation accumulates with the previous one.
leftArmSep->rotate(gstVector(0.0,1.0,0.0), M_PI);
// Put rightArmSep into root node
robot->addChild(leftArmSep);

// Search for rightArmSep under root and place into nodePtr;
gstNode *nodePtr = robot->getByName(gstNodeName("rightArmSep"));
if (nodePtr == NULL)
      printf("Didn't find rightArmSep under robot node!\n");
else
      printf("Found rightArmSep under robot node!\n");

// Check if rightArmSep is in scene graph.
//  Should not be in scene graph until this node or a
//  parent node is set as root of gstScene node.
if (nodePtr->getInSceneGraph())
      printf("%s is in scene graph\n", (char *)
              nodePtr->getName());
else
      printf("%s is not in scene graph\n", (char *)
              nodePtr->getName());

// Create gstScene object to handle haptic simulation
gstScene *scene = new gstScene;

// Setting robot as root of gstScene makes robot
//  the root of the scene graph.  After setting
//  robot as root of scene, robot and all nodes
//  under robot have been put in scene graph and
//  their state reflects that
scene->setRoot(robot);
```

```
       // Check if rightArmSep is in scene graph
       // Should be in scene graph now since robot is root of scene
       if (nodePtr->getInSceneGraph())
              printf("%s is in scene graph\n", (char *) nodePtr-
>getName());
       else
              printf("%s is not in scene graph\n", (char *) nodePtr-
>getName());

       // Additional code to be added in next section
}
```

Dynamic Objects in the Scene Graph

To use a PHANTOM haptic interface device to actually feel the objects in a scene graph (or to have objects move dynamically in the scene), the developer must include a node of the **gstDynamic** class, minimally as the **gstPHANToM** itself. The **gstDynamic** class adds *dynamic state* to the node. The new state variables are mass, damping, velocity, acceleration, force, torque, angular velocity, angular acceleration, and an inertia matrix. Derived classes of **gstDynamic** use the state variables as needed. On any path from the scene graph root to a leaf in the scene graph, one **gstDynamic** class, at most, should exist; otherwise, a **gstDynamic** will become a descendent of another **gstDynamic**, and this inheritance could generate unpredictable dynamic behavior.

All descendent geometry nodes of a **gstDynamic** node in the scene graph have a pointer to their ancestor **gstDynamic** node. The ancestor gstDynamic node is referred to as the geometry node's *dynamic dependent* node. It gets this pointer from the **getDynamicDependent** method. Using this information, the **gstScene** sends reaction forces and torque to the dynamic dependent of any geometry node touched.

Thus, if a **gstDynamic** node has a descendent **gstCube** node that is touched by a **gstPHANToM,** the opposite reaction force is sent to the **gstDynamic** node, and an appropriate torque is calculated. Using this information and the other dynamic state of the node, the **gstDynamic** calculates a new state for the **gstDynamic** node, and it resets the forces and torque to zero. Since position and orientation are part of the altered state of the **gstDynamic** node, the node can move dynamically in response to forces and torque.

NOTE: Never transform the **gstDynamic** object. The **gstDynamic** object transforms itself and should only be read, not set.

Placing PHANTOM Nodes in the Scene

To actually feel and interact with the geometric objects placed in a haptic scene graph, the application must place a **gstPHANToM** node somewhere in scene. Typically, the **gstPHANToM** node is put in a separator directly on the root of the scene graph. It is usually desirable to have the PHANToMs closely related to the world coordinate system. The **gstPHANToM** node is a derived class of **gstDynamic**, and it is used to inform the **gstScene** object of the PHANTOM haptic interface's end-point position and orientation. This information is used to send forces back to the PHANTOM. The **gstScene** object recognizes **gstPHANToM** nodes in the scene graph and ensures that forces generated from touching geometry objects are sent to the appropriate **gstPHANToM**. The *GHOST SDK* currently supports up to four PHANTOM haptic interface devices in the scene graph.

Every PHANTOM haptic interface device has two reference frames. Manipulating the base of the physical PHANTOM haptic interface device is equivalent to manipulating the *parent reference frame* of the device. Moving the end point and orienting the stylus or thimble is equivalent to manipulating the *reference frame* of the PHANTOM or **gstPHANToM** node. The **gstPHANToM** node transformation represents the PHANTOM device; the orientation and position of the PHANTOM device end-point (user interface) is reflected in the transform of the **gstPHANToM** node. The orientation and position of the PHANTOM haptic interface base are represented by the transform of the parent node of the **gstPHANToM** node.

The following robot example code shows the addition of the **gstPHANToM** node:

```
// Robot static scene example.
// SensAble Technologies, Inc. Copyright 1997
// Written by Chris Tarr

#include <stdio.h>
#include <stdlib.h>
#include <fstream.h>
#include <gstBasic.h>
#include <gstScene.h>
#include <gstCube.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>
#include <gstPHANToM.h>

void main ()
{
       // Create root separator of scene graph for robot
       gstSeparator *robot = new gstSeparator;
       robot->setName(gstNodeName("robot"));

       // By default, PHANTOM encoders are reset, so make sure
       // user has PHANTOM device in reset position
       printf("\nHold PHANTOM in reset position and press enter
              to continue\n");
       getchar();

       gstPHANToM *phantom = new gstPHANToM("PHANTOM name");
       if (!phantom->getValidConstruction()) {
              printf("gstPHANToM constructor failed\n");
              exit;
       }
       // Put phantom into separator.  This separator
       //  should be used to translate and orient the
       //  phantom in the scene.
       // For now leave phantom at origin.
       gstSeparator *phantomSep = new gstSeparator;
       phantomSep->addChild(phantom);

       robot->addChild(phantomSep);

       // Create torso
       gstCylinder *torso = new gstCylinder;
       torso->setHeight(60.0);
       torso->setRadius(20.0);
       // Position by default is at origin.  Keep it there
       // Add torso to root
       robot->addChild(torso);
.
.
.
       // Create gstScene object to handle haptic simulation
       gstScene *scene = new gstScene;

       // Setting robot as root of gstScene makes robot
       //  the root of the scene graph.  After setting
```

```
        //  robot as root of scene, robot and all nodes
        //  under robot have been put in scene graph and
        //  their state reflects that
        scene->setRoot(robot);

        // Check if rightArmSep is in scene graph
        // Should be in scene graph now since robot is root of scene
        if (nodePtr->getInSceneGraph())
              printf("%s is in scene graph\n", (char *)
                      nodePtr->getName());
        else
              printf("%s is not in scene graph\n", (char *)
                      nodePtr->getName());

        // Run simulation.
        if (!scene->startServoLoop()) {
              printf("servoLoop unable to start.\n");
              exit;
        }

        printf("Enter 'q' to quit\n");
        // Keep application running till 'q' is entered.
        while (getchar() != 'q')
              ;

}
```

NOTE: See **Appendix C** for an explanation of the "**PHANTOM name"** argument**.**

gstPHANToM Constructor

Notice that the constructor for **gstPHANToM** takes a character string pointing to a named PHANTOM device specification. This specification encapsulates the details of the particular PHANTOM haptic interface device being used and is generated automatically when the PHANTOM Configuration control panel is used on Windows NT or when the PHANToMConfiguration application is used on SGI IRIX.

On Windows NT, the **gstPHANToM** constructor examines the Windows NT Registry for an entry created either as the default or through the use of the PHANTOM Configuration dialog box.

On SGI IRIX, the **gstPHANToM** constructor searches for the configuration file by:

- Searching the directory in which the executable resides.
- Searching the directory pointed to by the environment variable **PHANTOM_CONFIG_FILES**.
- Searching the installation directory. On SGI IRIX, the installation directory is **/usr/local/SensAble/PHANToMDeviceDrivers**.

If the configuration data is *not* found the Registry (Windows NT) or in any of the above directories (SGI IRIX), the constructor generates an error. Any errors that occur during construction (for example, an invalid initialization file) generate an error to **gstErrorHandler**. For more information on the configuration file, see *Appendix C: PHANTOM Configuration File*.

Return Value

The return value of the **gstPHANToM** constructor is *not* NULL on an error. By default, errors are sent as messages to the screen. These errors can be intercepted through an error callback, discussed in more detail under the heading **gstErrorHandler** later in this chapter. To indicate a valid construction of a

gstPHANToM instance, **gstPHANToM::getValidConstruction** should return **TRUE** if construction was successful.

Resetting PHANTOM Encoders

The PHANTOM haptic interface device's sensors *must* be reset each time the computer is turned on or every time the PHANTOM haptic interface is disconnected in any way from the computer. The position and orientation of the PHANTOM haptic interface device when read from the software is *always* relative to the position in which the sensors were reset. By default, the encoders are reset when the **gstPHANToM** or **gstPHANToMDynamic** node is constructed. The constructors for **gstPHANToM** and **gstPHANToMDynamic** have a second optional parameter, **resetEncoders**, that is **TRUE** by default. If you desire to not have the encoders reset upon construction, set this second parameter to **FALSE** as shown:

```
        gstPHANToM *phantom = new gstPHANToM("PHANTOM name", FALSE);
```

When reset, the PHANTOM device must be held by the user in its *neutral position*. Thus, just prior to construction of a **gstPHANToM** or **gstPHANToMDynamic** node.

NOTE: Place the PHANTOM device in its correct position before proceeding. The above example requires you to press **Enter** when the PHANTOM device is in its neutral position to resume execution.

NOTE: The new PHANTOM Desktop model has an autocalibration feature. It will autocalibrate upon its first use after system startup. If the PHANTOM Desktop does not appear calibrated while using a 3D Touch application after startup, move the PHANTOM through its range of motion, exit and return. It will be calibrated from that point forward. Therefore, it is unnecessary to force the reset of the PHANTOM Desktop

Neutral Position

The device is in the neutral position when all control surfaces are at their right angle positions. The arms and motors should all be at right angles to one another as shown in Figure 6. The reset or sensor zero position (the neutral position) are one and the same. A relative offset, specified in angular (radian) terms, however, can be added to the reset position to accommodate the use of fixtures or other fiduciary positions for very precise positional-angular calibration of the PHANTOM device. If the **<reset angles>** entry in the PHANTOM configuration file is 0,0,0, no offset is added to the reset position. If other angles are specified to use the PHANTOM 3.0 reset arm, the neutral position and reset position are different. In this case, the PHANTOM device *should be* in the appropriate position for the specified angular offset when the **gstPHANToM** or **gstPHANToMDynamic** constructor is called without a second parameter. See *Appendix C: PHANTOM Configuration File* for more information about configuration settings.

Failure to have the PHANTOM device in the correct position when reset not only causes incorrect position and orientation values to be read but, in some cases, can cause the abrupt application of forces and thus compromise safety while the device is being used.

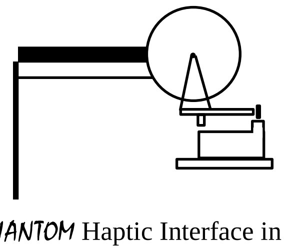


Figure 6: The PHANTOM Haptic Interface in Neutral Position

Boundary Classes

The boundary classes provide optional haptic bounding volumes for the workspace of one or more **gstPHANToM** nodes. Boundary objects behave like ordinary geometry objects, except they are generally used to constrain the PHANTOM device position only inside the object. The *GHOST SDK* provides the

gstBoundary abstract class and the derived **gstBoundaryCube** class for creating cube shaped bounding volumes.

Boundary nodes are added to the scene graph just like any other nodes derived from **gstShape**. The difference is that they also *must* be associated with a **gstPHANToM** node by passing a pointer to the boundary node to the **setBoundaryObj** method of the **gstPHANToM** node. If a boundary object is *not* associated with a particular PHANTOM device, that device *will not* receive any contact forces for the boundary object. In general, a boundary node is placed in the scene graph as a sibling (a child of the same separator) of the associated **gstPHANToM** node. Doing so assures that the PHANTOM device and the boundary share the same reference frame. In certain cases, however, you may desire to place them in separator reference frames. For example, two PHANTOM devices can be in different reference frames but share a common boundary object.

The following code fragment illustrates how to create a boundary cube and associate it with a PHANTOM device:

```
// Make workspace boundaries.  We confine the phantom
// inside the limits of the bounding cube.
gstBoundaryCube *workspaceH = new gstBoundaryCube();
workspaceH->setWidth(WORKSPACE_XMAX-WORKSPACE_XMIN);
workspaceH->setLength(WORKSPACE_ZMAX-WORKSPACE_ZMIN);
workspaceH->setHeight(WORKSPACE_YMAX-WORKSPACE_YMIN);
workspaceH->setPosition((WORKSPACE_XMAX+WORKSPACE_XMIN)/2,
                (WORKSPACE_YMAX+WORKSPACE_YMIN)/2,
                (WORKSPACE_ZMAX+WORKSPACE_ZMIN)/2);
// Bound the phantom object.  If this is not set,
// the phantom will ignore the boundary object.
phantom->setBoundaryObj(workspaceH);
```

NOTE: Only one boundary object can be associated with a given **gstPHANToM node**.

```
// This makes the phantom and boundary
// share the same reference frame.
phantom->getParent()->addChild(workspaceH);
```

Dynamic Haptic Environments

In addition to providing some of the functionality for the **gstPHANToM** class, the **gstDynamic** class is used to add dynamic properties to geometry nodes.

Geometry nodes beneath a **gstDynamic** node in the haptic scene graph are treated as a single object. The **gstDynamic** node adds movement ability. The behavior of the geometric nodes associated with the **gstDynamic** node is determined by the derived **gstDynamic** class. The *GHOST SDK* currently provides four derived **gstDynamic** classes: **gstDial**, **gstButton**, **gstSlider**, and **gstRigidBody**. Forces resulting from interactions with instances of the **gstPHANToM** class are used by **gstDynamic** derived classes to move (affect the dynamic state of) the geometric nodes associated with them. The *GHOST API Reference Manual* describes in more detail the **gstDynamic** derived classes. Additional behaviors can be created through extending the **gstDynamic** class as described in *Chapter 4: Extending the GHOST SDK*.

The subtree under a **gstDynamic** node represents one physically dynamic object (i.e., moving or moveable). For example, a physical dial in the scene may adjust the value of a parameter in the application. Forces generated from a **gstPHANToM** or possibly another geometry object colliding with one of the geometries of the dynamic object are added to the dynamic state of the **gstDynamic**. By integrating the state of the **gstDynamic** this group of geometries can behave like a real physical object.

NOTE: The geometry nodes in the tree below a **gstDynamic** *do not* contribute to the aggregate properties (mass, inertia) handled by **gstDynamic**.

Design of dynamic nodes involves several basic concepts. The first concept is the relationship between dynamics and their children. The second concerns how object transformations are affected by dynamic behavior. The third involves how dynamic objects generate events to notify application processes of state changes.

Children of dynamic nodes

The dynamic node itself is *not* a geometric node and thus does not represent a physical object. Instead, it enables other objects to inherit specific types of dynamic behavior. This design allows flexibility when developing haptic scenes with dynamics; any object or set of objects can be enabled with dynamic properties.

For example, when an instance of a **gstDial** is created, some geometry node *must be* placed below it to represent the physical dial. If a **gstCube** were added as the child of the **gstDial**, the cube would rotate when pushed. It would exhibit the rotational properties of the dial and would be governed by physical forces such as inertia and damping. If, on the other hand, a set of geometry nodes under a **gstSeparator** were added as a subtree of **gstDial**, the objects would collectively rotate around the center of the dial as if they were a single geometry.

Dynamic transformations

When the **gstCube** instance described above is rotated, two areas exist where the rotation could be performed to achieve the desired mathematical effect. The rotation could be applied to the **gstCube**, or it could be applied to the **gstDial**. In the *GHOST SDK*, the transformations are *always* applied to the **gstDynamic** node (i.e., **gstDial**), *not* its children. Conceptually, when the cubic dial is rotated, the entire dial abstraction is said to rotate, *not* the cube.

Thus, when geometry objects are moved in response to being touched, the changes are reflected in the transform of the **gstDynamic** node. For a **gstSlider**, the translation of the **gstSlider** node from its origin represents the offset of the slider. The offset of a **gstButton** from its origin represents how far the button has been pressed. For example, a **gstButton** dynamic has a spring force that is centered at the origin of the dynamic. The **gstButton** allows motion along the z-axis between the origin and its *throw distance*. When the **gstButton**'s physical representation is pushed, the dynamic is translated in the negative z-axis direction. The **gstButton** then moves itself back to the origin under the influence of its restoring spring force. Physically, the **gstButton** can be compared to a spring attached to the origin of the **gstButton** whose natural length is zero. The child of the **gstButton** is whatever physical object, such as a plastic cube, is connected to the end of the spring. When that cube is pushed, the spring is stretched and pulls back toward its rest length with a restoring force. Figure 7 shows a diagram of the button behavior.

Since the state of a **gstDynamic** is reflected in its position and orientation, applications *should not* directly perform transformations on the **gstDynamic**. Since translating a **gstButton** is analogous to stretching or compressing its spring, applications *should not* translate the **gstButton** dynamic to position the button. Instead, the **gstButton** should be added as the child of a **gstSeparator** whose position is the desired location of the button. Physically, the **gstSeparator** can be compared to the plate to which the spring is mounted. To move the button, one would reposition the plate, *not* pull the spring to the desired location.

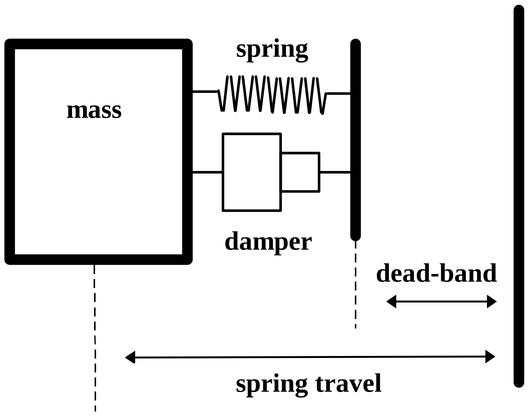


Figure 7. A **gstButton** Behavioral Example

In summary, if a button were comprised from a **gstSeparator**, a **gstDynamic** under that **gstSeparator**, and a geometry under that **gstDynamic**, the transformations would have the following meanings:

Dynamic Transformations	Description
*gstSeparator transform	The position of the gstDynamic; or the physical location of a button.
*gstDynamic transform	The offset of the gstDynamic from its initial resting, or zero, position; or the stretch or compression of the button's spring from its natural length.
*gstShape transform	The offset of the geometry from the natural position of the spring; or the distance from the center of the plastic button piece to the end of the spring.

Event generation

In addition to defining the physical response of an object to the **gstPHANToM**, **gstDynamic** nodes can also generate events to communicate state changes to the application. For example, a **gstButton** node maintains a *pressed or released* state, and a **gstSlider** updates information about the relative position of the slider to its origin. When a **gstDynamic** changes state, such as when a button is toggled from *pressed* to *released*, an event is generated that calls a *user-defined* callback function used to pass event information to the application process. This event mechanism provides a way for haptic behavior to communicate with and affect other processes. For example, a **gstButton** event callback may be programmed to cause the application to quit when the **gstButton** is pressed. See the *Graphic and Event Callback* section later in this chapter for a more detailed explanation of event handling.

Dynamic State

The **gstDynamic** abstract class provides the following state variables for all derived classes to use if needed:

• Mass	– mass in kilograms
• Damping	– Damping in Kilograms/(1000.0*sec)
• ReactionForce	– Newtons
• ReactionTorque	– Newton Meters
• Acceleration	– mm/sec^2
• Velocity	– mm/sec
• AngularAcceleration	– radians/sec^2
• AngularVelocity	– radians/sec

The reference frame they are defined in determines the last six state variables above. Each derived class can determine the frame of reference for these state variables independently. The following information lists existing derived **gstDynamic** classes and describes how they interpret the dynamic state fields of the **gstDynamicGraphicsCBData** structure:

gstDynamic Derived class	Dynamic State Representation
gstButton	ReactionForce – local reference frame ReactionTorque – local reference frame Acceleration – local reference frame Velocity – local reference frame AngularVelocity – Not used AngularAcceleration – Not used
gstDial	ReactionForce – local reference frame ReactionTorque – local reference frame Acceleration – Not used Velocity – Not used AngularVelocity – local reference frame AngularAcceleration – local reference frame
gstSlider	ReactionForce – local reference frame ReactionTorque – local reference frame Acceleration – local reference frame Velocity – local reference frame AngularVelocity – Not used AngularAcceleration – Not used
gstRigidBody	ReactionForce – local reference frame ReactionTorque – local reference frame Acceleration – local reference frame Velocity – world reference frame AngularVelocity – local reference frame AngularAcceleration – Not used
gstPHANToM	ReactionForce – parent reference frame ReactionTorque – parent reference frame Acceleration – not used Velocity – parent reference frame AngularVelocity – Not used AngularAcceleration – Not used
gstPHANToMDynamic	ReactionForce – local reference frame ReactionTorque – local reference frame Acceleration – local reference frame Velocity – local reference frame AngularVelocity – Not used AngularAcceleration – Not used

gstDynamic Example

The following robot code example has been modified to incorporate a **gstDial** node that allows the right arm to rotate about the x-axis when pushed in the world-reference frame:

```
// Robot static scene example.
// SensAble Technologies, Inc. Copyright 1997
// Written by Chris Tarr

#include <stdlib.h>
#include <gstBasic.h>
```

```
#include <gstScene.h>
#include <gstCube.h>
#include <gstCylinder.h>
#include <gstSphere.h>
#include <gstSeparator.h>
#include <gstPHANToM.h>
#include <gstDial.h>

void main ()
{
      // Create root separator of scene graph for robot
      gstSeparator *robot = new gstSeparator;
      robot->setName(gstNodeName("robot"));



      // create right arm
      gstCylinder *rightArm = new gstCylinder;
      rightArm->setRadius(5.0);
      rightArm->setHeight(40.0);
      rightArm->setTranslate(20.0,0.0,0.0);
      // rotate 90 degrees around z-axis
      rightArm->setRotate(gstVector(0.0,0.0,1.0),M_PI/2.0);

      // create right hand
      gstCube *rightHand = new gstCube;
      rightHand->setHeight(5.0);
      rightHand->setWidth(15.0);
      rightHand->setLength(5.0);
      // rotate 45 degrees around z-axis
      rightHand->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
      rightHand->setTranslate(40.0, 0.0, 0.0);

      // group right arm parts into rightArmSep.
      gstSeparator *rightArmSep = new gstSeparator;
      // give rightArmSep a name.
      rightArmSep->setName(gstNodeName("rightArmSep"));
      rightArmSep->addChild(rightArm);
      rightArmSep->addChild(rightHand);
      rightArmSep->setTranslate(20.0,25.0,0.0);
      // rotate 45 degrees on z-axis
      rightArmSep->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
      // Put rightArmSep into root node
      robot->addChild(rightArmSep);

      .
      .
      .

      // create right arm
      gstCylinder *rightArm = new gstCylinder;
      rightArm->setRadius(5.0);
      rightArm->setHeight(40.0);
      rightArm->setTranslate(20.0,0.0,0.0);
      // rotate 90 degrees around z-axis
      rightArm->setRotate(gstVector(0.0,0.0,1.0),M_PI/2.0);

      // create right hand
      gstCube *rightHand = new gstCube;
      rightHand->setHeight(5.0);
```

```
        rightHand->setWidth(15.0);
        rightHand->setLength(5.0);
        // rotate 45 degrees around z-axis
        rightHand->setRotate(gstVector(0.0,0.0,1.0),M_PI/4.0);
        rightHand->setTranslate(40.0, 0.0, 0.0);

        // group right arm parts into rightArmSep.
        gstSeparator *rightArmSep = new gstSeparator;
        // give rightArmSep a name.
        rightArmSep->setName(gstNodeName("rightArmSep"));
        rightArmSep->addChild(rightArm);
        rightArmSep->addChild(rightHand);

        // rotate -90 degrees around y-axis to align with dial.
        //  Remember that dial rotates in local ref. around z-axis.
        //  Dial is rotated 90 around y-axis to orient dial
        //  correctly at shoulder.  Thus arm must be oriented
        //  correctly with with dial to have arm rotate with dial
        //  into correct position.
        rightArmSep->rotate(gstVector(0.0,1.0,0.0),-M_PI/2.0);

        // Rotate right arm up 45 degrees so that arm is pointing
        // upwards instead of straight out.
        rightArmSep->rotate(gstVector(0.0,0.0,1.0),M_PI/4.0);

        gstSeparator *dialSeparator = new gstSeparator;
        // axis of rotation is z-axis of dial, so rotate
        //  z-axis to x-axis of body (equiv. to parent).
        dialSeparator->setRotate(gstVector(0.0,1.0,0.0), M_PI/2.0);
        // translate dial up to position of shoulder.
        dialSeparator->setTranslate(20.0,25.0,0.0);

        gstDial *armDial = new gstDial;
        armDial->setRotate(gstVector(0.0,0.0,1.0),M_PI);
        armDial->addChild(rightArmSep);

        dialSeparator->addChild(armDial);

        // Put rightArmSep into root node
        robot->addChild(dialSeparator);

        .
        .
        .

}
```

Effects, Manipulators, and special PHANTOM classes

The *GHOST SDK* supports the ability to append forces directly to the **gstPHANToM** so that effects such as vibration, viscosity, and constraint can be realized. The *GHOST SDK* also includes manipulators and special **gstPHANToM** nodes that can transform objects using the *PHANTOM* haptic interface and generate force effects based on the *PHANTOM* device imposed transformations, such that the forces are sent back to the *PHANTOM* haptic interface. The **gstEffect** derived classes implement the framework for appending forces to a **gstPHANToM** node; the **gstManipulator** and **gstPHANToMDynamic** derived classes perform transformations and force effects.

gstEffects

The **gstEffect** class allows the *GHOST SDK* to append forces to **gstPHANToM** or **gstPHANToMDynamic** class nodes. This feature can be used to add forces beyond object-surface collision forces. The *GHOST SDK* includes the following **gstEffect** derived classes: **gstBuzzEffect**, **gstConstraintEffect**, and **gstInertiaEffect**. Each class is described in more detail in the *GHOST API Reference Guide*.

To use a **gstEffect** class, the application *must* create an instance, and a pointer to that instance *must be* passed to an instance of a **gstPHANToM** or **gstPHANToMDynamic** through the **gstPHANToM::setEffect** or **gstPHANToMDynamic::setEffect** methods. The effect can is then activated through **gstPHANToM::startEffect** or **gstPHANToMDynamic::startEffect**. A **gstEffect** may be designed to stay active for a specified time interval, or until the **stopEffect** method is called depending on the type of effect class.

The following code example shows how to incorporate an effect in an application:

```
gstBuzzEffect *buzzEffect = new gstBuzzEffect;

phantom->setEffect(buzzEffect);
phantom->startEffect();
.
.
.
phantom->stopEffect();
```

NOTE: Only one effect at a time can be added to each **gstPHANToM** or **gstPHANToMDynamic** node.

gstManipulators

The **gstManipulator** abstract class is implemented in a manner similar to the **gstEffect** class, but the **gstManipulator** is used for a different purpose. It uses the **gstPHANToM** state to manipulate a specified **gstNode** in the scene graph, and it can send reaction forces back to the **gstPHANToM** node. The three derived **gstManipulator** classes are **gstTranslateManipulator**, **gstScaleManipulator**, and **gstRotateManipulator**. These classes are used to translate, scale or rotate, respectively, a specified node in the scene graph using the **gstPHANToM** node, and are designed such that if **gstPHANToM::startManipulator** or if the **gstManipulator::setUseStylusSwitch** have been called with TRUE as its only argument then a depressed stylus switch activates the manipulator. Once the stylus switch is released or **gstPHANToM::stopManipulator** is called, the manipulator becomes inactive. The three classes also calculate resistance forces to their transformation operations and allow those forces to be applied to the **gstPHANToM** node.

For example, the **gstRotateManipulator** restricts the user to a spherical shell. The PHANTOM haptic interface can move around this shell to rotate the object.

Manipulators are convenient in that the scene graph is left intact and a **gstManipulator** is the only object that needs to be created. After the **gstManipulator** is created, the developer must set the pointers in **gstPHANToM** and **gstManipulator** correctly. Figure 8 shows an example scene graph with a manipulator connecting a **gstPHANToM** to a node in the scene graph.

NOTE: **gstManipulators** and **gstEffects** are not formal nodes in the scene graph and have no effect of traversal state. These classes modify existing nodes in the scene graph.

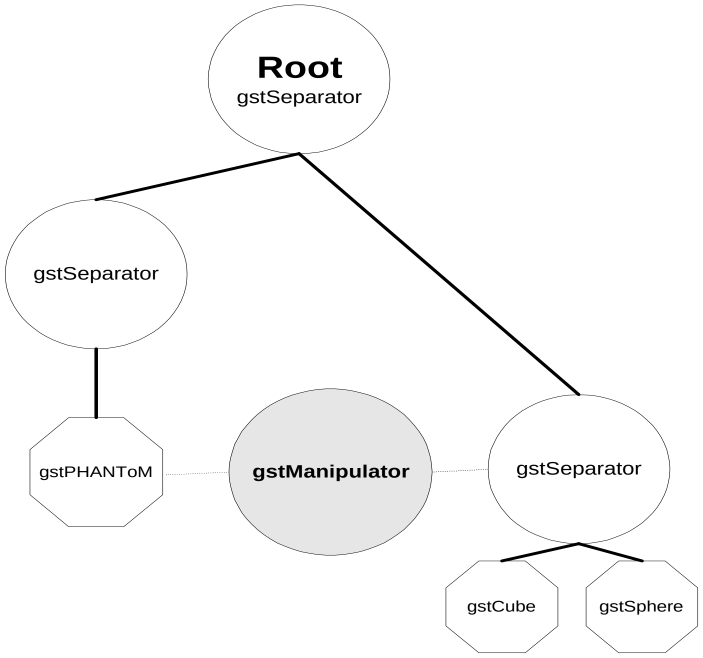


Figure 8: Manipulator in Scene Graph

The following code example shows how to define a new **gstTranslateManipulator** object.

```
        gstTranslateManipulator *translateManip = new
        gstTranslateManipulator;

        translateManipulator-
    >setNode(pointerToAnyNodeInSceneGraph);
        phantom->addManipulator(translateManip);
        phantom->startManipulator();
        .
        .
        .
        phantom->stopManipulator();
```

gstPHANToMDynamics

The **gstPHANToMDynamic** abstract class implements a special kind of **gstPHANToM** node that contains children; it is used specifically to transform its subtree in the scene graph and receive reaction forces from other **gstPHANToM** nodes touching geometry objects in that subtree. The **gstPHANToMDynamic** class attaches a geometric node to a **gstPHANToM** node that is then affected by the PHANTOM haptic interface device position and orientation.

This class differs from the **gstManipulator** class because forces from other **gstPHANToM** nodes touching the geometry nodes of its subtree are processed and added to the resulting forces on the **gstPHANToMDynamic**. In other words, when another **gstPHANToM** pushes against the subtree of a **gstPHANToMDynamic**, the **gstPHANToMDyamic** feels it. In addition, this GHOST class is not able to touch other geometry nodes in the scene. It may be helpful to consider the **gstPHANToMDynamic** as a solid volume, consisting of the subtree geometries, interacting with the infinitesimal points of the **gstPHANToM** nodes in the scene.

In scenes with two phantom devices, the **gstPHANToMDynamic** allows constructs where one hand controls the position and orientation of a target object through the **gstPHANToMDynamic** while the other hand controls another **gstPHANToM** that can touch that object. Physically, the user might hold and orient a can in one hand while painting on its surface with the other hand. This orientation is called the *two-handed* technique; it exploits the advantages of using the non-dominant hand for a reference frame (**gstPHANToMDynamic**) and the dominant hand for detailed work (**gstPHANToM**).

The derived **gstPHANToMDynamic** classes are **gstPHANToMTranslate** and **gstPHANToMRotate**. These classes use the associated PHANTOM haptic interface to translate and translate or orient its subtree using the translation or rotation of the PHANTOM haptic interface device respectively. As with the **gstPHANToMDynamic** class, **gstPHANToMTranslate** and **gstPHANToMRotate** append opposite reaction forces when the geometry nodes in the subtree is touched by another **gstPHANToM**.

gstPHANToMDynamic Constructor

The constructor for **gstPHANToMDynamic** functions similarly to that of **gstPHANToM**. Refer to the section **gstPHANToM** Constructor for more information.

gstErrorHandler

The **gstErrorHandler** implements an asynchronous mechanism for reporting and handling *GHOST SDK* errors. It is *not* always possible or convenient to return error codes by function return value. *GHOST SDK* errors, therefore, are passed into the **gstErrorHandler** function to be printed to the screen and optionally handled by a user defined error handler. By default, error messages are printed to **stderr** in IRIX and are displayed in a popup window under Windows NT. If defined, the user-defined handler is called and control is returned to the calling process. Error message printing can be turned on or off through the **printErrorMessages** function defined in **gstErrorHandler.h**. If error messages are turned off, **gstErrorHandler** relies solely on the user-defined handler to process errors. The user-defined handler is implemented as a callback function that should provide the following interface:

```
void (* newCallback)(int errorNumber, char *description, void
*userData)
```

Once set with **setErrorCallback**, the user-defined callback is executed for every error and is passed the error number, character description, and pointer to user data. Error numbers are defined in **gstErrorHandler.h**. See *Appendix C: GHOST SDK Error Codes* for more information about error numbers.

Graphic and Event Callbacks

In most cases, it is not feasible for the application to query all the nodes in the scene for state changes. Instead, it is more efficient and convenient for the application to be directly informed of state changes. Callbacks provide this mechanism, and the application simply calls the **gstScene** method **updateGraphics** or **updateEvents** to have event or graphics information updated through a user-defined callback. Each node in the *GHOST SDK* scene graph can contain an optional graphics callback function and an event callback function.

Callbacks pass new state information for the nodes in the scene from the *GHOST SDK* simulation to the application process. This state information provides a communication channel between the application and haptics processes so that the application can respond to changes in the haptics environment. Callbacks are most commonly used to synchronize access to state information of nodes in the scene graph between the haptics and application processes.

Graphics and events callbacks transfer different state information, each of which has separate requirements. In particular, graphic updates typically only need to keep track of the current state of the object, whereas event updates may need to additionally consider the object's history.

For example, an object traversing a path will take on many positions and orientations over time, but only the most current position and orientation is necessary for an application to maintain a real-time graphics representation. In contrast, other state information may actually need a complete history to maintain a correct context. For example, a **gstButton** node changes from a *released* to *pressed* state, then from *pressed* to *released* when the button is pressed and released. If the application is waiting for the **gstButton** to be pressed to perform a task, it is important that the application receives information that the button changed states from *released* to *pressed* and then back. If the current *released* state *only* is propagated to the application, the *pressed* state is missed, and the application never performs the task.

When the application calls **gstScene::updateGraphics**, nodes in the scene graph that have a graphics callback defined and have a new state since the last call to **gstScene::updateGraphics** will copy their current state to a defined callback data structure and call the graphics callback function with the copied state. The state copied into the data structure is specific to each node class, but it is always a superset of a parent class' specified graphics state.

The geometry class nodes and **gstSeparator** use the following graphic callback data structure:

```
typedef struct _gstTransformCBData {
      gstTransformMatrix transform; // local transform matrix
      gstTransformMatrix cumulativeTransform; // cumulative
                                    // transform matrix
} gstTransformGraphicsCBData;
```

The dynamic node classes use the following data structure:

```
typedef struct _gstDynamicCBData {
      gstTransformMatrix transform; // From
                              // gstTransformGraphicsCBData
      gstTransformMatrix cumulativeTransform; // also from
                              //gstTransformGraphicsCBData
      gstVector         reactionForce;          // Newtons
      gstVector         reactionTorque;         // Newton mm
      gstVector         velocity;               // mm/s
      gstVector         acceleration;           // mm/s^2
      gstVector         angularVelocity;        // rad/s
      gstVector         angularAcceleration;    // rad/s^2
} gstDynamicGraphicsCBData;
```

NOTE: The reference frame in which they are defined determines the last six fields of gstDynamicGraphicsCBData. Refer to the description of **Dynamic State** of *Dynamic Haptic Environments* earlier in this chapter to see how these fields are interpreted by the derived classes of **gstDynamic**.

Similarly, when the application calls **gstScene::updateEvents**, any nodes in the scene graph that have an event callback defined call their respective event callback function once (with the event data structure) for each event that has occurred since the last call to **gstScene::updateEvents**. Unlike the graphics callback data structure, events all use the same data structure. Each class, however, can interpret the fields differently; specifically, the event fields have different meanings for each dynamic. The following is a list of *GHOST SDK* classes that have defined events; it shows their representations:

```
typedef struct _gstEvent {
      int               id;
      int               data1d;
      int               data2d;
      int               data3d;
      double            data1f;
      double            data2f;
      double            data3f;
      void              *data1v;
} gstEvent;
```

```
gstButton:
      event.id =  BUTTONSTATE // gstButton::RELEASED or
                              // gstButton::PRESSED

gstDial:
      event.id = NOTCH              // int from 0-numberNotches
                                    // Default is 6 numbered
                                    // clockwise from 12oclock
      event.data1d =    DIRECTION   // gstDial::CLOCKWISE or
gstSlider:
      event.id =  NOTCH             // int from 0-numberNotches
                                    // (default is 4)
      event.data1d =    DIRECTION   // gstSlider::GST_UP or
                                    // gstSlider::GST_DOWN
gstRigidBody:
      none

gstShape:
      event.id = CONTACT_STATE      // gstShape::TOUCHED or
                                    // gstShape::UNTOUCHED
```

You *must* define the graphic and event callback, and both functions expect the following interface:

```
void ( *start_address )( gstTransform *nodePtr, void
*callbackData, void *userData )
```

NOTE: The **nodePtr** points to the node instance that this callback refers to. The **CallbackData** points to the graphic or event callback data structure defined for the node class this callback refers to. The callback function must cast **callbackData** to the correct data type to access the fields. The **userData** is a void pointer that contains specific user data passed in when the callback was defined. For example, one callback can be defined for many objects and **userData** used to specify the memory space in which to put the new state for that specific object.

Once the callback is defined, a pointer to itself and **userData** (if desired) can be passed into the **gstTransform::setGraphicsCallback** or **gstTransform::setEventCallback** methods, respectively, for each instance of a *GHOST SDK* node. Only those nodes that have a callback set through this method will update graphics or events when needed. Look to the example code sent with the *GHOST SDK* distribution for examples of using the event and graphics callbacks.

PHANTOM Surface Contact Point and the gstPHANToM_SCP node

When a **gstPHANToM** intersects a geometry node, the **gstPHANToM** establishes contact with that geometry node, and the node *must* calculate a point on its surface that signifies where it is being touched. This reference point is the SCP; it differs from the actual position of the **gstPHANToM** node as shown in Figure 9.

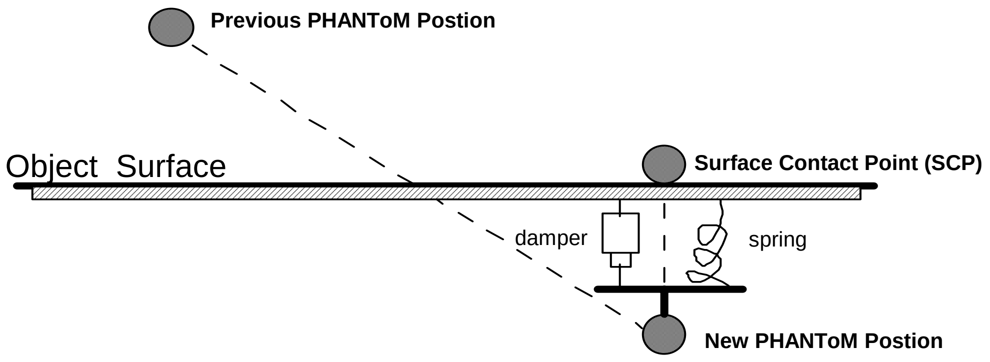


Figure 9: Surface Contact Point Figure

Two options exist for graphically representing the phantom position. The first and simpler option is to always use the actual **gstPHANToM** position. The second is to use the projection of the phantom position on an object it is touching (the SCP), if such a phantom position exists, or to return the actual position of the phantom device. This latter approach, although more complicated, is for most purposes the desirable graphic representation of the phantom position. Otherwise, using the real **gstPHANToM** position can cause the graphical phantom to appear to sink into the surface when significant force in applied by the user in that direction.

The **gstPHANToM_SCP** node enables the SCP to be used to graphically represent the phantom position. To implement the **gstPHANToM_SCP**, the developer should create an instance of it and the usual **gstPHANToM** node, but the graphics callback routine normally used to update the graphical position of the phantom should now instead be attached to the **gstPHANToM_SCP**.

The **gstPHANToM_SCP** node should be added to the scene graph directly under the same parent as the **gstPHANToM** node. It can be placed anywhere in the scene graph, just as any other node, but the location only dictates the local reference frame of the position and orientation. The world-reference frame position and orientation are *not* affected by the location of the node in the scene graph.

To enable the position and orientation of the **gstPHANToM_SCP** to reflect that of the SCP for a **gstPHANToM**, the **gstPHANToM** must receive a pointer to the **gstPHANToM_SCP** through the **gstPHANToM::setSCPNode** method. Afterward, the **gstPHANToM** will continually update the position of the **gstPHANToM_SCP** during each servo loop. The developer only needs to create a graphics callback for the **gstPHANToM_SCP** node to represent the graphics position of the *PHANTOM* haptic interface device.

Enabling and Disabling Force Output to PHANTOM devices.

The *GHOST SDK* provides the following three ways to specialize force output to the PHANTOM haptic devices associated with **gstPHANToM** and **gstPHANToMDynamic** nodes:

- **gstPHANToM::setForceOutput** and **gstPHANToMDynamic::setForceOutput** specifies for each PHANTOM device node whether or not any forces are sent to the associated PHANTOM haptic device during the simulation. The default value is **TRUE**. This method *should be* called before the servo loop is started.

- **gstTransform::setTouchableByPHANToM** specifies that a node and the subtree beneath it are touchable by any **gstPHANToM** or **gstPHANToMDynamic** node. If **TRUE**, then all **gstPHANToM** and **gstPHANToMDynamic** nodes are able to contact the node and its subtree. Otherwise, the node and its subtree are transparent to all **gstPHANToM** and **gstPHANToMDynamic** nodes. The default value for each node is **TRUE**. This function can be called at any time.

- **gstScene::setQuitOnDevFault** specifies if a device fault from any PHANTOM device will cause the servo loop to exit abnormally. If **TRUE**, any one or more device faults from a PHANTOM device will cause the servo loop to exit abnormally. Otherwise, only device fault on startup will cause the servo loop to exit abnormally. If set to **FALSE**, the kill switch may be used to stop forces from being sent to the PHANTOM device while leaving the simulation running. Turning the kill switch on again will restart force generation.

Detecting Contact with Nodes

The **gstShape** node will generate an event when going from untouched to touched and vice versa when an event callback is defined by the user. Details on the format of **gstEvent** and its use are explained in the *Graphic and Event Callbacks* section earlier in this chapter.

Generic Geometry and the gstTriPolyMeshHaptic Class

The GHOST scene graph node supports triangular polygons with the **gstTriPolyMeshHaptic** class. This class provides the haptic interaction and scene graph support for triangular polygons (hereafter referred to as triPolys). The actual database storing the triPoly mesh is the **gstTriPolyMesh** class. The **gstTriPolyMeshHaptic** class stores a pointer to the relevant **gstTriPolyMesh** database and together they support haptic triPolys. The construction of the haptic scene graph object and the triPoly database object will be described separately.

Creating the Triangle Database (**gstTriPolyMesh**)

First, creating the database of triPolys requires creating an instance of **gstTriPolyMesh**. This object has a default, copy, and indexed array constructor. The default constructor initializes the triPoly database with no triangles to start. The copy constructor is straightforward, and the indexed array constructor is of the same form as the original **gstPolyMesh** constructor that takes an indexed array of vertices and triangles defined by vertex indices:

```
// Constructor.
 gstPolyMesh(int numVertices,
        double vertices[][3],
        int numTrianglePolygons,
        int trianglePolygons[][3],
        gstBoolean checkForDuplicateVertices = TRUE);
```

In addition, the following constructor is also provided :

```
gstTriPolyMesh(
 int numVertices,             // the number of vertices
 int v_dimension,             // dimension of vertices == 3
 double *vertices,            // the vertices
 int numTrianglePolygons,     // the number of triangles
 int num_sides,            // number of sides of polys == 3
 int *trianglePolygons,       // the triangles
 gstBoolean useSpatialPartition = TRUE);
```

Here `vertices`[0], `vertices`[1] and `vertices`[2] are the co-ordinates of the first vertex; `vertices`[3], `vertices`[4] and `vertices`[5] are the co-ordinates of the second vertex and so on. Similarly, `trianglePolygons`[0] thorugh `trianglePolygons`[2] give the vertex indices of the first triangle; `trianglePolygons`[3] through `trianglePolygons`[5] give the vertex indices of the second triangle and so on. The parameters `v_dimension` and `num_sides` must both be set to 3.

After initializing the **gstTriPolyMesh** object it is then possible to add/remove vertices/polygons at any time through the createTriPoly, createVertex, removeVertex, and removePolygon methods. Each of these methods including the vertex and polygon accessors (getPolygon, getVertex) can access the vertex or polygon using its index or actual pointer. If the default constructor is used, then it is necessary to use the createVertex and createTriPoly methods to add triangles to the database. Before a triPoly can be created, the vertices used by it need to be created first using createVertex.

After triangles have been added to the **gstTriPolyMesh** it is necessary to call initSpatialPartition in order to add the latest triangles to the spatial partition if one is being use (the spatial partition is used by default).

Note: The spatial partition is initialized when using the indexed array constructor shown above.

The triangles of the database should always be created directly through createTriPoly. Thus the user should never directly create a **gstTriPoly** object. The gstTriPoly objects are created through **gstTriPolyMesh::createTriPoly** and can be retrieved through **gstTriPolyMesh::getPolygon**. The **gstTriPoly** represents a triangle and allows access to the **gstVertex** and **gstEdge** instances that make up the **gstTriPoly**. Access to these parts is allowed through methods of **gstTriPoly**, **gstEdge**, and **gstVertex**. For example, each **gstVertex** created by **gstTriPolyMesh::createVertex** can return the edge incident upon them through the **gstVertex::incidentEdgesBegin** and **gstVertex::incidentEdgesEnd**. The return values of these methods are STL iterators defined as **gstIncidentEdgeListConstIterator**. Notice that these functions return iterators that point to **gstIncidentEdges**.

gstIncidentEdge is a utility class that points to a **gstEdge** and also stores a direction. Therefore it is possible to detect if the incident edge points into or out of the **gstVertex**. For information on using STL and iterators please refer to:

http://www.sgi.com/Technology/STL

or any STL reference manual.

Along with adding/removing triangles interactively, **gstTriPolyMesh** supports modification of vertex location after triangles have been defined. In order to keep the spatial partition of the mesh consistent and efficient it is ABSOLUTELY necessary for the user to use **gstTriPolyMesh::beingModify**, **gstTriPolyMesh::endModify**, and **gstTriPolyMesh::endModifications**. The first two methods take the gstTriPoly Modified as its only parameter. This notifies the mesh's spatial partition when the triangles are invalid and valid again. The method endModification should be called after all modifications are done and the spatial partition should be optimized. Calling beginModify and endModify does NOT optimize the spatial partition.

Destruction of **gstTriPoly**, **gstEdge**, and **gstVertex** is handled by the **gstTriPolyMesh** class and should never be done explicitly on the objects themselves. **gstTriPolyMesh** has removePolygon and removeVertex to support cleaning up the database. If vertices are to be kept around even if they become stranded by removePolygon then the **gstTriPolyMesh::setDeleteStrandedVertices** method.

All of the classes just described have an arsenal of methods that allow very robust traversal and retrieval of information and the user should refer to the header files for now to locate these methods.

Defining Surface Normals for the Triangle Database

Each **gstTriPoly** now contains a reference to a special class called a **gstPolyPropertyContainer**, that can be accessed via the **gstTriPoly::getPropertyContainer** function. This new class contains various "polygon properties" or characteristics of the polygon's surface that are then interpreted haptically. For instance, to make a polygonal surface feel smooth at the edges between two adjacent polygons, which meet at a nonzero angle, it is necessary to define the surface normal for the two polygons. The *GHOST SDK* will then interpolate between these normals to create an appropriately smooth feel.

Each **gstTriPoly** in the mesh can have its normals defined in one of two ways. One can define a separate normal vector at each corner, with the setPropertyV1, setPropertyV2, and setPropertyV3 functions, which results in a surface normal interpolated between these three. Or one can define one normal vector for the entire polygon with the setPropertyPoly function.

For example, given a pointer to a **gstTriPoly** , *tp, and three **gstPoint** normal vectors n1, n2 and n3 defined at its vertices, one would apply these normals as follows:

```
      gstPolyPropertyContainer &ppc = tp->getPropertyContainer();
      ppc.setPropertyV1(new gstNormalPolyProperty(n1));
      ppc.setPropertyV2(new gstNormalPolyProperty(n2));
      ppc.setPropertyV3(new gstNormalPolyProperty(n3));
```

If, instead, one wanted to apply just one **gstPoint** normal vector n to the entire polygon:

```
      gstPolyPropertyContainer &ppc = tp->getPropertyContainer();
      ppc.setPropertyPoly(new gstNormalPolyProperty(n));
```

These **gstPolyPropertyContainers** can be defined at any time after the creation of the gstTriPoly.

Making the Triangle Database Touchable

Once the triangle database is created, it can be added to a **gstTriPolyMeshHaptic** object to become touchable by the PHANTOM. The **gstTriPolyMeshHaptic** constructor takes a pointer to a **gstTriPolyMesh** object that represents the triangles to touch. Once created this way, the

gstTriPolyMeshHaptic object can be placed in the scene like any other **gstShape** object and touched by starting the servoloop with a **gstPHANToM** in the scene.

The method **gstTriPolyMeshHaptic::getTouchedPolys** returns the current touched polygons. The return value is a vector of pointers to **gstTriPoly** objects. This vector can be 0-3 in length with zero meaning no contact and 3 objects indicating contact at a corner: 1 or 2 means the mesh is contacted at a face or edge respectively. The mesh can be touchable from the front, back, or both sides of each polygon. By default meshes are only touchable from the front of each triangle. This can be altered through the **gstTriPolyMeshHaptic::setTouchableFrom** method.

NOTE: The **gstTriPolyMeshHaptic::scale**, **setScale**, **center** and **setCenter** methods are not supported at this time.

Force Fields

The force field class allows the developer to send forces to the PHANTOM when the PHANTOM has entered the force field's bounding volume. The developer can specify the actual force sent based on the data passed in via the **gstPHANToM** instance. This class provides tremendous flexibility to the developer to define their own force characteristics of the scene.

The developer subclasses the **gstForceField** class and overloads the method:

```
gstVector calculateForceFieldForce(gstPHANToM *phantom)
```

The returned vector is expected to be in the local coordinate system of the force field.

The implementation of the gstForceField also includes the integration of force from gstForceField instances with forces resulting from touching objects in the scene. Without this implementation, side effects such as buzzing and unstable forces would result. In addition, the force field implementation attenuates the forces as you move out of the force field. This allows the developer to specify a force field that is in the direction of the normal of the bounding volume and not get a buzzing condition on the boundary. The buzzing is the result of the force field pushing the PHANTOM out of the force field and the user pushing it back in. The developer can specify the attenuation distance that this is achieved over through the call to:

```
void setAttenuationDistance(const double dist);
```

A subclass of the **gstForceField** called **gstConstantForceField** is implemented. This class allows the developer to specify a single force vector that is applied across the whole bounding volume. The force vector can be specified as part of the constructor or assigned later through an accessor call.

```
gstConstantForceField(gstVector &force);
void setConstantForceVector(const gstVector &force);
```

This class provides significant flexibility for application specific force-displacement models that are integrated with forces generated from objects in the scene. It is implemented as a direct result of the GHOST forum during the 1997 PHANTOM Users Group.

VRML 2.0 and the GHOST SDK

The GHOST SDK is capable of reading polygonal meshes from a VRML 2.0 file and adding them into a GHOST SDK scene. The following function is used to read in a VRML 2.0 file:

```
gstSeparator *gstReadVRMLFile(const char *filename)
```

This function returns a separator node which can then be added to the GHOST SDK scene graph. Any nodes of type IndexedFaceSet in the VRML 2.0 file are converted to GHOST nodes of type **gstTriPolyMeshHaptic.** The **gstSpatialPartition** for the meshes is automatically initialized.

When reading a VRML 2.0 GHOST SDK maintains a list of all of the errors encountered while reading the file. After a call to gstReadVRMLFile this list is populated with gstVRMLError objects which can be accessed with the following functions:

`int gstVRMLGetNumErrors() -` returns the total number of errors encountered while reading the file.

`gstVRMLError gstVRMLGetError() -` returns the first error encountered.

`gstVRMLError gstVRMLPopEarliestError() -` returns the first error encountered and removes it from the list.

`int gstVRMLWriteErrorsToFile(const char *filename)` – writes all errors to a file.

`const char *gstVRMLGetErrorTypeName(gstVRMLErrorType errType)` – converts an error code to a string.

The gstVRMLError object contains the error code, line number and error message for the error. These can be acessed with the methods `GetError()`,`GetLine()`, and `GetMSG()` respectively.

Here is an example of a function that uses the above routines to read in a VRML 2.0 and print out all errors to stdout.

```
gstSeparator *my_fcn_to_get_sep_from_file(const char* file)
{
    gstSeparator *vrmlSep = gstReadVRMLFile(file);
    while (gstVRMLGetNumErrors() > 0) {
        gstVRMLError err = gstVRMLPopEarliestError();
        cout << "Error in VRML file ";
        cout << gstVRMLGetErrorTypeName(err.GetError()) << " ";
        cout << err.GetMSG() << " ";
        cout << "on line " << err.GetLine() << endl;
    }
    return vrmlSep;
}
```

NOTE: GHOST SDK reads VRML 2.0 files. VRML 1.0 files are not supported although third party VRML 1.0 to VRML 2.0 converters do exist.

The following example program uses the GHOST SDK VRML 2.0 reader to read in a VRML file and display it haptically. It uses the GHOSTGL described in the next chapter to add graphic display of the VRML object.

```
#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstVRML.h>
#include <ghostGLUTManager.h>

void main(int argc, char *argv[])
{
    gstScene myScene;

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

    // read in the VRML file
    // this creates a separator which then gets added to scene
    gstSeparator *vrmlSep = gstReadVRMLFile("foo.wrl");
    rootSep->addChild(vrmlSep);

    // print out all the errors generated reading the file
    while (gstVRMLGetNumErrors() > 0) {
        gstVRMLError err = gstVRMLPopEarliestError();
        cout << "Error in VRML file ";
        cout << gstVRMLGetErrorTypeName(err.GetError()) << " ";
        cout << err.GetMSG() << " ";
        cout << "on line " << err.GetLine() << endl;
    }

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

    // create a PHANTOM instance and check to make sure it is valid
    gstPHANToM *phantom = new gstPHANToM("Default PHANTOM");
    if (!phantom || !phantom->getValidConstruction()) {
        cerr << "Failure to create a valid Phantom construction." << endl;
        exit(-1);
    }

    // add the PHANTOM object to the scene
    rootSep->addChild(phantom);

    // start GHOST SDK servo loop
    myScene.startServoLoop();

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

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

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

}
```

Note that this program assumes that the VRML model being read in will fit correctly into the workspace of the PHANTOM device. Since the scale and position of VRML models varies considerably you may have to adjust the transformation of the vrmlSep node to correctly fit the model into the workspace.

The gstDeviceIO class and low-level PHANTOM access

A new addition to the GHOST SDK is the gstDeviceIO class. This class allows developers to directly access the encoders and motors of the PHANTOM, and will allow users to define a servo loop that need not run at 1000 Hz. This class is not meant to interact with any of the other GHOST classes (with the exception of the various data classes).

The gstDeviceIO class is a powerful new tool in the GHOST developer's arsenal. The developer can now filter encoder values, directly send forces to the motors, and test the motor temperatures, among numerous other uses.

All of the functions in this new class are fairly self-explanatory. The functions can be used in a GHOST servo loop, started by `gstStartServoScheduling(gstServoSchedulerCallback pCallback, void *userData)`, which starts the typical 1000 Hz servo loop.

If the developer would rather use a non-1000 Hz servo loop rate, the `gstStartServoScheduling()` function is not used. Instead, the developer can design a loop that performs the specific tasks that are desired, and then calls `gstUpdatePhantom(int id)`, to update the internal state of the PHANTOM.

The HelloSphere application (in the examples directory of your GHOST SDK installation) is a great example of the usage of the various gstDeviceIO functions.

NOTE: The gstDeviceIO class is intended to be used by expert GHOST developers.

Chapter 4: Adding graphics with GHOSTGL

GhostGL is a library that can render any GHOST SDK scene using OpenGL. It provides a quick and easy way to add graphics to any GHOST SDK application. Once a GHOST SDK scene graph has been created it can be passed to the GHOSTGL routines that traverse and render a graphical representation of each node in the scene. Full source code is provided for the GHOSTGL library.

Using GHOST GL with GLUT

GLUT is a freely available cross-platform library that acts as the glue between an OpenGL program and the native windowing system. A program written to use GLUT can be compiled without modification on Windows NT or Irix. GHOST GL can be used with GLUT to add OpenGL rendering to any GHOST SDK program.

To use GHOSTGL with GLUT simply add four lines to your program:

- Include the ghostGLUTManager header file.

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

- Create an instance of ghostGLUTManager.

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

  The arguments to this method are the command line arguments passed in to your main function and the title of the window that the graphics will be rendered in.

- Load your gstScene object into the ghostGLUTManager.

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

- Start the ghostGLUTManager main graphics loop.

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

  This gives control of the program to the ghostGLUTManager, which will automatically redraw the scene approximately 30 times a second. As objects in the GHOST SDK scene move around the corresponding graphical objects will be redrawn in their new locations.

The following program renders a sphere using GHOST, GHOSTGL and GLUT:

```
#include <gstScene.h>
#include <gstPHANToM.h>
#include <gstSphere.h>
#include <iostream.h>
#include <ghostGLUTManager.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();

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

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

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

}
```

Programs that use the ghostGLUTManager must be linked with the ghostGLUTManager library, the GLUT library and the OpenGL libraries (gl and glu) in addition to the GHOST SDK library.

Using GHOSTGL with other Window Frameworks

The GHOSTGL library can be used independently of GLUT. It can be used with MFC, Motif or any other windowing framework that is compatible with OpenGL. After creating the GHOST SDK scene you create an instance of ghostGLManager:

```
ghostGLManager * glManager = new ghostGLManager();
```

The ghostGLManager is the base class of the ghostGLUTManager so it shares many of the same methods. You use the same routine to load the scene into the ghostGLManager:

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

The ghostGLManager must be informed about changes in the dimensions of the viewport by calling

```
glManager->reshape(width, height);
```

whenever the window size changes and when the ghostGLManager is first created. Most frameworks provide a callback function that is called whenever the window size changes, so the call to reshape can be placed there.

The ghostGLManager should be told to redraw the scene at each iteration of the graphics loop by calling

```
glManager->redraw();
```

In most window frameworks this routine is scheduled to be called 30 times a second using a timer.

Programs that use GHOSTGL without GLUT must link with the ghostGLManager library.

The RobotFramework program in the examples directory is a simple example of using GHOSTGL with MFC or Motif.

Customizing GHOSTGL Rendering

By default all of the primitives in the scene are drawn using the same color, lighting, etc… GHOSTGL provides a way to customize the graphic rendering of each node in the scene in order to use different colors and textures for different nodes.

Each node in the scene has a set of OpenGL display lists associated with it. These lists are stored in a class called gfxDisplaySettings. The display settings for any node in the scene can be retrieved from the glManager:

```
gfxDisplaySettings *settings = glManager->getDisplaySettings(myNode);
```

The gfxDisplaySettings class contains a pre display list, which is called before the node is rendered and a post display list, which is called after the node is rendered. You can customize the pre display list to set colors or textures to be used for drawing the node and then set the post display list to clean up the changes made by the pre display list. The following example uses this technique to draw a node in red:

```
// Get ghostGLManager display settings for the node
gfxDisplaySettings *settings = glManager>getDisplaySettings(mySphere);

// 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);
       GLfloat headColor[] = {1.0f, 0.0f, 0.0f, 1.0f};
       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();
```

Note that if the post display list were not set to restore the old color, the red color would be used to draw all of the nodes after drawing mySphere.

Any settings done in the pre display list of a separator node will affect the rendering of all of the children of the separator.

The GHOSTGL Camera

By default GHOSTGL sets the camera at a fixed location (a fixed distance along the z axis). GHOSTGL allows you to move the camera and also to set the camera so that it automatically synchronizes with the PHANTOM workspace.

The ghostGLManager can be queried for a pointer to the camera:

```
ghostGLCameraBase *myCamera = glManager->getCamera();
```

The pointer to the camera can be used to set the camera position, orientation and the front and back clipping planes. For details, see the file ghostGLCameraBase.h in the include directory of the GHOST SDK distribution.

GHOSTGL also provides a specialized camera called ghostGLSyncCamera that will automatically synchronize with the PHANTOM workspace. To add one of these cameras to the scene, simply create one and pass it in to the constructor of the glManager.

```
ghostGLSyncCamera *myCamera = new ghostGLSyncCamera();
ghostGLManager *glManager = new ghostGLManager(myCamera);
```

There are two different modes for camera synchronization: synchronize camera to workspace and synchronize workspace to camera. The mode is set with the method

```
myCamera->setSyncMode(mode);
```

where mode is ghostGLSyncCamera::SYNC_CAMERA_TO_WORKSPACE or ghostGLSyncCamera::SYNC_WORKSPACE_TO_CAMERA.

In synchronize workspace to camera mode, the parent node of the gstPHANToM node in the scene graph is automatically transformed by GHOSTGL to follow the camera. As the camera moves, the reference frame of the PHANTOM device also changes to follow it. This means that as the graphic view of the scene changes, the haptic view or the area of the scene that is reachable with the PHANTOM device follows it. This guarantees that what you see is what you touch – the graphical and haptic orientation, position and scale are synchronized.

In synchronize camera to workspace mode the same synchronization is done but instead of the PHANTOM workspace following the camera, the camera follows the PHANTOM workspace. When the parent of the gstPHANToM node is transformed, then the camera is also transformed to follow it. In order for this to work the gstPHANToM must have a boundary cube attached to it, and the camera must be informed about it. The boundary cube is a touchable box around the workspace of the PHANTOM device. The simplest way to create one is by calling gstPHANToM::attachMaximalBoundary.

```
gstBoundaryCube *myBoundary = myPhantom->attachMaximalBoundary();
myCamera->setWorkspaceBoundary(myBoundary);
```

As the reference frame of the PHANTOM device is changed (by transforming the parent of the gstPHANToM node) the transform of the boundary cube will also change and GHOSTGL will transform the camera to always look at the boundary cube from the correct orientation.

Calling ghostGLSyncCamera::setSyncEnabled turns on and off camera/workspace synchronization.

Note that for the ghostGLSyncCamera to work in either mode, the gstPHANToM node must have a parent separator node distinct from the rest of the geometry in the scene. In particular, the parent of gstPHANToM node can not be the root node of the scene. This parent node is the node that is transformed to follow the camera or the node that the camera is transformed to follow.

The Pinch Transform

GHOSTGL provides a method for the user to adjust the view using the PHANTOM device. The user holds down the stylus switch and moves the stylus around to interactively position and orient the camera. This mode is best combined with a ghostGLSynchCamera in synchronize workspace to camera mode which will automatically orient the workspace to follow the camera changes. To add the pinch transform behavior to your application, simply create an instance of ghostGLPinchXForm and add it to the ghostGLManager.

```
ghostGLPinchXForm *pinchXFormObj = new ghostGLPinchXForm();
glManager->addActionObject(pinchXFormObj);
```

For an example of the pinch transform, see the PhantomContext program in the examples directory of GHOST SDK distribution.

The pinch transform is derived from ghostGLActionObject. The action object is a simple way to add your own processing into the GHOSTGL graphics loop. The ghostGLActionObject has two important methods: preDisplay which is called before rendering each frame, and postDisplay which is called after rendering each frame. By deriving your own class from ghostGLActionObject and overriding these methods you can add your own processing to the graphics loop.

Chapter 5: The HapticView Framework

All of the demos distributed with GHOST SDK make use of a platform independent interface called HapticView. This interface provides hook functions for managing a GHOST application within a Windows/MFC or X Windows/Motif environment. Note: the HapticView framework was designed to be used with OpenGL and will not work with other graphics libraries unless modified to do so. Full source code is provided for the HapticView framework.

In order to use the HapticView framework, you ***must*** derive from the HapticView class and override the following interface functions.

```
virtual void    StartProgram(BOOL bPHANToMMouseEnabled);
virtual void    EndProgram();
virtual BOOL    ProgramDone();
virtual void    InitGraphics();
virtual void    UpdateGraphics();
virtual void    ResizeGraphics(int cx, int cy);
virtual void    TermGraphics();
virtual void    EnableServoLoop(BOOL bEnable);
virtual LPCSTR* QueryPHANToMNames();
virtual BOOL    QueryCursorPos(double* pX, double* pY, double* pZ);
```

HapticView Functional Overview

Before describing the job of each individual function, is it worthwhile to understand how they all fit into the execution flow of a GHOST application. In a typical GHOST application, the view becomes the central graphical interface for the user. It is within the view that you perform the graphical rendering of the scene, and therefore most of the fundamental framework hooks focus on the creation and maintenance of the view.

When the view is first created and the rendering context is established, InitGraphics will get called. This routine should be used for initializing OpenGL state, like depth buffering, culling, lighting, etc. It could also potentially be used for performing additional tasks, like creating display lists. While the view is getting created, the framework needs to know whether any of the PHANTOM devices being used by the application are going to require a reset (i.e. be placed in reset position). QueryPHANToMNames() should return an array of PHANTOM configuration names, like “Default PHANTOM”. Before the view creation has completed, you will receive at least one call to ResizeGraphics. This function will give you the new dimensions of the view. Given these dimensions, you must at least setup the OpenGL viewport, or else you won’t see anything in the view.

Following the creation of the view, the next hook to get called is StartProgram. If you’re using a PHANTOM Desktop model, then StartProgram will get called automatically. Otherwise, the view will wait for the PHANTOM device to be placed in its reset position while displaying a reset bitmap to the user. Upon hitting <Enter> or clicking the left mouse button in the view, the StartProgram hook will get called. This is your main entry point into the program initialization. You should be creating your scene graph as well as whatever graphics data structures you’ll be needing. When you finish, the HapticView will call the EnableServoLoop() hook and then begin periodically calling UpdateGraphics() to trigger a redraw of the OpenGL context.

During program execution, the ProgramDone() routine will be called periodically to ensure that the servoloop is still running. If this routine returns TRUE, then the application will cleanup and exit.

Finally, when the program is exiting, it will call both the TermGraphics() and EndProgram() hooks. You should use these routines for freeing up whatever graphics or haptics resources were acquired during the life of the application.

HapticView Interface Functions

Function Prototype:

```
virtual void StartProgram(BOOL bPHANToMMouseEnabled);
```

Function Description:

The main entry point into the haptic and graphic initialization of the program. Primarily used for creating the gstScene and gstPHANToM instances as well as populating the scene graph with its initial GHOST primitives. The bPHANToMMouseEnabled argument will let you know whether the PHANTOM needs to be reset. If the PHANTOM Mouse is enabled, it is not necessary to reset the PHANTOM. Otherwise, you should set the reset argument in the gstPHANToM constructor to TRUE. For example:

```
BOOL bResetPHANToM = !bPHANToMMouseEnabled;
gstPHANToM *myPHANToM = new gstPHANToM("Default PHANTOM", bResetPHANToM);
```

Function Prototype:

```
virtual void EndProgram();
```

Function Description:

Cleanup the haptics related resources. Generally, this just involves deleting the gstScene instance, which will in turn delete all children under the root separator.

Function Prototype:

```
virtual BOOL ProgramDone();
```

Function Description:

This function gets called periodically to check whether the servoloop is still running. If the routine returns TRUE at any time, then the application will proceed to quit. Usually, you'll want to have some additional logic in your program to keep track of whether the servoloop has been stopped via a call to EnableServoLoop(FALSE). In these cases, you'll want to keep the application going, even though the servoloop isn't running. For instance, a typical ProgramDone could be:

```
BOOL HapticViewExample::ProgramDone()
{
    return (!m_bSuspended && m_pScene && m_pScene->getDoneServoLoop());
}
```

Function Prototype:

```
virtual void InitGraphics();
```

Function Description:

InitGraphics gets called following the creation of the view and the OpenGL rendering context. Use this call to setup rendering state like depth buffering, culling, lighting, etc.

Function Prototype:

```
virtual void UpdateGraphics();
```

Function Description:

HapticView will call this routine to initiate a redraw of the view at a rate of approximately 30 Hz. This is where you handle the graphics thread processing, like calling updateGraphics and updateEvents on the scene graph and drawing the scene. HapticView will handle clearing the OpenGL color and depth buffers before calling UpdateGraphics() as well as swapping the offscreen buffer when the routine returns.

Function Prototype:

```
virtual void ResizeGraphics(int cx, int cy);
```

Function Description:

ResizeGraphics will get called whenever the view incurs a resize. The two parameters received in this routine are the new view dimensions along the x and y axes respectively. At the very least, this routine should setup the OpenGL viewport, or else nothing will be displayed.

```
glViewport(0, 0, cx, cy); // sets the viewport origin and dimensions
```

Function Prototype:

```
virtual void TermGraphics();
```

Function Description:

Cleanup whatever resources you used for rendering the graphics, e.g. display lists, textures, etc.

Function Prototype:

```
virtual void EnableServoLoop(BOOL bEnable);
```

Function Description:

HapticView will call this method to start or stop the GHOST servoloop. You must override this function to call gstScene::startServoLoop() or gstScene::stopServoLoop() on your gstScene instance. The value of bEnable will tell you whether the servoloop needs to be enabled or disabled.

Function Prototype:

```
virtual LPCSTR* QueryPHANToMNames();
```

Function Description:

During initialization of the view, HapticView will need to know if any of the PHANTOMs being used will require a reset. If the PHANTOMs don't require a reset, then StartProgram can get called immediately after creating the view. Otherwise, the view displays a bitmap that instructs the user to place the PHANTOM in its reset position. The QueryPHANToMNames function gets used in this context to obtain an array of PHANTOM configuration filenames to be used. HapticView has placed a limit of 2 on the number of PHANTOM names although GHOST can actually handle up to 4.

```
LPCSTR *MyHapticView::QueryPHANToMNames()
{
    static LPCSTR apszNames[] = {
                    "Default PHANTOM",
                        NULL
                    };
    return apszNames;
}
```

Function Prototype:

```
virtual BOOL QueryCursorPos(double* pX, double* pY, double* pZ);
```

Function Description:

This is a special function needed by the HapticMouse to get the current PHANTOM position in world coordinates. This position is used when the PHANTOM is transitioning from 3D space to the 2D mouse plane. HapticView will project the 3D position returned onto the viewport. This routine will get called during an OnPhantomLeave() event.

PHANTOM Mouse Integration

In addition to providing a foundation for a standard GHOST application, HapticView also provides support for interfacing with the PHANTOM Mouse. The PHANTOM Mouse is a mode of operation in which the PHANTOM device emulates the abilities of a 3-button mouse. When you transition into mouse mode, you can actually use the PHANTOM device in place of your mouse, allowing you to interact with windows, menus, icons, etc. The basic PHANTOM mouse functionality is built-in to the HapticView framework, so your application can readily make use of it without any additional programming. However, there are a series of default attributes about the mouse that can be modified using the functions described herein. In particular, there is support for several 2D-to-3D and 3D-to-2D transitioning mechanisms, 3-button mouse emulation, and an interface for dynamically configuring the mouse plane.

3D to 2D Transitioning Mechanisms:

The list below describes two mechanisms by which you can trigger a transition from the 3D scene into 2D mouse mode.

- Initiate a transition by pressing the <Esc> key while the view is active. This is the default behavior of any HapticView application.
- Perform an event based transition by calling OnPhantomLeave(). For instance, the PhantomContext example application uses this mechanism to force a pop-thru when applying force against a haptic frustum. This offers an intuitive way of pushing against the edge of the view to transition out of the 3D world.

2D to 3D Transitioning Mechanisms:

This list below describes three mechanisms by which you can trigger a transition from 2D mouse mode back into the 3D scene.

- **Fall thru** : Allows you to fall back into the 3D scene only after the mouse cursor has left the view and then re-entered it. This is the default behavior of any HapticView application.
- **Click thru** : Causes a transition back into the 3D scene only if you click the stylus button or left mouse button while the cursor is in the view. *Note: the stylus button must be emulating a left mouse click in order for the stylus click to work.*
- **Push thru** : Simulates a mouse plane that behaves like thin ice. This allows you to cause a transition by pushing against the plane while the cursor is in the view.
- Perform an event based transition by calling OnPhantomEntry()

As mentioned above, the default configuration for the PHANTOM Mouse uses the <Esc> and fall thru transitioning mechanisms. In addition, the stylus button emulates a left mouse button click. In order to customize this configuration, you must use the following interface functions. *Note: OnPhantomEntry() and OnPhantomLeave() are two functions that you can override to perform some additional tasks when a transition occurs. The other functions described below are to be used in configuring the HapticMouse.*

Function Prototype:

```
virtual void OnPhantomEntry();
```

Function Description:

HapticMouse calls this hook when a 2D to 3D transition has occurred. You can force a PHANTOM mouse entry transition by calling this function yourself. In general, you should allow HapticMouse to handle detecting the appropriate entry event.

Function Prototype:

```
virtual void OnPhantomLeave();
```

Function Description:

HapticMouse calls this hook when a 3D to 2D transition has occurred. You can force a PHANTOM Mouse leave transition by calling this function yourself. You may also want to override this routine to modify mouse configuration, like adjusting the mouse movement scale based on the PHANTOM devices's current Z depth.

HapticMouse Configuration

```
void EnablePHANToMMouse(BOOL bEnable);
BOOL IsPHANToMMouseEnabled();
```

Use these functions to enable/disable OnPhantomLeave() transitioning and query the current transitioning state. Calling EnablePHANToMMouse(FALSE) will disable mouse integration.

```
BOOL SetWhichButton(UCHAR buttonMask);
```

Allows you to modify the mouse buttons being emulated when a stylus click occurs. You can provide any bitwise combination of the following three masks:

MOUSE_LEFT_BUTTON_MASK
MOUSE_MIDDLE_BUTTON_MASK
MOUSE_RIGHT_BUTTON_MASK

Example:

To emulate a left / right mouse button chord click , pass in the following button mask.

