# OPENHAPTICS™ TOOLKIT

version 2.0

## PROGRAMMER'S GUIDE

# Open Haptics™ Toolkit
version 2.0

## Copyright Notice



## Trademarks



## Warranties and Disclaimers

SensAble Technologies does not warrant that this publication is error free. This publication could include technical or typographical errors or other inaccuracies. SensAble™ may make changes to the product described in this publication or to this publication at any time, without notice.

## Questions or Comments

If you have any questions for our technical support staff, please contact us at support@sensable.com. You can also phone 1-888-SENSABL (U.S.A. only) or 1-781-937-8315 (International).

If you have any questions or comments about the documentation, please contact us at documentation@sensable.com.

## Corporate Headquarters

SensAble Technologies, Inc.
15 Constitution Way
Woburn, MA 01801
Phone: 1-888-SENSABL (U.S.A. only)
E-mail: support@sensable.com
Internet: http://www.sensable.com

Last Updated - 27 July, 2005 6:19 pm P/N - 02523 R4

# Preface

This guide explains the SensAble OpenHaptics™ toolkit. This document will introduce you to the architecture of the toolkit, how it works, and what you can do with it. The guide will also introduce you to the fundamental components of creating haptic environments. and walk you through installing the toolkit and deploying your haptically enabled application.

## What is Haptics?

Haptics is the science of incorporating the sense of touch and control into computer applications through force (kinesthetic) or tactile feedback. By using special input/output devices—called haptic devices—with a haptically enabled application, users can feel and manipulate virtual three-dimensional objects. The type of feedback used to convey the sense of touch is determined by the type of haptic device being used.

Application areas for haptics are varied and continually expanding. These include:

- Surgical simulation and medical training
- Painting, sculpting, and CAD
- Military applications such as aerospace and military training and simulation
- Assistive technology for the blind or visually impaired
- Simple interaction techniques with the standard user interface such as opening/closing windows, and interacting with menus
- Gaming

### Audience

This guide assumes that the reader has an intermediate to advanced background in software development, is familiar with the C programming language, and is somewhat familiar with 3D graphics programming. Although the core API is C based, some of the utility libraries and the source code examples use C++. For additional information about haptics, see the SensAble Developer Support Center.

## Resources for Learning OpenHaptics

Sensable provides the following documentation and other materials for learning OpenHaptics:

**OpenHaptics Installation Guide** This guide walks you through installing the toolkit and deploying your haptically enabled application. Detailed instructions for installing the PHANTOM® haptic device can be found in the *PHANTOM Device User's Guide* that came with your device. This can also be found on the OpenHaptics CD.

**OpenHaptics Programmer's Guide** This guide explains the OpenHaptics™ toolkit. and introduces you to the architecture of the toolkit, how it works, and what you can do with it. The guide will also introduce you to the fundamental components of creating haptic environments.

**OpenHaptics API Reference** This manual is meant to be used as a companion to the *OpenHaptics toolkit Programmer's Guide*. It contains reference pages to all the OpenHaptics HDAPI and HLAPI functions and types as well as appendices with tables that describe all the parameters.

**Source Code Examples** Several examples with source code to illustrate commonly used functionality of the HDAPI and HLAPI are installed with the toolkit. These include both console examples and graphics examples. A guide to these examples is located in <*OpenHaptics Install directory*>/doc.

**Developer Support Center** The Developer Support Center is described in more detail below.

## The Developer Support Center

A more recent version of this document may be available for download from the SensAble online Developer Support Center (DSC). To access the DSC, visit the SensAble Support page at http://www.sensable.com/support/.

The DSC provides customers with 24 x 7 access to the most current information and forums for the OpenHaptics and GHOST® SDKs. Please note that you will be asked to create a registration profile and have your customer information authenticated before you will have access to the DSC.

## Typographical Conventions

This guide uses the following typographical conventions:

| Convention | Description | Example |
|---|---|---|
| *Italics* | Reference to another document or file; first use of a new term. | See the *Programmer's Guide.* |
| `Courier` | Identifies code. | `hdBeginFrame(hHD);` |
| **Note, Warning, Important** | Calls out important additional information. | **Important** Code snippets |
| **Bold** | Embedded functions. | Capabilities are set using **hdEnable().** |
| ***<Italics>*** | Identifies a variable such as a file name or location. | *<sensable install directory>*/OpenHaptics |

**Important** Code snippets included in this document may contain soft or hard line breaks for formatting purposes.

# Contents

# 1

# Introduction

The OpenHaptics toolkit includes the Haptic Device API (HDAPI), the Haptic Library API (HLAPI), Utilities, PHANTOM Device Drivers (PDD), Source Code Examples, this Programmer's Guide and the API Reference.

The HDAPI provides low-level access to the haptic device, enables haptics programmers to render forces directly, offers control over configuring the runtime behavior of the drivers, and provides convenient utility features and debugging aids.

The HLAPI provides high-level haptic rendering and is designed to be familiar to OpenGL® API programmers. It allows significant reuse of existing OpenGL code and greatly simplifies synchronization of the haptics and graphics threads. The PHANTOM Device Drivers support all currently shipping PHANTOM devices.

## HDAPI vs. HLAPI

The HLAPI is designed for high-level haptics scene rendering. It is targeted for developers who are less familiar with haptics programming, but desire to quickly and easily add haptics to existing graphics applications.

The HDAPI is a low-level foundational layer for haptics. It is best suited for developers who are familiar with haptic paradigms and sending forces directly. This includes those interested in haptics research, telepresence, and remote manipulations. Experts can still use HLAPI and HDAPI in conjunction to take advantage of both SDKs.

The HLAPI is built on top of the HDAPI and provides a higher level control of haptics than HDAPI, at the expense of flexibility in comparison to HDAPI. The HLAPI is primarily designed for ease of use; for example, HLAPI programmers will not have to concern themselves with such lower level issues as designing force equations, handling thread safety and implementing highly efficient data structure for haptic rendering. HLAPI follows traditional graphics techniques such as those found in the OpenGL API. Adding haptics to an object is a fairly trivial process that resembles the model used to represent the object graphically. Tactile properties, such a stiffness and friction, are similarly abstracted to materials. The HLAPI also provides event handling for ease of integration into applications.

For example, when using HDAPI, creating a haptic/graphics sphere involves writing the graphics code and creating scheduler callbacks for handling the sphere forces. When using HLAPI, the process involves creating a graphics sphere then calling **hlBeginShape()** with the desired haptic shape type when drawing the graphics sphere.

## When to Use One over the Other

HDAPI requires the developer to manage direct force rendering for the haptic device whereas HLAPI handles the computations of haptic rendering based on geometric primitives, transforms, and material properties. Direct force rendering with HDAPI requires efficient force rendering / collision detection algorithms and data structures. This is due to the high frequency of force refreshes required for stable closed-loop control of the haptic device. HLAPI differs in this regard, since it shields the developer from having to implement efficient force rendering algorithms and managing the synchronization of servo loop thread-safe data structures and state. HLAPI allows the developer to command the haptic rendering pipeline from the graphics rendering loop, which makes it significantly more approachable for a developer to introduce haptic rendering to an existing graphics loop driven application.

HLAPI enables an event driven programming model, which eases the implementation of complicated haptic interactions involving events like touching geometry, button clicks, and motion. HDAPI does not offer events as part of the API. However, the OpenHaptics toolkit does offer a HapticDevice C++ utility class that offers a basic event callback infrastructure for use with HDAPI.

HLAPI only deals with the device at the Cartesian space level whereas HDAPI offers access to lower-level control spaces, like the raw encoder and motor DAC values.

## What Parts Can be Used Together?

HLAPI is built on top of HDAPI, therefore developers can leverage pieces of functionality from HDAPI to augment an HLAPI program.

HDAPI must be used to initialize and configure the haptic device handle (HHD). The HHD from HDAPI is used by the HL haptic rendering context (HHLRC) to interface with the haptic device. This allows the developer to control behaviors for the haptic device that will be realized by the haptic rendering library. For instance, **hdEnable()/hdDisable()** can be used to turn on or off capabilities of HDAPI, like force output, force ramping, and max force clamping.

HDAPI can be used to query properties and capabilities of the device, for instance: input and output DOF, the nominal max force, workspace dimensions.

HDAPI can be used to modify the rate of the servo loop. Increasing the servo loop rate has the benefits of improved stability and responsiveness of the device as well as increased nominal max stiffness of forces. However, this comes at the expense of higher CPU utilization (i.e. 2000 Hz means an update every 0.5 ms instead of 1 ms for the default 1000 Hz rate). Conversely, the servo loop rate can also be lowered to decrease the amount of CPU used for force rendering (i.e. 500 Hz means an update every 2 ms instead of 1 ms for

the default 1000 Hz rate). This has the benefit of freeing up valuable CPU cycles for other threads, but at the expense of reduced stability of the device and lower nominal max stiffness.