```
SetWhichButton(MOUSE_LEFT_BUTTON_MASK | MOUSE_RIGHT_BUTTON_MASK);
```

```
void SetHapticEntryMode(HapticEntryMode mode);
```

Allows you to specify the entry mechanism that should be detected in order to transition from 2D back to 3D. Please refer back to the section on transitioning mechanisms to understand how these modes can be used. The possible modes are listed below:

PHANTOM_MOUSE_FALL_THRU
PHANTOM_MOUSE_CLICK_THRU
PHANTOM_MOUSE_PUSH_THRU

```
void SetPHANToMMouseConfig(const PHANToMMouseConfig &config);
```

This routine makes use of the following structure to maintain configuration settings for the PHANTOM Mouse. In general, you shouldn't have to modify these settings, since HapticView takes care of setting up the default values. However, in the event that you want to tweak some parameters about the mouse, this is the structure to look at. Changes made to this structure only get passed through to the mouse driver when a call to UpdateConfig() is made. In general, HapticView will handle updating the mouse driver configuration each time before a 3D-to-2D transition occurs. If the mouse driver is already active though, you should make the call to UpdateConfig() yourself.

If you're changing the configuration settings from within the HapticView derived class, then you can access the parameters directly via the m_phantomMouseConfig member. Otherwise, you'll need to use the GetPHANToMMouseConfig() and SetPHANToMMouseConfig() functions.

The fields below have limits that are denoted using the notation [min, max]. Additionally, the value in the ( ) will indicate the default setting

```
typedef struct {

    BOOL  bEnableMouse;        // must be set in order for the mouse to
                               // activate (TRUE)

    float nMouseScale;         // scale ratio between (x, y) movement in
                               // PHANTOM space and pixel space [1, 10] (1.5)

    float nMouseAccel;         // mouse acceleration [1, 10] (5.0)

    BOOL  bPlaneEnabled;       // determines whether a contact plane will be
                               // used for allowing mouse movement (TRUE)

    BOOL  bAutoPlaneDepth;     // overrides the plane depth setting to be at
                               // the initial Z depth of the PHANTOM (TRUE)

    float nPlaneDepth;         // z depth of the mouse plane [-200, 200] (0.0)

    float nPlaneAngle;         // angle (degrees) of cw rotation of plane
                               // about the X axis [0, 90] (0.0)

    float nPlaneForce;         // magnifier for the relative force between
                               // planes [1, 10] (8.0)

    float nPlaneDistance;      // magnifier for the relative distance between
                               // planes [1, 10] (5.0)

    BOOL  bPushThruEnabled;    // send an event message back signifying a push
                               // through event (FALSE)

    float nPushThruDepth;      // Z depth relative to the mouse plane that will
                               // cause a push through event [0, 20] (5.0)

    char  szPHANToMName[128];  // the name of the configured PHANTOM device
                               // ("Default PHANTOM")

} PHANToMMouseConfig;
```

Chapter 6: Extending the *GHOST SDK*

While the *GHOST SDK* provides an extensive set of tools for touch-enabling applications, new features may be added as desired. If existing *GHOST SDK* classes do not support some capability that is needed, the developer can add new capabilities on top of the *GHOST SDK's* architecture. The *GHOST SDK* lets you create new or modified geometry, dynamic, effects, and manipulator classes utilizing abstract node classes. See *Chapter 1: The GHOST API* for a description of node classes.

Creating New Geometry and Dynamic Nodes

This section describes the general methods needed for the creation of geometry and dynamic nodes. Specific methods for these nodes are described in later sections of this chapter. See *Chapter 2: Using the GHOST API* for a description of these node types and how they can be implemented in a scene graph.

Required User Defined Methods

All scene nodes share a set of common methods that *must be* defined when a new derived node class is created. The following methods--**getClassTypeId**, **getTypeId**, **isOfType** and **staticIsOfType** --*must* be present in all derived node classes. These methods allow the developer to query for the class type and subclass types of a node instance so that an application can perform class-specific operations on a node of initially unknown type. The following code fragments *must be* in the header and source files of the new derived class. The **<CLASSNAME>** should be replaced with the name of the new class and **<PARENTCLASS>** should be replaced with the name of the class that is being sub-classed.

Header file:

```
public:
      // Get type of this class.  No instance needed.
      static gstType getClassTypeId()
      {
            gstAssignUniqueId(<CLASSNAME>ClassTypeId);
            return <CLASSNAME>ClassTypeId;
      }

      // Get type of this instance.
      virtual gstType getTypeId() const
      {
            return getClassTypeId();
      }

      // Returns TRUE if class is same or derived class of type.
      virtual gstBoolean isOfType(gstType type) const
      {
            return staticIsOfType(type);
      }

      // Returns TRUE if subclass is of type.
      static gstBoolean staticIsOfType(gstType type) {
            if (type == getClassTypeId()) return TRUE;
            else return (<PARENTCLASS>::staticIsOfType(type));}

private:
      static gstType <CLASSNAME>ClassTypeId;
```

```
Source File:

gstType <CLASSNAME>::<CLASSNAME>ClassTypeId;
```

Optional User-Defined Methods

The methods described in this section are optional but can be used to create additional functionality.

Insert/remove from scene graph methods

putInSceneGraph and **removeFromSceneGraph** are optional. They allow the class to perform its own set of operations when the node is added and removed from the scene graph.

The following code excerpt illustrates how these methods should be used:

```
// Put transform in scene graph.
virtual void putInSceneGraph()
{
      <PARENTCLASS>::putInSceneGraph();

      // ADD YOUR CODE HERE
}

// Remove transform from scene graph.
virtual void removeFromSceneGraph()
{
      <PARENTCLASS>::removeFromSceneGraph();

      // ADD YOUR CODE HERE
}
```

NOTE: It is essential that the parent class's putInSceneGraph and removeFromSceneGraph methods get called from within the new class' definitions of these functions as shown above.

Adding state information to the Graphics Callback data structure

To add fields to the graphics callback data structure of the new node class, the developer should:

- Define the new data structure
- Allocate memory and point **graphicsCBData** to it in the constructor
- Create **prepareToUpdateGraphics** method that will copy the new fields into that new data structure when **gstScene::updateGraphics** is called

The following code excerpt shows callback data structures for derived **gstShape** and **gstDynamic** classes, respectively:

```
// Graphics callback data structure for this class.
// This replaces (i.e., is a superset of) gstShape’s data struct.
typedef struct _<CLASSNAME>GraphicsCBData {
      gstTransformMatrix  transform;
       // From gstTransformGraphicsCBData

      gstTransformMatrix cumlativeTrnsform;
      //cumulative transform from gstTransformGraphicsCBData

      // Add your own fields here

} <CLASSNAME>GraphicsCBData;

// Graphics callback data structure for this class.
// This replaces (i.e., is a superset of) gstDynamic’s data
// struct.
typedef struct _<CLASSNAME>GraphicsCBData {
      gstTransformMatrix  transform;
       // From gstTransformGraphicsCBData

      gstTransformMatrix cumlativeTrnsform;
      //cumulative transform from gstTransformGraphicsCBData

      gstVector         reactionForce;    // Newtons
      gstVector         reactionTorque;   // Newton mm
      gstVector         velocity;         // mm/s
      gstVector         acceleration;           // mm/s^2
      gstVector         angularVelocity;  // rad/s
      gstVector         angularAcceleration; // rad/s^2

      // Add your own fields here
} <CLASSNAME>GraphicsCBData;
```

Upon construction of an instance of the class, the developer *must* allocate space for the new data structure, and **gstTransform::graphicsCBData** *must* point to it. Since **gstShape** and **gstDynamic** already allocate space and set **graphicsCBData** to point their data structures in their own constructors, the developer should free the memory (as shown in the following code excerpt) *before* the value of **graphicsCBData** is allocated. The following is an example constructor for a new derived class.

```
// Constructor
<CLASSNAME>::<CLASSNAME> () {
      // If graphicsCBData was allocated by parent class free
      // it and replace with callback data struct for this
      // class which includes the same fields as the parent
      // classes callback struct plus those needed by this
      // class.

      free(graphicsCBData);
      graphicsCBData = malloc(sizeof(<CLASSNAME>GraphicsCBData));

      return 0;
}
```

The **prepareToUpdateGraphics** method is another optional method used to copy the added graphics callback fields defined for this class. If the class has a new callback data structure, all new fields in the structure *must* be updated here; then the parent class version of this function is called to update the remaining fields. The following code excerpt shows the definition of **prepareToUpdateGraphics**:

```
void <CLASSNAME>::prepareToUpdateGraphics()
{
      // Copy graphics data into callback data struct for
      //  this class.
      //  Remember to create space for the struct in your
      //  constructor and point cbData to it.
      ((<CLASSNAME>GraphicsCBData *)graphicsCBData)-><NEWFIELD>=
            <CURRENTSTATEOFFIELD>;

      <PARENTCLASS>::prepareToUpdateGraphics();
}
```

The new graphics callback data should now be correctly added and passed into any callback functions defined for instances of the corresponding class.

Creating a New Geometry Class Node

The previous functions are necessary when creating new *GHOST SDK* nodes. Geometry nodes require additional methods and are explained in this section.

GHOST SDK geometry objects are derived from the **gstShape** abstract class. The **gstShape** class defines a number of methods and state variables to support the common needs of any geometry class so that implementing the class requires a small set of geometry-specific information. The representation of these nodes is dependent on the assumptions the *GHOST SDK* makes about geometry nodes. These assumptions are described throughout the remainder of this section.

Detecting collisions with gstPHANToM nodes

To detect collisions with a **gstPHANToM** node, *GHOST SDK* expects every geometry object to have the methods **intersection** and **collisionDetect**, defined. These methods check for intersections or collisions and report the surface contact point (SCP) and normal for any collisions. Before describing how to implement these methods a general description of the collision detection process in *GHOST SDK* should prove helpful in fully understanding how these methods are utilized. In addition, we describe some utility structures and methods to help implement **collisionDetect**.

Collision Detection with *GHOST SDK*

GHOST assumes geometry class objects are impenetrable by **gstPHANToM** nodes. When the position of any **gstPHANToM** node in the scene graph crosses from the outside to inside (penetrates) the surface of a geometry node, the geometry node is treated as being in contact with the **gstPHANToM** node. It is then expected to suggest the representative point of contact on its surface, or SCP. For example, the surface contact point for a **gstPHANToM** crossing a planer surface is the projection of the **gstPHANToM** position onto the planer surface, as depicted in Figure 5 in Chapter 2.

The SCP does not always lie at the point of intersection of the line segment joining the last and current position of the **gstPHANToM** nodes with the surface -- as was the case for the planar surface. Instead, the SCP is best described as a point attached by a spring to the **gstPHANToM** position that moves to a surface position of minimum energy.

The SCP returned by a geometric node is described as *suggested* because it's used by the *GHOST SDK* to determine the actual SCP for the **gstPHANToM**. If a **gstPHANToM** is simultaneously contacting multiple geometry nodes, then each suggested SCP returned from the contacted geometry nodes is used to determine a single acceptable SCP for the **gstPHANToM**. One **gstPHANToM** contacting multiple geometry nodes implies that the **gstPHANToM** is at an edge or corner intersection of the geometries; thus, the final SCP reflects the edge or corner SCP.

The SCP's final position is further influenced by the addition of friction. The friction constants of the geometry object and the penetration depth of the **gstPHANToM** determine a final placement of the SCP somewhere between the last SCP and the proposed new SCP. After initial contact, contact with a geometry node is not broken until the line segment from the last SCP for the **gstPHANToM** to the current **gstPHANToM** position contains no intersections with the geometry's surface. In other words, after a surface is initially contacted, it is considered *touched* and a line segment is maintained from the last SCP to the current **gstPHANToM** position in each following servo loop until that line segment no longer intersects the object.

Allowing the PHANTOM point to pass inside an object

Under normal circumstances, *GHOST SDK* guarantees that the SCP remains on the outside of any (closed) surfaces. It is possible, however, for a geometry object to be placed over a **gstPHANToM** such that the **lastSCP** and the **gstPHANToM** appear inside the geometry (if a cube is created with its origin at the current **gstPHANToM** position). In this case, the object should *not* generate a contact state with the **gstPHANToM**. Instead, once the SCP is inside the geometry surface, the geometry should assume no contact is being made with the **gstPHANToM** until the SCP returns to the exterior of the geometry. This assumption prevents large forces from being generated when a **gstPHANToM** suddenly appears deep inside an object after the object is transformed.

Supporting code

To support the implementation of **collisionDetect** and maintain contact state information, **gstShape** provides the following structure and methods:

```
typedef struct _gstShapeStateArrayStruct {
      gstPHANToM *phantom;
      int inContact;
} gstShapeStateArrayStruct;

protected:
      gstShapeStateArrayStruct *stateArray;

public:

      virtual gstBoolean getStateForPHANToM(gstPHANToM
      *curPHANToM);
      virtual gstBoolean updateStateForPHANToM(gstPHANToM
      *curPHANToM);
      gstBoolean addCollision(gstPHANToM *);
```

stateArray:

This array holds the contact state of up to four **gstPHANToMs** currently in contact with the geometry. By default, the **gstShape** constructor points **stateArray** to an array of four **gstShapeStateArrayStructs**. The **phantom** field of **gstShapeStateArrayStruct** stores a pointer to a **gstPHANToM** node currently in contact with the geometry. The contact state is held in the **inContact** field of the **gstShapeStateArrayStruct** and is represented as an integer. A value of zero specifies no contact, while non-zero values of the contact state are supplied by the derived class and can be used to specify how the geometry is being contacted.

For example, **gstCube** defines 1-6 as valid values of the contact state, and it uses those values to identify the side on which the cube is being contacted. Thus, when entries in **stateArray** exist, they will always have non-zero values of **inContact** since the array *only* stores **gstPHANToMs** currently contacting the geometry. The methods **getStateForPHANToM** and **updateStateForPHANToM** should be used to access and store information in **stateArray**. The developer can add contact state information by defining a new state array structure and pointing **stateArray** to it in the constructor. If the user does define a new contact state structure then **getStateForPHANToM** and **updateStateForPHANToM** must be redefined in the new class to handle the new information in **stateArray**.

`getStateForPHANToM(gstPHANToM *phantom, integer inContact):`
This method is used to access the **stateArray**. Given the pointer **phantom** to a **gstPHANToM**, **getStateForPHANToM** finds an entry in **stateArray** corresponding to the **gstPHANToM** and retrieves the **inContact** field into the parameter *inContact*. If no entry for **phantom** exists in the array, no contact is assumed, and **inContact** is set to zero.

`updateStateForPHANToM(gstPHANToM *phantom, integer inContact):`
This method is used to store or update an entry in **stateArray**. Given the pointer to a **gstPHANToM**, **updateStateForPHANToM** finds an existing entry corresponding to **phantom** (or creates a new entry) and sets the **inContact** field to *inContact*. If no entry exists that corresponds to **phantom**, and if no space is available to add a new entry, then `FALSE` is returned; otherwise, **updateStateForPHANToM** returns `TRUE`.

`addCollision(gstPHANToM *phantom, gstPoint SCP, gstVector SCPnormal):`
addCollision informs **phantom** of contact with this geometry node, and stores the SCP and surface normal **SCPnormal** of contact. The **SCP** and **SCPnormal** should be in the local reference frame.

Required User Defined Methods

Intersection and **collisionDetect** are the *only* methods that *must* be defined in the new derived class to implement the new geometry. The following code excerpt shows the prototypes for these methods:

```
            virtual gstBoolean
      collisionDetect(gstPHANToM*phantomNode);

            virtual gstBoolean intersection(const gstPoint &startPt_WC,
              const gstPoint &endPt_WC,
                    gstPoint &intersectionPt_WC,
                    gstVector &intersectionNormal_WC,
                    void **data);
```

`collisionDetect`

The servo loop calls **collisionDetect** whenever a **gstPHANToM** enters the boundary sphere of the geometry object to check if the **gstPHANToM** is in contact with the object. See the next section for a description of how to maintain the boundary sphere for a geometry object. **collisionDetect** is passed a pointer to the **gstPHANToM** node and is expected to supply a suggested SCP and surface normal to the **gstPHANToM** node, using **gstShape::addCollision**, and return `TRUE` if the **gstPHANToM** is in contact with the geometry node; otherwise, it returns `FALSE`.

`intersection`

In **collisionDetect**, an object uses the current phantom position to return a suggested SCP based on where it believes the **gstPHANToM** is contacting the object. It is possible, however, that the surface of another geometry object could lie between the **gstPHANToM** position and the suggested SCP.

The **intersection** method is used by the servo loop to ensure that the SCP never intersects the surface of another geometry object. If an intersection with the line segment from **startPt_WC** to **endPt_WC** exists, the closest intersection to **startPt_WC** should be placed into **intersectionPt_WC**, the surface normal at the point of intersection placed in **intersectionNormal_WC**, and **TRUE** returned; otherwise, it returns **FALSE**. The post-fixed **_WC** indicates that these points are all in the world coordinate reference frame.

Maintaining a Correct Bounding Volume

Because collision detection routines consume processor time, *GHOST SDK* provides bounding volumes as an optimization to allow the haptics process to only check for collisions if the **gstPHANToM** position is within the vicinity of an object.

GHOST Version 2 supports both a Sphere and Box bounding volume for use by the gstBoundedHapticObj classes. The gstShape and gstForceField classes now inherit from the gstBoundedHapticObj class.

Previously, the gstShape class explicitly used internal data members to implement a bounding sphere to trigger the evaluation of the PHANTOM interaction with the shape. The developer would set a boundingRadius data member to define the size of the bounding sphere used.

Through interfaces of the gstBoundedHapticObj class, the developer now has the capability to specify a BoundingSphere or BoundingBox to encapsulate the shape. The effect is that the developer can pick a more tightly fitting bounding volume for their shape and reduce the number of calls to collisionDetect for gstShape instances.

The gstShape class implements a BoundingSphere as its default bounding volume. In order to adjust the bounding sphere radius, the developer would call `setBoundingRadius(this, const double new_radius);` from within a method a class inheriting from gstBoundedHapticObj. This call can be made from outside the classes method by using the first argument to point to the gstBoundedHapticObj to be modified.

The gstBoundedHapticObj supports two methods for easy construction of the bounding volume:

```
gstBoolean boundBySphere(const gstPoint &center, const double radius);
gstBoolean boundByBox(const gstPoint &center, const double xlen, const
double ylen, const double zlen);
```

The developer would call the appropriate method for box or sphere with the parameters for defining the geometry of the volume. The gstBoundedHapticObj.h file should be consulted for other interfaces supported.

Example

The following is a self-contained example that creates a simple disk class in the *GHOST SDK* and describes how the above methods should be implemented and used. The disk a symmetrical planar object aligned along the z=0 plane, with a specified radius.

Step 1: Constructors
Each geometry class object should contain a set of constructors. The following constructors enable the application to create a default object or duplicate an existing object. Any geometry-specific information, such as radius or height, should be initialized as shown.

```
gstObject::gstObject() {
}

gstObject::gstObject(const gstObject *);
gstObject::gstObject(const gstObject &);
```

Step 2: Class information
Each geometry class contains an identifier that allows the developer to query an abstract object to determine if it is of a certain type or is a subclass of a certain type.

A: Creating the identifier
Each *GHOST SDK* object is assigned a unique identifier that can be used to distinguish between object types. The identifier is created the first time it is accessed during runtime. Each class uses a global function, **gstAssignUniqueId(gstType &)**, to uniquely assign its identifier exactly once. The function takes care of all worries about setting the identifier more than once.

```
gstType gstObject::gstObjectClassTypeId;
```

B: Checking class type
The following functions allow the developer to access the type of this object.

```
static gstType getClassTypeId()
{
      gstAssignUniqueId(gstObjectTypeId);
      return gstObjectTypeId;
}

virtual gstType getTypeId() const
{
      return getClassTypeId();
}
```

C: Checking subclass type
The following functions allow the developer to query to determine if this object is of **OBJECT** class, or if the object is a subclass of **OBJECT**.