**Servo Loop**

The servo loop refers to the tight control loop used to calculate forces to send to the haptic device. In order to render stable haptic feedback, this loop must be executed at a consistent 1khz rate or better. In order to maintain such a high update rate, the servo loop is generally executed in a separate, high-priority thread. This thread is referred to as the servo thread.

HLAPI allows for custom effects. A custom effect is principally responsible for adding to or modifying a force to be sent to the haptic device. Since forces are computed in the servo loop thread, the user can choose to use HDAPI routines in tandem with the custom effect callback to gain access to additional information about the device, for instance, device velocity and instantaneous rate. In addition, the HDAPI scheduler can be used as a synchronization mechanism for custom effects so that the main application thread can safely modify state or data used by the effect.

Since the last **hdEndFrame()** will actually commit the force to the haptic device. Please note that it is not necessary to call **hdStartScheduler()** or **hdStopScheduler()** when using HDAPI in tandem with HLAPI. HLAPI will manage starting and stopping the scheduler when the context is created and deleted as long as a valid handle to a haptic device is provided to the context.

# 2

# Creating Haptic Environments

Haptics programming, at its most fundamental level, can be considered a form of rendering that produces forces to be displayed by a haptic device. Haptics programming can be highly interdisciplinary, combining knowledge from physics, robotics, computational geometry, numerical methods, and software engineering. Despite this, achieving compelling interactions with a haptic device is actually quite approachable. The purpose of this chapter is to introduce some of the fundamental concepts and techniques used in haptics programming and to stimulate thought about the wealth of possibilities for incorporating haptics into a virtual environment.

This chapter contains the following sections:

# Introduction to Forces

If you are altogether unfamiliar with haptics, you may be wondering how forces can be used to extend user interaction within a virtual environment. For the class of haptic devices supported by the OpenHaptics toolkit, forces are typically used to either resist or assist motion (i.e. force feedback).

There are a variety of ways to compute the forces that are displayed by the haptic device. Some of the most interesting force interactions come from considering the position of the device *end-effector* (the end of the kinematic chain of the device you hold in your hand) and its relationship to objects in a virtual environment. When zero force is being rendered, the motion of the device end-effector should feel relatively free and weightless. As the user moves the device's end-effector around the virtual environment, the haptics rendering loop commands forces at a very high rate (1000 times per second is a typical value) that impedes the end-effector from penetrating surfaces. This allows the user to effectively feel the shape of objects in a virtual environment.

The way in which forces are computed can vary to produce different effects. For example, the forces can make an object surface feel hard, soft, rough, slick, sticky, etc. Furthermore, the forces generated by the haptics rendering can be used to produce an ambient effect. For instance, inertia, and viscosity are common ways to modify the otherwise free space motion of the user in the environment. Another common use of forces in a virtual environment is to provide guidance by constraining the user's motion while the user is selecting an object or performing a manipulation.

# Force Rendering

The force vector is the unit of output for a haptic device. There are numerous ways to compute forces to generate a variety of sensations. The three main classes of forces that can be simulated are: motion dependent, time dependent, or a combination of both.

## Motion Dependent

A force that is motion dependent means that it is computed based on the motion of the haptic device. A number of examples of motion dependent force rendering follow:

**Spring:** A spring force is probably the most common force calculation used in haptics rendering because it is very versatile and simple to use. A spring force can be computed by applying *Hooke's Law* (F = kx, where k is a stiffness constant and x is a displacement vector). The spring is attached between a fixed anchor position $p_0$ and the device position $p_1$ . The fixed anchor position is usually placed on the surface of the object that the user is touching. The displacement vector x = $p_0 - p_1$ is such that the spring force is always directed towards the fixed anchor position. The force felt is called the restoring force of the spring, since the spring is trying to restore itself to its rest length, which in this case is

zero. The stiffness constant k dictates how aggressively the spring will try to restore itself to its rest length. A low stiffness constant will feel loose, whereas a high stiffness constant will feel rigid.

**Damper:** A damper is also a common metaphor used in haptics rendering. Its main utility is for reducing vibration since it opposes motion. In general, the strength of a damper is proportional to end-effector velocity. The standard damper equation is F = -bv, where b is the damping constant and v is the velocity of the end-effector. The force is always pointing in the opposite direction of motion.

**Friction:** There a number of forms of friction that can be simulated with the haptic device. These include coulombic friction, viscous friction, static friction and dynamic friction.

**Coulombic friction** The most basic form is coulombic friction, which simply opposes the direction of motion with a constant magnitude friction force. In 1-D, the coulombic friction force can be represented by the equation F=-c sgn(v); where v is the velocity of the end-effector and c is the friction constant. Typically, this is implemented using a damping expression with a high damping constant and a small constant force clamp. Coulombic friction helps to create a smooth transition when changing directions, since friction will be proportional to velocity for slow movement.

**Viscous friction** A second form of friction that is also common is viscous friction, which is very similar to coulombic friction in that the friction is also computed using a damping expression and a clamp. The difference is that the damping constant is low and the clamp value tends to be high.

**Static and dynamic friction** This form of friction is typically referred to as stick-slip friction. It gets its name because the friction model switches between no relative motion and resisted relative motion. The friction force is always opposing lateral motion along a surface, and the magnitude of the friction force is always proportional to the perpendicular (normal) force of contact.

**Inertia:** Inertia is a force associated with moving a mass. If one knows a given trajectory (for example, the solution of the equations of motion), one can easily calculate the force one would feel during that motion using Newton's Law: F=ma (force is equal to mass x acceleration).

## Time Dependency

A force that is time dependent means that it is computed as a function of time. Below are a number of examples of time dependent force rendering:

**Constant:** A constant force is a force with a fixed magnitude and direction. It is commonly used for gravity compensation such as to make the end-effector feel weightless. Conversely, it can also be used to make the end-effector feel heavier than normal.

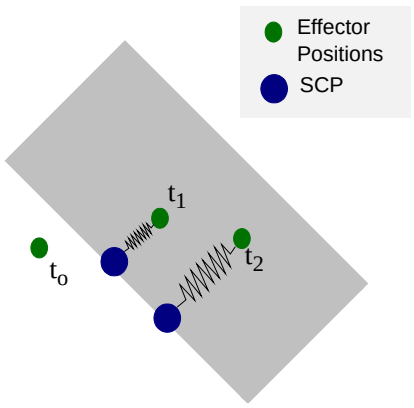
**Periodic:** A periodic force comes from applying a pattern that repeats over time. Patterns include saw tooth, square, or sinusoid. A periodic force is described by a time constant, which controls the period of the pattern's cycle, and an amplitude, which determines how strong the force will be at the peak of the cycle. Additionally, the periodic force requires a

direction. The amplitude should not exceed the maximum force the device can output or the shape of the waveform will be clipped. Additionally, the frequency of the waveform is limited by the servo loop rate of the device. For instance, the theoretical upperbound of vibration frequency is half the servo loop rate of the device.

**Impulses:** An impulse is a force vector that is instantaneously applied. In practice, an impulse with a haptic device is best applied over a small duration of time. Also, achieving believable impulses, such as the sort used for a gun recoil requires that the force transient be as sharp as possible. Humans are perceptually more sensitive to discontinuities in force than steady state force. Therefore, a larger force derivative at a lower magnitude will feel more compelling than a smaller force derivative at a higher magnitude. Note however that, trying to render a force that has too high a derivative may be ineffective due to physical limitations of the device.

## Contact and Constraints

Simulating contact with a virtual object amounts to computing forces that resist the device end-effector from penetrating the virtual object's surface. One approach to simulate this interaction is through the concept of a *proxy* that follows the transform of the device end-effector in the virtual environment.




SCP is a point that attempts to follow the end-effector position but is constrained to be on the surface of the object. In free space the SCP is at the end-effector position as shown in $t_0$. When touching an object the SCP can be calculated by moving the last SCP towards the end-effector position without violating the surface. The force is calculated by simulating a spring stretched from the end-effector position to the SCP. $t_1$ shows penetration into the object. $t_2$ shows further penetration – the spring is stretched longer and hence the user will feel greater resistance.