```
static gstBoolean staticIsOfType(gstType type)
{
      if (type == getClassTypeId()) return TRUE;
      else return (gstShape::staticIsOfType(type));
}

virtual gstBoolean isOfType(gstType type) const
{
      return staticIsOfType(type);
}
```

NOTE: The actual gstObjectTypeId appears only FOUR (4) times. Once in the declaration of the class, once in the source for instantiation purposes, and twice in the static function getClassTypeId(). The string appears nowhere else.

Step 3: Interaction information
The **collisionDetect** and **intersection** routines determine how the object interacts with the PHANTOM haptic interface. Each object will have a different set of intersection and collision routines specific to its. This is the *only* geometry-specific code that must be included when defining a new geometry class.

A: The intersection routine checks to determine if a line segment intersects the object. If it returns **TRUE**, the *GHOST SDK* expects the intersection point and normal vector at that point to be set accordingly. Some psuedo-code is provided here, then the actual code for the **OBJECT** class.

Pseudo-code:

```
Convert start and end points to the local coordinate system
      of the object.
Check if the line segment defined from startPt_LC to endPt_LC
      intersects the object.
If (the line segment defined from startPt_LC to endPt_LC
      intersects the object) then
      Calculate the point of intersection and set
            intersectionPt_WC to that point in world
            coordinates.
      Calculate the normal at the point of intersection
            and set intersectionNormal_WC to be that point
            in world coordinates.
      return TRUE
  else
      return FALSE
```

General template:

```
gstBoolean intersection(const gstPoint &startPt_WC,
                        const gstPoint &endPt_WC,
                        gstPoint &intersectionPt_WC,
                        gstVector &intersectionNormal_WC,
                        void **userData) {
      gstVector startPt_LC = fromWorld(startPt_WC);
      gstVector endPt_LC = fromWorld(endPt_WC);
      if (object_intersects_line(startPt_LC,endPt_LC)) {
            intersectionPt_LC =
            pointOfIntersection(startPt_LC,endPt_LC);
            intersectionNormal_LC =
            normalOfObjectAtPoint(intersectionPt_WC);
            intersectionPt_WC = toWorld(intersectionPt_LC);
            intersectionNormal_WC =
            toWorld(intersectionNormal_LC);
            return TRUE;
      }
      else return FALSE;
}
```

Code for OBJECT:

```
gstBoolean gstObject::intersection(const gstPoint &startPt_WC,
                                   const gstPoint &endPt_WC,
                                   gstPoint &intersectionPt_WC,
                                   gstVector
&intersectionNormal_WC,
                                   void **) {
  // Get the start and end position of the line segment against
  // which the collision will be checked.
  gstPoint P1 = fromWorld(startPt_WC);
  gstPoint P2 = fromWorld(endPt_WC);

  // Check if the line segment intersects the plane.
  // In this case, check if the line segment crosses the
  // Z-axis.  This is true if the signs of the Z end points
  // are different, i.e. the product of the Z end points is
  // negative.
  // Return FALSE if there is no intersection.
  if (P1[2]*P2[2] > 0)
    return FALSE;

  // Calculate the intersection point.
  gstPoint intersectionPoint = P1+gstVector(P2-
P1).normalize()*P1[2];

  // Check if the collision is within the disk radius.
  // If not, return FALSE.
  if (intersectionPoint.norm() > 1)
    return FALSE;

  // Set the intersection point and normal.
  intersectionPt_WC = toWorld(intersectionPoint);
  intersectionPtNormal_WC = toWorld(gstVector(0,0,1));
  return TRUE;
}
```

B: The **collisionDetect** routine determines whether the PHANTOM haptic interface device intersects an object. The geometry-specific code will be similar to intersection, with the exception that **collisionDetect** should set the SCP and the normal of that point as shown.

Pseudo-code:

```
Retrieve contact state for gstPHANToM node.
Create a line segment from the previous position of the SCP
      to the current position, in the local coordinate frame of
      the object.
If (the line segment intersects the object) {
      Calculate the surface contact point, defined as the point
      on the surface of the object such that the distance between
      that point and the phantom position is minimized. Set the
      variable SCP, defined in gstShape, to this value in local
      coordinates.
      Calculate the normal of the object at the SCP. Set the
      Variable SCPnormal, defined in gstShape, to this value in
Local coordinates.
      Update contact state in state array for PHANTOM
      Add the object to the collision list of the PHANTOM.
      return TRUE
   else
      Update contact state in state array for PHANTOM
      return FALSE
```

General template:
```
gstBoolean collisionDetect(gstPHANToM *PHANTOM) {
      gstPoint lastPos, SCP;
      gstVector SCPnormal;
      int inContact;

      inContact = getStateForPHANToM(PHANTOM);

      // Refer to release notes for info on this if statement.
      if (!_touchableByPHANToM || _resetPHANToMContacts) {
            _resetPHANToMContacts = FALSE;
            inContact = FALSE;
            (void) updateStateForPHANToM(PHANTOM,inContact);
            return inContact;
      }

      PHANTOM->getLastSCP_WC(lastPos);
      lastPos = fromWorld(lastPos);
      gstPoint curPos;
      PHANTOM->getPosition_WC(curPos);
      curPos = fromWorld(curPos);
      if (object_intersects_line(lastPos,curPos) {
            SCP = SCPofIntersection(lastPos,curPos);
            SCPnormal = normalOfObjectAtPoint(SCP);
            inContact = TRUE;
            (void) updateStateForPHANToM(PHANTOM,inContact);
            addCollision(PHANTOM,SCP,SCPnormal);
            return inContact;
      }
      inContact = FALSE;
      (void) updateStateForPHANToM(PHANTOM,inContact);

      return inContact;
}
```

Code for OBJECT:
```
      gstBoolean gstObject::collisionDetect(gstPHANToM *PHANTOM) {
            gstPoint lastSCP,phantomPos, intersectionPoint;

            int inContact;

            inContact = getStateForPHANToM(PHANTOM);

            // Refer to release notes for info on this if statement.
            if (!_touchableByPHANToM || _resetPHANToMContacts) {
                  _resetPHANToMContacts = FALSE;
                  inContact = FALSE;
                  (void) updateStateForPHANToM(PHANTOM,inContact);
                  return inContact;
            }

            // Get the start and end position of the line segment
            // against which the collision will be checked.
            PHANTOM->getLastSCP_WC(lastSCP);
            lastSCP = fromWorld(lastSCP);
            PHANTOM->getPosition_WC(phantomPos);
```

```
        phantomPos = fromWorld(phantomPos);

        // Check if the line segment intersects the plane.
        // In this case, check if the line segment crosses the
        // Z-axis.  This is true if the signs of the Z end points
        // are different, i.e. the product of the Z end points is
        // negative.
        // Return FALSE if there is no intersection.
        if (lastSCP[2]*phantomPos[2] > 0) {
              inContact = FALSE;
              (void) updateStateForPHANToM(PHANTOM,inContact);
              return inContact;
        }

        // Calculate the intersection point.
        gstPoint intersectionPoint = lastSCP+gstVector(phantomPos-
        lastSCP).normalize()*lastSCP[2];
        // Check if the collision is within the disk radius.
        // If not, return FALSE.
        if (intersectionPoint.norm() > 1) {
              inContact = FALSE;
              (void) updateStateForPHANToM(PHANTOM,inContact);
              return inContact;
        }

        // Project the current phantomPosition onto the plane to
get
        // the SCP.
        // In this case, this is simply
        // gstPoint(phantomPos[0],phantomPos[1],0).
        gstPoint SCP = gstPoint(phantomPos[0],phantomPos[1],0);
        // Set the SCPnormal to be the normal of the plane.
        // In this case, the normal is (0,0,1);
        gstVector SCPnormal = gstVector(0,0,1);

        // Add the SCP and SCPnormal to the collision list of
        // the PHANTOM.
        inContact = TRUE;
        (void) updateStateForPHANToM(PHANTOM,inContact);
        addCollision(PHANTOM,SCP,SCPnormal);
        return inContact;
}
```

Creating a new dynamic node class

The **gstDynamic** abstract node class is used to implement dynamic groups in the scene graph that enable geometries to react to contact with **gstPHANToM** nodes and other geometry nodes in the scene. For example, a cube in the scene graph can be given dynamic state such that a PHANTOM device can push along the negative Z-axis to displace the cube (as if pressing a button). Realistic reaction forces can then be calculated so that the cube behaves as a spring-mass system resisting movement from its place of rest. More complicated dynamics can be implemented, such as for rigid body dynamic objects. These objects can be pushed and rotated with the PHANTOM device, collide with walls, and collide with other geometries in the scene graph.

The dynamics mechanism is designed such that you can use existing geometries in the scene graph. A hierarchical node above the geometries provides these geometries with their desired dynamic behavior. This mechanism generalizes the dynamic behavior away from a particular geometry representation, allowing previously defined geometries to be enabled with dynamic state. Thus, just as a cube can be made into a button, multiple geometries under a separator can also be made to behave as a single button.

The *GHOST SDK* considers nodes derived from **gstDynamic** to be dynamic state nodes. These nodes provide the geometry represented in the subtree below the **gstDynamic** node with state such as velocity, mass, and angular acceleration. Contact forces of any **gstPHANToM** node contacting a geometry node in the subtree are used to calculate a reaction force and torque that *GHOST SDK* then passes to the **gstDynamic** node. Additional reaction forces and torques may be sent to the dynamic node from other dynamic nodes or the dynamic node can add them from within itself. During each servo loop, *GHOST SDK* calls a user-defined method to update the dynamic state of any **gstDynamic** node with existing linear or angular velocity, or with a force or torque acting on it.

Supporting code
The **gstDynamic** abstract class provides the framework needed to generate and update the dynamic state of an object. The following excerpt lists and explains the relevant supporting code:

```
gstVector       reactionForce;  // Newtons
gstVector       reactionTorque; // Newtons/m
static double   deltaT;         // 1/updateRate
double          damping;        // damping constant
double          mass;           // mass of dynamic object
                                // represented by sub-tree
                                //  below this node.
gstVector       velocity;       // mm/s
gstVector       acceleration;   // mm/s^2
gstVector       angularVel;     // rad/s
gstVector       angularAccel;   // rad/s^2

// Set homogenous transformation matrix for dynamic object
// (i.e., member of gstDynamic class).
void setTransformMatrixDynamic
const gstTransformMatrix &matrix);

// Set center (absolute) for dynamic object
// (i.e., member of gstDynamic class).
void  setCenterDynamic(const gstPoint &newCenter);

// Set position (absolute) for dynamic object
// (i.e., member of gstDynamic class).
void  setPositionDynamic(const gstPoint &newPos);

// Set translation (cumulative) for dynamic object
// (i.e., member of gstDynamic class).
void  translateDynamic (const gstPoint &translation);

// Set translation (absolute) for dynamic object
// (i.e., member of gstDynamic class).
void  setTranslateDynamic (const gstPoint &translation);

// Set rotation (cumulative) for dynamic object
// (i.e., member of gstDynamic class).
void  rotateDynamic(const gstVector &axis, double rad);

// Set rotation (absolute) for dynamic object
// (i.e., member of gstDynamic class).
void  setRotateDynamic(const gstVector &axis, double rad);
```

```
// Set scale (cumulative) for dynamic object
// (i.e., member of gstDynamic class).
void  scaleDynamic(double scale);

// Set scale (absolute) for dynamic object
// (i.e., member of gstDynamic class).
void  setScaleDynamic(double newScale);

// Used by system or for creating sub-classes only.
// Add force and torque pair if force is above threshold
// and node is in scene.  Force and torque calculated
// from this call are used in the next call to
// updateDynamics.
void  addForceTorque(const gstVector &force_WC,
                     gstPoint &SCP_WC);
```

reactionForce & reactionTorque

These vector fields hold the reaction force and torque in the local reference frame.

deltaT
This member variable is the time change (in fractions of a second) since the last dynamic update. It is expected to be close to 0.001seconds.

Optional state variables
The *GHOST* SDK provides the following state variables for convenience that can be optionally used by the derived dynamic class. Those variables that are not useful to the new class can be ignored.

mass
This member variable is the mass in kilograms of the dynamic object. It represents the cumulative mass of all geometry nodes in the subtree beneath this node.

damping
This member variable is the damping constant for the dynamic object. Damping adds force proportional to the dynamic object's velocity. Force attributed to damping is **F = -damping*velocity**. Units are measured in Kg /(1000.0*sec).

velocity, acceleration, angular velocity & angular acceleration
These state variables can be used to hold linear velocity (mm/sec), linear acceleration (mm/(sec^2)), angular velocity (rad/sec) and angular acceleration (rad/(sec^2)).

Dynamic transforms
It is possible for an object to be transformed such that the **gstPHANToM** suddenly becomes entirely encapsulated within the object (for example, if an object is scaled so that its extent encompasses the **gstPHANToM**). Ordinarily, the *GHOST SDK* considers the object untouched until the **gstPHANToM** exits the object. In the case of dynamic objects, however, this behavior is undesirable. If a dynamic object moves against a **gstPHANToM**, the **gstPHANToM** should be pushed away instead of allowed into the object. The *GHOST SDK*, therefore, provides a list of transformation methods ending in **Dynamic** that are used to dynamically position and orient the **gstDynamic** node. These methods differ from the normal transformation methods defined in **gstTransform** in that they prevent the PHANTOM haptic interface device from falling into the moving geometry.

addForceTorque

This method can be used to add additional force and torque to a dynamic object. The force is converted from the world reference frame to the dynamic node's local reference frame, then it is added to **gstDynamic::reactionForce**. The torque is calculated as the cross product of the force in the local reference frame with the vector - from the local reference frame origin to the local reference frame position of the SCP.

Required User Defined Methods

```
            // Used by system or for creating sub-classes only.
            // When subclassing, call gstDynamic::updateDynamics()at
            // the end of your updateDynamics() procedure.
            virtual void       updateDynamics();
```

updateDynamics
The new derived gstDynamic class defines this method. It uses the previously mentioned dynamic state and time step to calculate a new dynamic state. Normally, an Euler integration of the state variables over the time step (deltaT) is used to calculate a new dynamic state. Refer to the provided **gstFreebody** *GHOST SDK* code in the multy and multy3d programs for an example of this type of *GHOST SDK* extension.

Creating a New gstEffect Class

The **gstEffect** classes use the current state of a **gstPHANToM** node and calculate an additional force to be appended to the output force of that **gstPHANToM** node. These can create forces such as vibration (buzz), viscous field, constraints, or any additional effects beyond object surface collision forces. *GHOST SDK* includes a few classes derived from **gstEffect**: **gstBuzzEffect**, **gstConstraintEffect**, and **gstInertiaEffect**. Refer to the *GHOST API Reference Guide* for detailed description of each class.

To use a **gstEffect** class, the developer *must* create an instance of the effect and define a pointer to that instance, which is passed to an instance of **gstPHANToM** through the **gstPHANToM::setEffect** method. The effect is then activated by a call to **gstPHANToM::startEffect**. A **gstEffect** can be designed to remain active for a specified time interval, or until the **gstPHANToM::stopEffect** method is called.

Supporting code

The **gstEffect** abstract class implements a basic framework to support derived effects classes. The following excerpt lists and explains the relevant supporting code:

```
      double                  time;
      gstBoolean              active;

      // Start the effect.  WARNING:  When re-starting an effect,
      // make sure to reset any state, such as past PHANTOM position.
      // Otherwise, the next call to calcEffectForce could
      // generate unexpectedly large forces.
      virtual gstBoolean start() {
           active = TRUE; time = 0.0; return TRUE;
      }

      // Stop the effect.
      virtual void stop() {
           active = FALSE;
      }
```

time
This member variable is an optional state variable that holds the elapsed time since the effect was started. It is useful for generating timed effects. The explanation for **calcEffectForce** in the *Required Defined Code* section describes how to keep time current.

active
This state variable is set to TRUE when **gstEffect::start** is called, and it is reset to FALSE when **gstEffect::stop** is called. The **calcEffectForce** method also is described in the *Required User-Defined Code* section.

start & stop
Start resets **time** to zero and sets **active** to TRUE.
Stop sets **active** to FALSE.

Required User-Defined Code

```
      // Calculate the force.  When subclassing, the first
      // parameter should be cast to gstPHANToM to retrieve
      // any information about the state of the PHANTOM that
      // is needed to calculate the forces.  deltaT should
      // be used to update the time.  Also, if the effect is
      // not active, the zero vector should be returned.
      virtual gstVector calcEffectForce(void *PHANToMNode) {
           PHANToMNode; // To remove compiler warning.
           return gstVector(0.0,0.0,0.0);
      }
```

calcEffectForce
This method is called during every servo loop and immediately before a force is sent to **PHANToMNode**, if a pointer to this instance of a **gstEffect** derived class has been passed to **PHANToMNode** through the **gstPHANToM::setEffect** method. If **gstEffect::active** is TRUE, a non-zero force is. This force is then appended to the output force of **PHANToMNode**; otherwise, the zero vector is returned.

The **gstEffect::time** state variable is set to zero when **active** is switched from FALSE to TRUE. The **gstEffect::time** can then be updated during each call to **calcEffectForce** by adding the value **PHANToMNode->getDeltaT()** to **gstEffect::time**.

NOTE: The force returned from the effect should be with respect to the reference frame of the parent of **PHANToMNode** because the local reference frame of a PHANTOM device is the stylus reference frame. This reference frame differs from the PHANTOM device ground reference frame. See the previous section describing the **gstPHANToM**. Refer to the *GHOST SDK* effects example code for more information.

Creating a New gstManipulator Class

The **gstManipulator** abstract class is implemented in a manner similar to the **gstEffect** class but it is used differently. A **gstManipulator** derived class uses **gstPHANToM** state to manipulate a specified **gstNode** in the scene graph, and it can send reaction forces back to the **gstPHANToM** node.

The three derived **gstManipulator** classes are **gstTranslateManipulator**, **gstScaleManipulator**, and **gstRotateManipulator**. These classes are used to translate, scale, or rotate a specified node in the scene graph using the **gstPHANToM** node. They are designed such that **gstPHANToM::startManipulator** or a depressed stylus switch activates the manipulator. Once the stylus switch is released or **gstPHANToM::stopManipulator** is called, the manipulator becomes inactive. These classes also calculate resistance forces to their transformation operations and allow those forces to be applied to the gstPHANToM node. For example, the **gstRotateManipulator** restricts the user to a spherical shell that the **gstPHANToM** can move around to rotate the object, and appropriate rotational dynamics are calculated for the object modeled as a mass spinning freely on a point of rotation.

Manipulators are convenient because the scene graph is left intact and a **gstManipulator** is the *only* object that needs to be created. Once the **gstManipulator** is created, the developer *must* set the pointers in **gstPHANToM** and **gstManipulator** correctly. See Chapter 2 for an example of a manipulator connecting a **gstPHANToM** to a node in the scene graph.

Supporting Code

The **gstManipulator** abstract class implements a basic framework to support a derived manipulator class. The following excerpt lists and explains the relevant supporting code.

```
gstBoolean     active;
gstTransform   *manipNode;


virtual void start() {
     active = TRUE;
}

// Stop the effect and reset state.
virtual void stop() {
     active = FALSE; resetState();
}

// Any dynamic state of manipulator is reset.
virtual void     resetState() {}
```

active
This state variable is set to `TRUE` when **gstManipulator::start** is called; it is reset to `FALSE` when **gstManipulator::stop** is called. The **calcManipulatorForce** method described in the *Required User-Defined Methods* section describes how to use this variable.

start & stop
Start resets **time** to zero and sets **active** to `TRUE`.
Stop sets **active** to `FALSE`.

Required User Defined Methods

```
// Calculate the force.  When subclassing, the first
// parameter should be cast to gstPHANToM to retrieve
// any information about the state of the PHANTOM that
// is needed to calculate the forces and move manipulated
// node.  Also, if the manipulator is not active, the
// zero vector should be returned.
// ACTUNG!
// WARNING!: Never call PHANTOM->setForce or
//  PHANTOM->setForce_WC from this function.
//  It will cause an infinite recursion.
virtual gstVector calcManipulatorForce(void *PHANToMNode) {
     PHANToMNode; // To remove compiler warning.
     return gstVector(0.0,0.0,0.0);
}
```

calcManipulatorForce

This method is called during every servo loop immediately *before* a force is sent to **PHANTOM** node *if* a pointer to this instance of a **gstManipulator** derived class is passed to **PHANTOM** node through the **gstPHANToM::setManipulator** method. If **gstManipulator::active** is **TRUE**, a non-zero force is returned. This force is appended to the output force of **gstPHANToM**; otherwise, the zero vector is returned.

NOTE: The force returned from the manipulator should be with respect to the reference frame of the parent of **gstPHANToM** because the local reference frame of a *PHANTOM* device is the stylus reference frame. This reference frame differs from the ground reference frame of the *PHANTOM* device. This is documented in the **gstPHANToM** section above. See the previous section describing the **gstPHANToM**. Refer to the effect *GHOST SDK* example code for more information.

Chapter 7: Demonstration and Example Applications

A collection of demonstration programs is provided to illustrate some of the *GHOST SDK's* capabilities. Also, the source code for most of the demonstrations (along with the appropriate Visual C++) is provided. Although these programs do *not* cover all the capabilities of the API, they *do* illustrate key features of the *GHOST SDK* and provide a foundation for building more complex, haptically enabled applications. The OpenGL graphics API is used in all examples because of its flexibility and platform independence.

NOTE: The *GHOST SDK* is independent of OpenGL and can be used with other graphics toolkits and environments that interface with C++ libraries.

These applications are divided into two categories: simple examples that illustrate how to use some part of the API and more complex demonstration applications that put it all together.

Example Programs

The example programs and source code are located in the exmaples directory of the distribution.

HelloGhost

The simplest GHOST SDK program. It creates a sphere that you can touch with no graphics.

HelloGhostGL

Uses the GHOSTGL library to add graphics to HelloGhost.

HelloSphere

Uses the gstDeviceIO class to gain low-level access to the PHANTOM

Robot

Demonstrates building a more complicated scene graph. Creates the torso of a robot with a head and arms by combining spheres and cylinders. No graphics.

RobotGhostGL

Adds graphics using GHOSTGL to the robot example.

RobotColor

Demonstrates how customize GHOSTGL rendering by changing the color of the robot's head.

RobotWorkspace

Adds a workspace boundary cube object to the RobotGhostGL example.

RobotFramework

Demonstrates using the HapticView Framework. Uses the same robot scene but uses the HpaticView framework to integrate with MFC (on Windows).

TriPolyMesh

Demonstrates using the gstTriPolyMesh class to create polygonal models in GHOST SDK.

PhantomContext

An advanced demonstration of the HapticView library and the PHANTOM Mouse Driver. Shows how to change the mouse transitioning modes and the stylus button behavior. Also demostrates the use of the GHOSTGL pinch transform.

ExtendingEffect

Demonstrates deriving your own class from gstEffect. Creates a viscosity effect.

ExtendingForceField

Demonstrates creating your own force field derived class. Creates a sphere with magnetic attraction.

ExtendingShape

Demonstrates creating your own shape geometry by subclassing gstShape. Creates a toucahble plane.

Dual PHANTOM

This program is identical to Blocks except that it allows you to use two PHANTOM devices at once. The two devices should be facing each other and there should be six centimeters between the gimbal endpoints when when the devices are in the reset position. If you have thimbal adapters, you can insert your index finger into one thimble and your thumb into the other, allowing you to grab the virtual blocks between your thumb and forefinger and lift them up. Source code is provided.

Demonstration Applications

These applications show some of the capabilties of GHOST SDK. Source code is only provided for some of these applications in the examples directory but not for others.

Blocks

The Blocks program consists of two cube objects located inside a box-shaped haptic workspace. The objects have simple three-dimensional dynamics and collision detection. There is an option for enabling/disabling gravity.

Dice

The Dice demo consists of a single cube bounded by a box-shaped haptic workspace. This program makes use of more advanced dynamics and collision detection. When used with a 6DOF PHANTOM device this demo demonstrates GHOST SDK's ability to generate both forces and torques. There are options for enabling/disabling gravity, adjusting dynamic properties of the cube, and if you click the stylus while touching the cube, you can pick it up. Source code is provided.

Lump

In the lump demo there is a kidney that you can feel. You can push through into the interior of the kidney that will feel viscous. Inside the kidney there is a spherical tumor. By touching the tumor and holding the stylus button you can drag the tumor out of the kidney. You can also use the stylus on the PHANTOM device to orient the model by holding down the "g" key and the stylus button. This program makes use of the VRML reader to load in the polygonal model of the kidney. It also uses the effects and manipulator classes. Source code is provided.

Effects

The Effects program demonstrates some of the different force effects in the *GHOST SDK* such as buzzing, constraints and inertia.

TouchVRML

The TouchVRML demo allows you to visually and haptically render a large subset of VRML 2.0 files. It utilizes the GHOST VRML 2.0 parser and some simple OpenGL code to demonstrate some basic features and interactions with VRML 2.0 objects.

WorldBuilder

WorldBuilder is an interactive scene editor. It lets you create touchable objects by combining different basic shapes. In addition to setting the color of any of the shapes you can also set the haptic stiffness and damping. Scenes created with WorldBuilder can be saved and reloaded. A set of sample scenes is provided as well as full source code. For more information on WorldBuilder refer the file WorldBuilderREADME.doc located in the 3DTouchDemos directory.   WorldBuilder is not provided on Irix. Instead there is a WorldViewer program which lets you see and feel files created with WorldBuilder but not edit them. To use WorldViewer you should specify the name of the file to read as a command line argument.

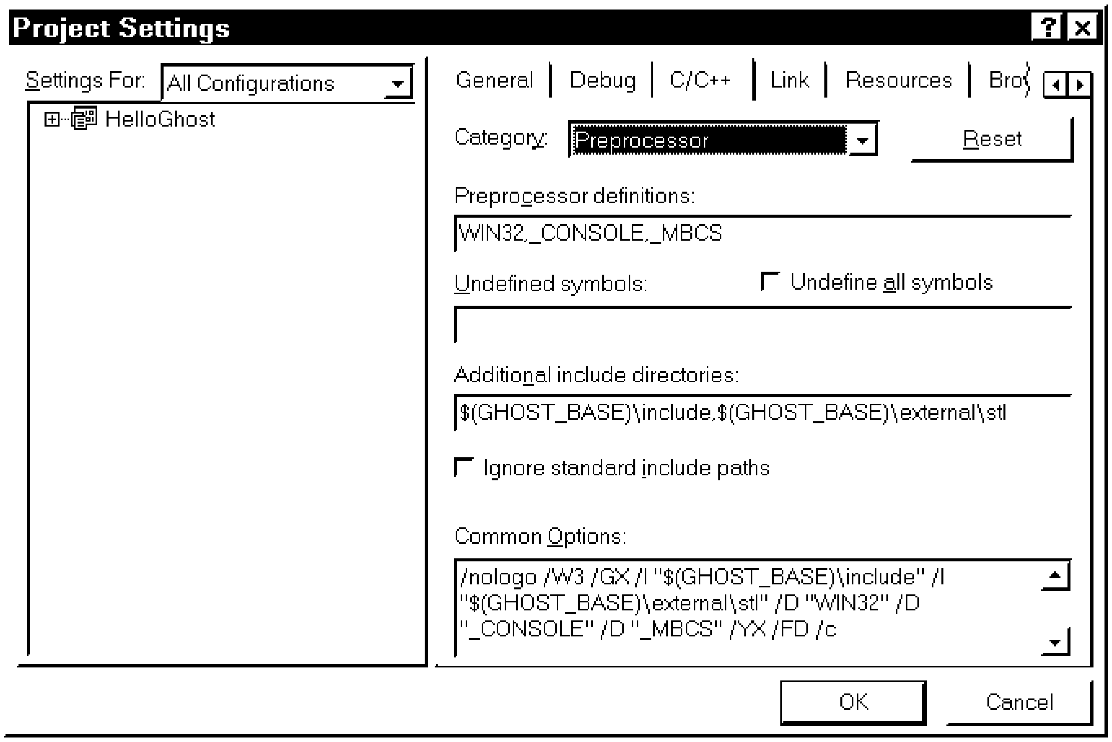
Glossary

Accessor	A member function of a class object that returns internal state values.
Ancestor	Any *node* in the *path* from the *root node* to any given *node* in a *node graph*, excluding the terminating *node* of the *path*. An ancestor is a parent (or a parents parent etc.) of a *node* in the *node graph*.
Base Class	A class from which other classes are derived.
Child	A *node* directly below a *separator node* in a *node graph*. A *node* added to a *separator* through the addChild method.
Children	All *nodes* directly below a *separator node* in a *node graph*. All *nodes* added to the graph through a *separator's* addChild method.
Class	The declaration and definition of an object's interface and data.
Descendant	A *node* that can be reached by a valid *path* starting from a given node in a *node graph*.
Haptic	Of or relating to 3D Touch technology or active touch -- the active employment of small feature detection and macro force sensation.
Homogeneous Transform Matrix	A 4x4 matrix whose upper left 3x3 sub-matrix specifies a rotation and scaling and the right three elements of the last row specifies a translation.
Local-Reference Frame	Reference frame defined by coordinate axes with origin and orientation defined by an object or *node* in a *node graph*. For example, the local reference frame of a **gstCube** has the coordinate axes origin at the center of the cube with axes orientation pointing outward towards the right, top and front cube face for x, y, and z respectively.
Node	An instance of one of the *node* classes defined by the *GHOST SDK* (for example, **gstCube**, **gstSeparator**, etc.) and which is used in the *scene graph*.
Node Graph	Collection of *nodes* forming a tree structure such that one *node*, the *root node*, has no *parent* and all other *nodes* in the collection are *descendents* of the *root node*.
Object	An instantiated class.
Parent	The *node* directly containing one or more other *nodes*.
Path	A list identifying the sequence of *nodes* leading from the root node of a *scene graph* to another *node* contained within the *scene graph*.
Root Node	The top-level *node* of a *node graph*
Scene Graph	A *node graph* whose *root node* was added to a **gstScene** instance through the **gstScene:setRoot** method.

Siblings	*Nodes* with the same *parent*.
Separator	A *node* in a *node graph* that can have children *nodes*.
Transform	A geometric transformation. Usually specified by a 4x4 *homogenous transform matrix*.
World-Reference Frame	Reference frame of the *root node* in the *scene graph*. All *nodes* in the *scene graph* are relative to this reference frame.

Appendix A: Compiling GHOST SDK Programs on Windows

This section describes how to set up a Microsoft Developer Studio project to use GHOST SDK. GHOST SDK supports version 6 of Microsoft Visual C++.



Settings For: All Configurations

Category: Preprocessor

Preprocessor definitions: WIN32,_CONSOLE,_MBCS

Undefined symbols:

Undefine all symbols

Additional include directories: $(GHOST_BASE)\include,$(GHOST_BASE)\external\stl

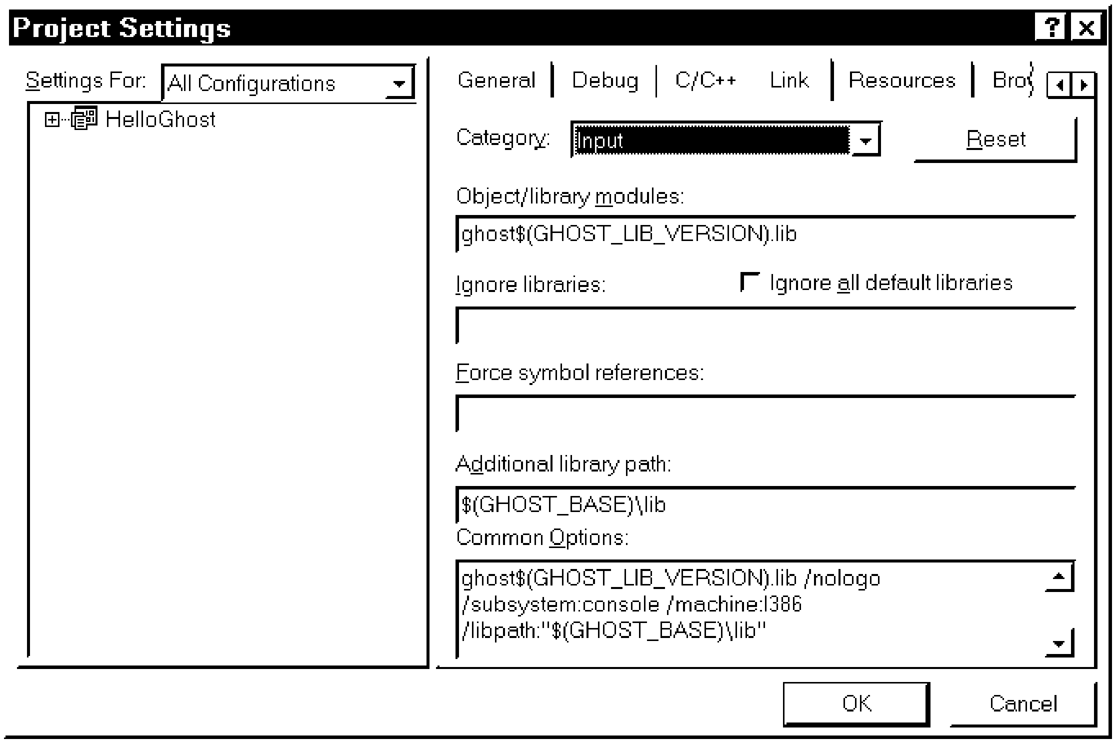
Ignore standard include paths

Common Options:

```
/nologo /W3 /GX /I "$(GHOST_BASE)\include" /I
"$(GHOST_BASE)\external\stl" /D "WIN32" /D
"_CONSOLE" /D "_MBCS" /YX /FD /c
```

First the project include path must point to the GHOST SDK include files and the Standard Template Library (STL) include files that are shipped with GHOST SDK. The STL that comes with Visual C++ is not compatible with GHOST SDK so you must have the include path for STL before any of the standard Visual C++ include paths. To set up the include path select “Settings...” from the “Project” menu. Select the “C/C++” tab. In the “Category” pop-up menu select “Preprocessor”. Select “All Configurations” in the “Settings For:” pop-up menu. Add the following to the “Additional include directories:” field: “$(GHOST_BASE)\include,$(GHOST_BASE)\external\stl”.

These include paths reference a system environment variable “GHOST_BASE” which stores the location of the GHOST distribution on the local machine. This variable is set during installation and should only be changed if you move the GHOST installation to a new location.



Object/library modules: ghost$(GHOST_LIB_VERSION).lib

Ignore libraries:

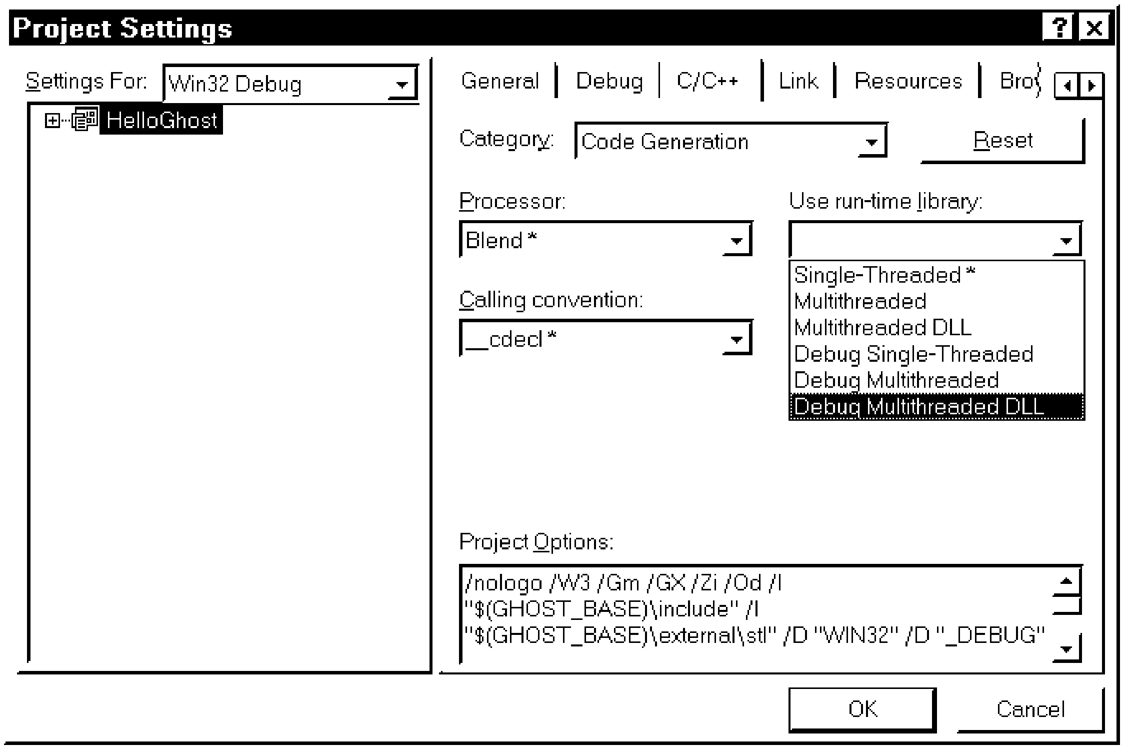
Force symbol references:

Additional library path: $(GHOST_BASE)\lib

Common Options:

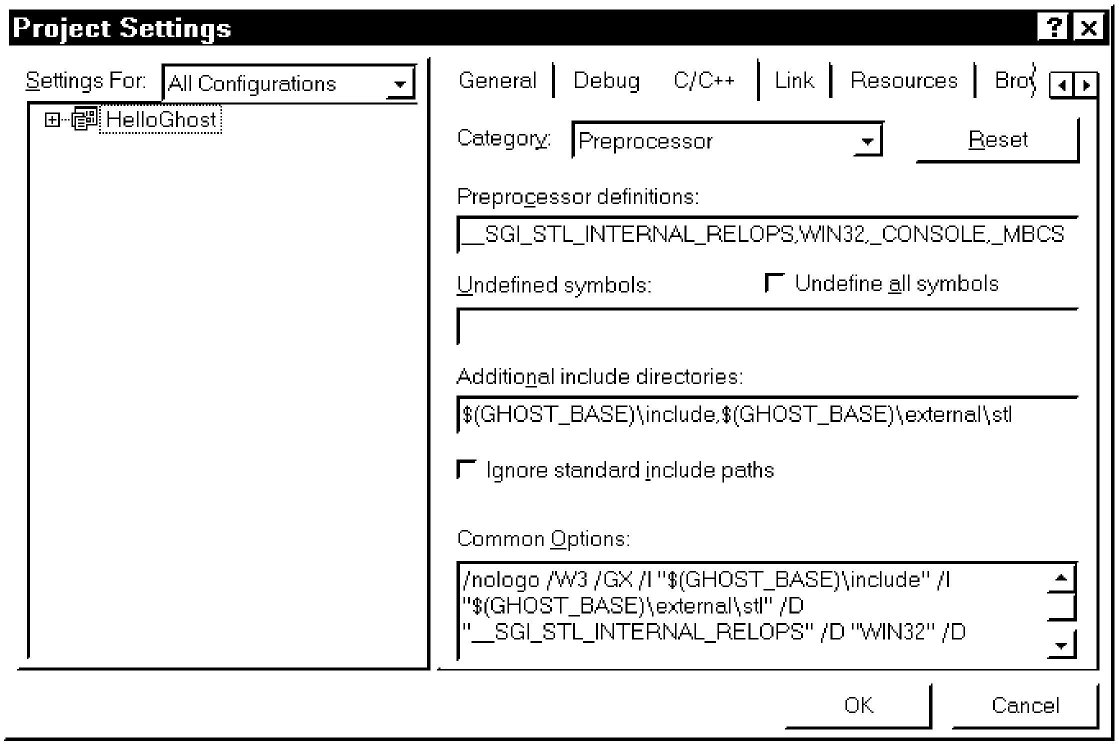
```
ghost$(GHOST_LIB_VERSION).lib /nologo
/subsystem:console /machine:I386
/libpath:"$(GHOST_BASE)\lib"
```

In addition to setting the include path you must also set the correct library path. To do so select "Settings..." from the "Project" menu. Select the "Link" tab. Select "All Configurations" in the "Settings For:" pop-up menu. In the "Category" pop-up menu select "Input". Add the following to the "Additional library path:" field: "$(GHOST_BASE)\lib". If you wish to use the GLUT library in your program also add "$(GHOST_BASE)\external\glut". In the "Object/library modules:" field enter ghost$(GHOST_LIB_VERSION).lib. This will make the program link with the appropriate version of the GHOST library. You may also enter the names of any of the optional libraries here such as ghostGLUTManager.lib, glut32.lib, ghostGLManager.lib, HapticView.lib, and gstVRML$(GSTVRML_LIB_VERSION).lib. Note that the GHOST and gstVRML libraries use the system environment variables GHOST_LIB_VERSION and GSTVRML_LIB_VERSION which are set to the current versions of these libraries during installation.