The geometry for the proxy is typically a point, sphere or collection of points. If it is a point, it is sometimes referred to as the *SCP* (*surface contact point*); see image for more information. When the device end-effector penetrates a surface, a transform for the proxy should be computed that achieves a minimum energy configuration between the contacted surface and the device end-effector. In addition, the proxy should respect spatial coherence of contact. For example, always reference the proxy's last transform in the process of computing its new transform. As a result of computing a constrained proxy transform, forces can then be determined that will impede the motion of the haptic device end-effector from further penetrating the contacted surface. A simple spring-damper control law can be used for computing the force. There are numerous papers in haptics rendering literature that present computational models to solve for the proxy. This technique of maintaining a constrained proxy can be applied to feeling all kinds of

geometry, such as implicit surfaces, polygonal meshes, and voxel volumes. It can also be applied to feeling geometric constraints, such as points, lines, planes, or combinations of those.

## Combining Haptics with Graphics

Typically, haptic devices are not employed in isolation. They are most often used to enhance the user experience in conjunction with a virtual 3D graphical environment. The first issue to consider when combining haptics with 3D graphics is that the refresh rate for displaying forces on the haptic device is more than an order of magnitude higher than the refresh rate necessary for displaying images on the screen. This difference stems from the psycho-physics of human perception. Typically, a graphics application will refresh the contents of the framebuffer approximately 30-60 times a second in order to give the human eye the impression of continuous motion on the screen. However, a haptic application will refresh the forces rendered by the haptic device at approximately 1000 times a second in order to give the kinesthetic sense of stiff contact. If the frame rate of a graphics application is run at a rate lower than 30 Hz, the user may perceive discontinuities in an animation such that it no longer appears visually smooth. Similarly, the user may perceive force discontinuities and a loss in fidelity when the haptic device is refreshed at a rate below 1000 Hz. As a result, haptics and graphics rendering are typically performed concurrently in separate threads so that each rendering loop can run at its respective refresh rate. Beyond the refresh rate of the two rendering loops, it's also important to consider the time duration of each rendered frame. For graphics rendering, maintaining a 30 Hz refresh rate requires that all rendering take place within $1/30^{th}$ of second (i.e. 33 ms). Contrast this with the 1 ms frame duration of the haptics rendering loop, and it becomes apparent that there's a significant disparity in the amount of time available to perform haptics rendering computations versus graphic rendering computations. In both cases, the loops are generating frames, except the haptics loop is generating ~33 frames every time the graphics loop generates one. This is a very important point to keep in mind, especially in an application where more than one object is moving on the screen or being felt by the haptic simulation at the same time.

The HLAPI allows you to specify geometry for haptic rendering in the same thread and at the same rate as graphics. It takes care of the 1000hz haptics updates for you so that you do not have to implement any haptic rendering in the 1000hz haptics thread or implement state synchronization between haptics and graphics threads.

## State Synchronization

State synchronization becomes important when managing a user interface that involves both haptics and graphics because there are two rendering loops that need access to the same information. This typically involves making thread-safe copies of data available to each thread as a snapshot of state. One might be inclined to just use mutual exclusion as a synchronization technique, but that can easily introduce problems. Most importantly, the haptics rendering loop must maintain a 1000 Hz refresh rate and thus should never wait on another lower priority thread to release a lock, especially when that lower priority thread is making other system calls that may block for an indeterminate amount of time. Secondly, the disparity in refresh rate between the graphics loop and haptics loop makes it very easy to display inconsistent state if multiple objects are moving on the screen at the same time. Therefore, it is advised to always treat state synchronization between the threads as a snapshot copy operation versus accessing data in a disjointed fashion using a mutex.

## Event Handling

In addition to state synchronization, event handling is the other common interface between haptics and graphics loops that needs special consideration. Event handling, in the context of a haptic device, typically involves responding to button presses and haptic specific events, such as touching or popping through a surface. However, the event handler must often times respond to the event in a dual-pronged fashion where the event is first handled in the haptics thread, so as to provide an immediate haptic response, then queued and handled by the graphics thread to affect the content displayed on the screen or the application state. The important point is that haptic response to an event often needs to be immediate, whereas the visual response can at least wait until the next graphics frame. One example is when applying a constraint.

If a constraint is, for example, actuated by a button press, it needs to be enabled immediately when that button press is detected in the haptics loop. Otherwise, the user will feel the constraint only after a delay.

## Combining Haptics with Dynamics

A haptic device is a natural interface for a dynamic simulation because it allows the user to provide both input to the simulation in the form of forces, positions, velocity, etc. as well as receive force output from the simulation. There are a number of powerful dynamic toolkits available that have been successfully used with haptic devices. Using a dynamic simulation with a haptic device requires special treatment, however. First a dynamic simulation works by integrating forces applied to bodies. When dealing with a position controlled impedance style haptic device, such as the kind currently supported by the OpenHaptics toolkit, forces are not directly available as input. Additionally, the mechanical properties and digital nature of the haptic device make it challenging to directly incorporate as part of the simulation.

Combining a haptic device with a dynamic simulation tends to be much more approachable and stable if a *virtual coupling* technique is used. Virtual coupling introduces a layer of indirection between the mechanical device and the simulation. This indirection is most readily accomplished using a spring-damper between a simulated body and the device end-effector. The spring-damper provides a stable mechanism for the haptic device and the simulated body to exchange forces. Optionally, the spring-damper can use different constants for computing the force for the device versus the force for the simulated body, which allows for easier tuning of forces appropriate for the device versus forces appropriate for the simulation.

There is an additional issue with integrating a dynamic simulation with a haptic device that often needs to be addressed, which is *update rate* (or step size). Only simple dynamic simulations will be able to run at the haptic device's servo loop rate (for example, ~1000 Hz). Typically, the dynamic simulation is only optimized to run at the same rate as the graphics loop (~30 Hz), or rarely faster (~100 Hz). This means that the simulation will need to be stepped in a separate thread, and there needs to be a synchronization mechanism to update the positional inputs used by the virtual coupling. Each thread will deal with a sampling of the respective spring-damper positions. The spring-damper used by the haptic device will be attached to an anchor that updates its position every time the dynamic simulation is stepped. Similarly, the dynamic simulation will sample the device position before each simulation step so that it can compute an input force to apply to the simulated body. In some instances, it may also be necessary to introduce interpolation or extrapolation of the exchanged positions to provide for more fluid forces, otherwise the user may experience a drag sensation or jerky motion.

# Haptic UI Conventions

There are a variety of ways to apply haptics to create a compelling, intuitive and satisfying user experience. Below are a number of user interface conventions that are commonly applied when using haptics in a virtual environment.

- Gravity Well Selection
- View-Apparent Gravity Well Selection
- Depth Independent Manipulation
- Relative Transformations
- Coupling Visual and Haptic Cues
- Stabilize Manipulation with Motion Friction

## Gravity Well Selection

It is very common in haptics programming that the user is allowed to select a point in 3D for manipulation. The gravity well is a useful UI convention for allowing the user to more readily select 3D points with the assistance of force feedback. The gravity well is used as a way to attract the device towards a point location; sometimes referred to as a *haptic snap* or a *snap constraint*. Typically, the gravity well has some radius of influence. When the device passes within that radial distance from the gravity well, a force is applied to attract the device towards the gravity well center. The most common attraction force is a spring force, which can be simulated by applying Hooke's Law for a spring with zero rest length. For example, a simple gravity well may use F=kx, where k is the spring constant and x is the displacement vector pointing from the device position to the center of the gravity well.

## View-Apparent Gravity Well Selection

Even with full six degrees of freedom (6DOF) control over the position and orientation of a 3D cursor in the scene, it is still a challenge to quickly locate an object in 3D when viewing the scene on a 2D display. This is typically due to the lack of visual depth cues. To overcome this limitation, one can borrow from the traditional 2D mouse *ray picking* approach. Ray picking uses the view vector or perspective reference point and a point on the near plane to perform a ray intersection query with the scene. This effectively allows the user to select an object by virtue of placing the mouse cursor overtop of it in 2D. The same principle holds when applied to a 3D haptic device. Instead of having to actually locate an object directly in 3D, it is often faster and easier to merely hover overtop of the object in the view. This concept can be extended to provide a higher dimensionality gravity well. Instead of the user fishing around to find a 3D point, the haptic device can be snapped to a 3D line passing from the view through the point of interest. This is sometimes also referred to as *boreline selection*. It is simple to implement, and very effective especially when attempting to select points or handles in 3D with little or no visual depth cues.

## Depth Independent Manipulation

The concept of depth independent manipulation goes hand-in-hand with view-apparent gravity well selection. Depth independent manipulation allows the user to initiate a manipulation relative to the object's initial location in 3D. This is somewhat like having a extendable arm that can automatically offset to the depth of the object to be manipulated. The offset is preserved during the manipulation so that the object effectively moves relative to its original location, but then the offset is removed once the manipulation is complete. This is most readily implemented by applying a translation to the device coordinates so that its position at the start of the manipulation is at the object's initial position.