```
/nologo /W3 /Gm /GX /Zi /Od /I
"$(GHOST_BASE)\include" /I
"$(GHOST_BASE)\external\stl" /D "WIN32" /D "_DEBUG"
```

Finally you must use one of multithreaded DLL versions of the C run-time library. To do this select "Settings..." from the "Project" menu. Select the "C/C++" tab. In the "Category" pop-up menu select "Code Generation". Select "Debug" in the "Settings For:" pop-up menu. In the "Use run-time library" pop-up menu select "Debug Multithreaded DLL". Select "Release" in the "Settings For:" pop-up menu. In the "Use run-time library" pop-up menu select "Multithreaded DLL". This will configure debug builds of your program to use the debug version of the C run-time library and release builds of your program to use the release version of the C run-time library.



Preprocessor definitions:

```
__SGI_STL_INTERNAL_RELOPS,WIN32,_CONSOLE,_MBCS
```

Undefined symbols: ☐ Undefine all symbols

Additional include directories:

```
$(GHOST_BASE)\include,$(GHOST_BASE)\external\stl
```

☐ Ignore standard include paths

Common Options:

```
/nologo /W3 /GX /I "$(GHOST_BASE)\include" /I
"$(GHOST_BASE)\external\stl" /D
"__SGI_STL_INTERNAL_RELOPS" /D "WIN32" /D
```

Using the STL will cause the compiler to give a number of warnings about standard C++ relative operators such as:

```
warning C4666: 'bool __stdcall operator !=(const class CString &,const
class CString &)' : function differs from 'bool __cdecl operator
!=(const  &,const  &)' only by calling convention
```

You can avoid this by defining the symbol “__SGI_STL_INTERNAL_RELOPS”. To do this select “Settings...” from the “Project” menu. Select the “C/C++” tab. Select “All Configurations” in the “Settings For:” pop-up menu. In the “Category” pop-up menu select “Preprocessor”. In the “Preprocessor definitions” field add the symbol “__SGI_STL_INTERNAL_RELOPS” to the list of defined symbols.

You may also get the following warning when compiling GHOST SDK programs in Visual C++:

```
LINK : warning LNK4098: defaultlib "MSVCRT" conflicts with use of other
libs; use /NODEFAULTLIB:library
```

You may safely ignore this warning.

Appendix B: PHANTOM Configurations

“PHANTOM name” argument to gstPHANToM and gstPHANToMDynamic

The PHANTOM™ configuration data is used by the **gstPHANToM** node to obtain system configuration information. This data is automatically generated by the PHANTOM Configuration utility described in the PHANTOM Device Drivers Installation Guide. The PHANTOM Configuration utility allows the user to create as many as 20 uniquely named PHANTOM configurations. Each named PHANTOM configuration can refer to a particular system configuration that is used frequently. An application uses a particular PHANTOM configuration by passing the PHANTOM’s “name” into the **gstPHANToM** constructor. The default name for the configuration used by the *GHOST SDK* examples is **“Default PHANTOM”**. For a program using the *GHOST SDK* library to work properly, a **“Default PHANTOM”** configuration must have been created with the PHANTOM Configuration utility. This is the default configuration created by the utility, although others can be created as well.

On Windows, the data maintained by the PHANTOM Configuration utility is stored in the Windows NT Registry. If necessary, these registry entries may be edited manually for custom applications.

The registry entries are found in:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PHANTOM_IO\Devices\PHANTOM_IO_Device\#

where # is a number (starting at 0) representing a PHANTOM Configuration.

Many of the entries are self-explanatory. The following are some of the pertinent entries:

The numbers **l1** and **l2** refer to the lengths of the horizontal and vertical arms linking the end-point to the PHANTOM mount. For a model 1.0 PHANTOM device, **l1 = l2 = 5500**. For a model 1.5 PHANTOM device, **l1 = l2 = 8250**. For a model 3.0 PHANTOM device and a 6DOF PHANTOM device, **l1 = l2 = 18000**. For the PHANTOM Desktop device, **l1 = l2 = 5250**.

For PHANTOM Premium-style devices, the **TimeoutCircuit** is 1 if the power amplifier box has its watchdog safety circuitry enabled; otherwise, it’s 0. If a power amplifier box has a silver back-plate, it has watchdog safety circuitry, and this circuitry is enabled when shipped from STI. If the box has a black back-plate, it has no watchdog safety circuitry installed. If your power amplifier box was built before June 1996, it’s safe to assume that you *do not* have a time-out enable. For PHANTOM Desktop devices this value should be 0.

The **ChannelSelect** is set to different values depending on the system setup:
- For situations where a single PHANTOM Premium device is driven off an ISA, PCI or VME interface, the **ChannelSelect** should be set to 0 if the PHANTOM device is used with no encoder gimbal, and 2 if the PHANTOM device is used with an encoder gimbal.
- For situations where a single PHANTOM Desktop device is used, the **ChannelSelect** should always be set to 2.
- For situations where two PHANTOM Premium devices without encoder gimbals are driven off of a 6 channel ISA or a PCI interface, a special 6-channel ribbon cable is shipped to connect one card with two amplifier boxes. The PHANTOM device connected to the *first* amplifier box should have its **ChannelSelect** set to 0. The PHANTOM device connected to the *second* amplifier box should have its **ChannelSelect** set to 1.
- For situations where two PHANTOM Desktop devices are daisy-chained, the first device in the chain (the one that plugs into the computer’s parallel port) should have a **ChannelSelect** value of 0, and the

second device (the one that plugs into the first PHANTOM Desktop device) should have a **ChannelSelect** value of 1.

The PHANTOM Configuration utility arranges for these fields to have the correct values when two devices are set up in a Dual Configuration, and placed on the same interface card or parallel port.

The **r1** through **r6** values refer to the reduction ratios on each axis. For a model 1.0 PHANTOM device, **r1 = 0.11507** and **r2 = 0.13456**. For a model 1.5 PHANTOM device, **r1 = 0.076912** and **r2 = 0.090066**. For a PHANTOM 3.0 device, **r1 = 0.07555**, and **r2 = 0.086949**. For the PHANTOM Desktop device, **r1 = 0.050633**, **r2 = 0.065488**, **r3 = 0.081452**, and **r4 = 0.847458**. For the 6DOF PHANTOM device, **r1 = 0.07555**, **r2 = 0.086949**, **r3 = 0.086949**, **r4 = 0.052016**, **r5 = 0.052016**, and **r6 = 0.052016**. For PHANTOM Premium A and T models, only **r1** and **r2** need to be supplied. For PHANTOM Desktop devices, **r1** through **r4** need to be supplied. For the 6DOF PHANTOM device, all six values must be supplied.

The **Ttm1** through **Ttm6** values are the ratios for digital torque values to motor output torque at the thimble. For the PHANTOM 1.0 and 1.5 models, **Ttm1 = 114293.0**. For the PHANTOM 3.0 model, **Ttm1 = 23202.0**. For the PHANTOM Desktop device, **Ttm1 = 340000.0**. For the PHANTOM 6DOF device, **Ttm1 = 23202.0, Ttm2 = 23202.0, Ttm3 = 23202.0, Ttm4 = 452914.0, Ttm5 = 452914.0,** and **Ttm6 = 452914.0**. For all products except the 6DOF device, only one **Ttm** number needs to be provided; for the PHANTOM Premium 6DOF device, all six values must be supplied.

The **cpr1** through **cpr6** values are motor angular sensor (encoder) counts per revolution. The **cpr1** refers to the three basic PHANTOM motor encoders. The **cpr2** is used for the orientation sensor (gimbal encoder) values, if such a system is installed on your PHANTOM device. For older, low resolution PHANTOM 1.0 and 1.5 haptic interface devices, **cpr1 = 2000**, and for newer, high resolution models, **cpr1 = 4000**. For the PHANTOM 3.0 device, **cpr1 = 10160**, and for PHANTOM 3.0L devices, **cpr1 = 10000**. For older, low resolution encoder gimbals, **cpr2 = 512**, and for newer, high resolution models, **cpr2 = 2000**. For the PHANTOM Desktop device, **cpr1 = 4096**, and **cpr2 = 4468**. For the 6DOF PHANTOM device, **cpr1 = 10160, cpr2 = 10160, cpr3 = 10160, cpr4 = 1440, cpr5 = 1440,** and **cpr2 = 1440**. For all products except the 6DOF device, only **cpr1** and **cpr2** need to be provided; for the PHANTOM Premium 6DOF device, all six values must be supplied.

The **MotorType1** through **MotorType6** values are the type of motor on the PHANTOM device. For 1.0 and 1.5 models, **MotorType1** should be set to 20, and for the PHANTOM 3.0 device, **MotorType1** should be 90. For the PHANTOM Desktop device, **MotorType1** should be set to 7. For the 6DOF PHANTOM device, the first three **MotorType** values should be set to 90 and the next three set to 45. For all products except the 6DOF device, only one motor type needs to be provided; for the PHANTOM Premium 6DOF device, all six values must be supplied.

The **MaxStiffness** is the maximum stable stiffness in N/mm of the PHANTOM model, given an update rate of 1KHz. This value is an approximation since the maximum stable stiffness depends on the surface model used. For PHANTOM 1.0 and 1.5 models, we suggest a value of 0.6. For the PHANTOM 3.0 device and the 6DOF PHANTOM device, we suggest a value of 0.3. For the PHANTOM Desktop device, a value of 1.0 is suggested.

The **ResetAngles** refer to a relative offset of the PHANTOM device end-point, specified in angular (radian) terms, from the PHANTOM device reset position to the PHANTOM device sensor zero position. For most installations, sensor readings *should be* zeroed at the position where the **gstPHANToM** and **gstPHANToMDynamic** constructors are called with their second argument set to TRUE; thus the **reset angle** entries should be 0, 0, 0, 0, 0, 0 (the defaults if these values are not specified). For the PHANTOM Desktop device, the entries should be 0, 0, 0, 0.261799, 0.261799, 0.261799. If the physical reset arm of the PHANTOM 3.0 device (*only* in the A configuration) is used, the **ResetAngle** entries should be 0, 1.0386262, 0.0610865.

At the bottom of the file are three rows of workspace dimension data. The first set of numbers indicates the range of motion on each axis. The second set provides the boundary description for a workspace box. All corners of the box will be touchable by the configured phantom device. The third set of numbers describe a cube workspace boundary, where all sides have equal length. The data for these workspace dimensions was obtained by actually measuring values using the PHANTOM devices. Therefore, the (0, 0, 0) position within the workspace corresponds with the reset position of the particular PHANTOM. The last value in the set, known as the **TableTopOffset**, gives you a rough measure of the y offset from the reset position to the tabletop. This value will only be valid if you PHANTOM is sitting directly on the tabletop.

Appendix C: *GHOST SDK* Real-Time Issues

Real-Time Requirements

The *GHOST SDK* touch-enables a haptic scene by generating a separate process to run the haptic simulation (servo loop) at a reliable periodic rate. This simulation is similar to graphic update loops, however, graphic updates need to be performed at a rate of approximately 30Hz and the haptic interface must be updated on the order of 1 kHz. While fluctuations and delays in graphic updates may create temporary visual glitches, haptics instabilities may cause jolting and buzzing. The *GHOST SDK*, therefore, *must* exist as a separate process that is given high priority in order to eliminate any haptic instabilities due to update rate. Specifically, the *GHOST SDK* servo loop process generated by calling **gstScene:startServoLoop** is automatically created as a real-time process.

At a periodic rate of 1kHz, the **servoLoop** function has, at most, 1 millisecond to finish operations before the next **servoLoop** begins. The application and computer operating system, however, need time to perform their tasks. If no processing time is given to these processes, the screen could freeze or the system might crash. In order to prevent these types of situations, the *GHOST SDK* imposes a limit on the amount of processor time that may be used by a single iteration of the servoLoop. The *GHOST SDK* process will exit and **gstErrorHandler** will receive a **GST_DUTY_CYCLE_ERROR** if this limit is exceeded.

If the haptic duty cycle limit is reached, it is an indication that the haptic scene is too complex for the computer system on which it is being executed. A variety of factors contribute to scene complexity. First of all, scene complexity is directly proportional to the number of objects in the scene graph. Large **gstTriPolyMeshHaptic** nodes particularly impact scene complexity. In addition, a large number of **gstDynamic** objects updating simultaneously in the scene add significantly to the haptics load. Finally, overlapping **gstTriPolyMeshHaptic** nodes also can contribute significantly to processor load when the **gstPHANToM** is contacting the corner or edge between multiple **gstTriPolyMeshHaptic** nodes.

Haptics Processor Load (HLOAD)

The *GHOST SDK* includes the Haptics Processor Load (**hload)** program to let you monitor the load which haptics processes are placing on a computer system. This application resembles the process-load viewer available on UNIX and Windows NT machines, except that it shows *only* the duty cycle times of the *GHOST SDK* processes. A running bar graph is updated once per second to reflect a duty cycle time between zero and one millisecond. The bottom area of the window signifies zero, and the top is one millisecond. The horizontal red line indicates the threshold at which the **servoLoop** exits. The **hload** program provides an accurate indication of the relationship between the scene complexity and the *GHOST SDK* duty cycle for a given machine.

Information Sharing

Another real-time issue arises as a result of scene objects and information being shared between the application and the *GHOST SDK* **servoLoop** process. Since these two processes access the same information, one process can access memory simultaneously being modified by the second process. The *GHOST SDK* has been designed to handle these issues, but you *should* work within the constraints of the following guidelines:

> Multiple, successive calls to access the state of a scene node are *not* guaranteed to occur within the same iteration of the simulation loop. For example, calling **gstDynamic::getVelocity** and **gstDynamic::getAcceleration** consecutively *does not* guarantee that the velocity and acceleration values are returned from the same simulation loop. This *does not* generally cause difficulties because values normally not change significantly enough to create a noticeable difference in operations. If, however, you want to be assured that multiple state variables accessed from a node come from the

same loop iteration, you can use a graphics callback as described in Chapter 2 of this guide. Graphic callbacks automatically copy state information from the same iteration of the servo loop to a safe data structure.

In the current release, no data accessed from a node in the scene are guaranteed to be retrievable while the servo loop is active for a multiprocessor machine. All class methods described in the *GHOST API Reference Guide*, however, are atomic for uniprocessors. This means that any data retrieved from a scene node through a method is guaranteed to be current since the last simulation loop. Callback mechanisms should be used by the application on mutiprocessor machines to get information concerning the haptic scene

Appendix D: Autocalibration for the PHANTOM Desktop

The PHANTOM Desktop has an autocalibration feature. This allows the device to be calibrated without direct intervention by the user. With other PHANTOM models the user must hold the device in a specific position while it is calibrated. By default the calibration for the PHANTOM Desktop is updated whenever the user exits a GHOST SDK application. This feature of GHOST SDK is transparent to both the user and the application programmer. This requires the user to run an application and then exit in order to calibrate the device. In version 3.0 and later GHOST SDK allows you to update the calibration while the application is running.

The GHOST SDK supports two methods for maintaining the calibration of the PHANTOM Desktop. By using these methods, you can update the calibration at runtime as opposed to forcing the user to quit and restart the application before making use of the newly determined autocalibration values.

The two aforementioned methods are getCalibrationStatus() and updateCalibration(), both of which belong to the gstPHANToM instance. Maintaining PHANTOM calibration at runtime requires that your application periodically poll the getCalibrationStatus routine to determine if any of the axes need to be updated. If it is determined that an axis needs updating, then you should momentarily disable the servo loop and make a call to updateCalibration.

Determining calibration status

int gstPHANToM::getCalibrationStatus()

This routine returns an integer containing bitwise status flags that indicate whether valid data has been received for a particular axis and whether an update needs to be performed. On application startup, the function will return a status of 0, meaning that none of the axes have collected valid calibration data and none of them require an update. As you begin to move the PHANTOM device around, the status byte will change. Here is a table illustrating the states of interest. Note that axis 0, 1 and 2 generally correspond to the x, y and z axes respectively.

State Label	**Hex status**
Axis 0 has valid data	0x01
Axis 1 has valid data	0x02
Axis 2 has valid data	0x04
Axis 0 requires an update	0x11
Axis 1 requires an update	0x22
Axis 2 requires an update	0x44
Calibration is okay	0x0F

The lower half of the byte (the right-most value) represents the validity of data for each axis. If both axis 0 and 1 have valid data for example, then the lower hex value will be 0x03 (i.e. 0x01 | 0x02). If all axes have valid data, then the value will be 0x0F. The upper half of the byte indicates whether the calibration needs to be updated. These bits will never be active without the corresponding lower byte being active too. This means that you can not require an update for an axis without first having valid data.

To test for the update status on a particular axis, simply perform an AND operation of the desired state label with the status integer.

```
if ((status & 0x11) == 0x11)
       axisCount++;
```

Performing a calibration update

void gstPHANToM::updateCalibration()

Once you’ve determined that one or more axes need to be updated, you can use the updateCalibration() routine to introduce the new calibration values. There are some precautions though that you should consider before updating the calibration at runtime.

- Ensure that the PHANTOM device is in a safe state to change the calibration. For instance, you don’t want your user to be in the middle of some haptic task and then force the PHANTOM position to change without warning.
- You must disable the servoloop before calling updateCalibration(). Otherwise, it is likely that a max velocity error will occur, since GHOST SDK thinks that the PHANTOM position has moved a considerable distance in the span of one timestep. It is safe to simply wrapp the call to updateCalibration with a stopServoLoop before and a startServoLoop afterwards.

For an example of how to use these features see the source code for the Dice application in the examples directory of the distribution.

Appendix E: *GHOST SDK* Error Codes

The following table lists the *GHOST SDK* error codes:

Value	Symbolic Constant	Explanation
-101	GST_OUT_OF_RANGE_ERROR	Value out of range.
-102	GST_INVALID_SCALE	Attempt to apply a non-uniform scale to a gstSeparator node.
-103	GST_ALLOC_MEMORY_ERROR	Unable to allocate memory.
-104	GST_CHILD_NOT_FOUND	Unable to find child of gstSeparator node in scene graph.
-105	GST_BOUNDS_ERROR	Value beyond bounds (of an array).
-106	GST_MISC_ERROR	Miscellaneous GHOST error.
-107	GST_NESTED_DYNAMICS_ERROR	Attempt to nest dynamic nodes (make one dynamic node the ancestor of another in the scene graph).
-108	GST_PHANTOM_INIT_ERROR	Unable to initialize the PHANTOM. There is a problem with the configuration file or with the PHANTOM device driver.
-109	GST_ALGEBRAIC_SURFACE_MAX_ITER	Internal to GHOST.
-110	GST_PHANTOM_MAX_COLLISIONS	Exceeded maximum number of simultaneous PHANTOM-object collisions.
-111	GST_TIMER_ERROR	Error setting up servo-loop timer.
-112	GST_INVALID_TRANSFORM	Attempt to perform operation on an invalid transformation matrix.
-113	GST_DUTY_CYCLE_ERROR	Haptics servo loop is taking too long. The scene is too complicated to be rendered in real-time on your machine.
-114	GST_PHANTOM_FORCE_ERROR	The maximum force of the PHANTOM was exceeded, the motor amplifiers are not enabled, or the motors have overheated. If the PHANTOM motors are warm you should let them cool down.
-115	GST_PHANTOM_ERROR	PHANTOM error from the I/O Library.
-116	GST_TRANSFORM_RESET_ERROR	User defined transform matrix reset (data lost) due to call to translate, setTranslation, …