## Relative Transformations

Haptic devices are typically *absolute devices*, because of the mechanical grounding necessary to provide the force actuation. Therefore, the only way to accommodate non-absolute manipulation is to apply additional transformations to the device coordinates such that the device appears to be moving relative to a given position and orientation instead of its mechanically fixed base. A relative transformation is a generalization of the depth independent manipulation idea. Instead of just being a translational offset, the transform modifications are relative to the initial affine transform relationship between the device and the object to be manipulated. Imagine spearing a potato with the prongs of a fork. The fork fixes the relative transformation between your hand and the potato, making it possible to position and rotate the potato relative to its initial transform, despite the fact that your hand is some arbitrary distance away holding onto the fork. This metaphor can be applied to virtual object manipulations with a haptic device by introducing a relative transformation between the device coordinates and the object coordinates.

## Coupling Visual and Haptic Cues

A first-time user may be surprised by how much more satisfying and engaging a user interaction can be when more than one sense is involved. In the case of haptics, the sense of feeling something can be improved dramatically by providing a visual representation of the contact. The trick is to provide the correct visual. For instance, one of the most common mistakes with haptics is to haptically render contact with a rigid virtual object yet visually display the device cursor penetrating its surface. The illusion of contacting a rigid virtual object can be made significantly more believable if the cursor is never displayed violating the contact. In most cases, this is simply a matter of displaying the constrained proxy instead of the device position. Haptic cues can also be used to reinforce visual cues. For instance, it is common for selection of an object to be preceded or accompanied by highlighting of the object. An appropriate haptic cue can make that highlighting even more pronounced by providing a gravity well or a localized friction sensation.

## Stabilize Manipulation with Motion Friction

In some instances, it will be desirable to have a small amount of friction applied while performing an otherwise free space manipulation. The friction helps to stabilize the hand as the user tries to achieve a desired position. Without the friction, the device may feel too “free” or loose and it may be difficult for a user to make small or precise manipulations.

# 3

# HDAPI Overview

The Haptic Device API (HDAPI) consists of two main components: the *device* and the *scheduler*. The device abstraction allows for any supported 3D haptics mechanism (see the Installation Guide or readme for a list of supported devices) to be used with the HDAPI. The scheduler callbacks allow the programmer to enter commands that will be performed within the servo loop thread. A typical use of the HDAPI is to initialize the device, initialize the scheduler, start the scheduler, perform some haptic commands using the scheduler, then exit when done.

This chapter includes the following sections:

## Getting Started

The HDAPI requires a supported 3D haptic device with installed drivers, and the installed HDAPI. Projects should use the main HDAPI headers or utilities and link with the HDAPI library as well as any utility libraries for those being used.

The general pattern of use for the HDAPI is to initialize a device, create scheduler callbacks to define force effects, enable forces, and start the scheduler. Force effects are typically calculated based on the position of the device, one example of a force effect may query the position of the device during each scheduler tick and calculate a force based on that.

**A simple example**

Consider a simple haptic plane example. The example application creates a plane that repels the device when the device attempts to penetrate the plane. The steps in this example are:

1. Initialize the device.
2. Create a scheduler callback that queries the device position and commands a force away from the plane if the device penetrates the plane.
3. Enable device forces.
4. Start the scheduler.
5. Cleanup the device and scheduler when the application is terminated.

## The Device

Calls to the device typically involve managing state, setting parameters, and sending forces. The device interface also allows for managing multiple devices. Device routines broadly fall into a few categories:

**Device initialization** Includes everything necessary to communicate with the device. This generally involves creating a *handle* to the device, enabling forces, and calibrating the device.

**Device safety** Includes routines to handle force feedback safety checks. Examples include overforce, overvelocity, force ramping, and motor temperature parameters. Some safety mechanisms are controlled by the underlying device drivers or hardware and cannot be overwritten.

**Device state** Includes getting and setting state. Examples include querying buttons, position, velocity, and endpoint transform matrices. Forces and torques are commanded by setting state. Forces and torques can be specified in Cartesian space or with raw motor DAC values.

## The Scheduler

The scheduler manages a high frequency, high priority thread for sending forces and retrieving state information from the device. Typically, force updates need to be sent at 1000 Hz frequency in order to create compelling and stable force feedback. The scheduler interface allows the application to communicate effectively with the servo loop thread in a thread-safe manner, and add operations to be performed in the servo loop thread.

## Developing HDAPI Applications

To develop an HDAPI enabled application you will need to:

**1** Link the multi-threaded C-runtime (CRT), as shown below in “Use run-time library.”

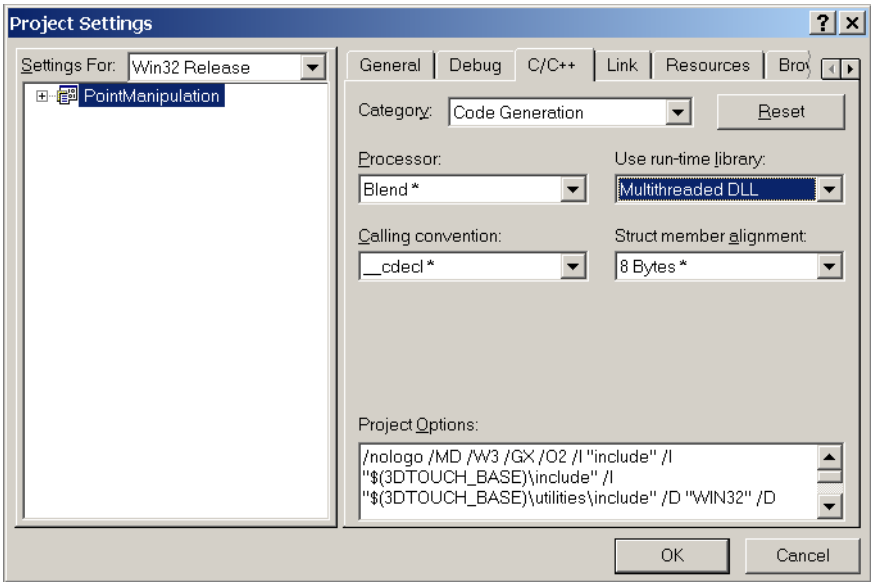



**FIGURE 3-1.** Linking the multi-threaded C-runtime

**Warning** As an SDK developer, you need to make sure that you link your application with the multi-threaded CRT. Otherwise unpredictable behavior, including crashes, can occur.

**2** Set the correct include path, as shown below in “Additional include directories.”

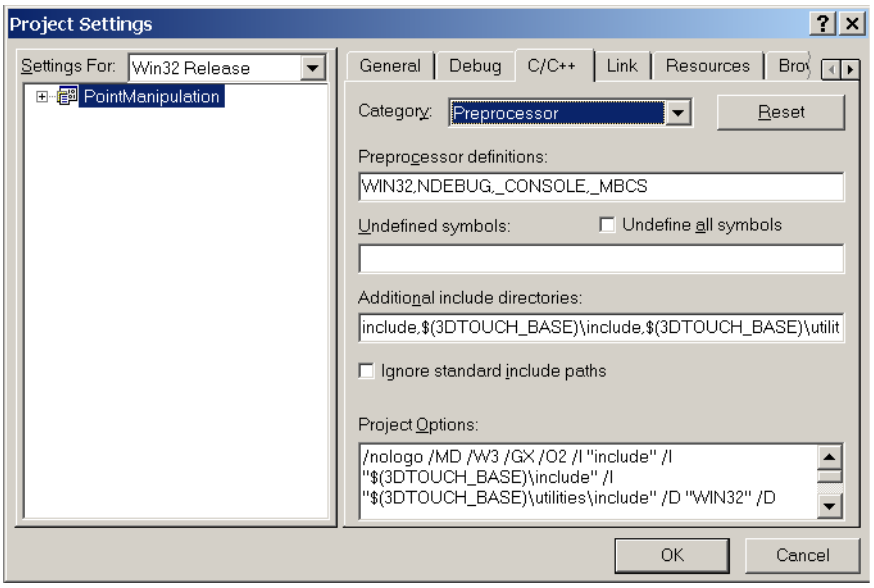



**FIGURE 3-2.** Set the include paths

Examples for setting include paths:

| | |
|---|---|
| **`$(3DTOUCH_BASE)\include`** | Main include directory for the HD library. |
| **`$(3DTOUCH_BASE)\utilities\include`** | Include directory for utilities. |

Setting the include path as indicated above will enable the developer to include header files as follows:

```
#include <HD/hd.h>
#include <HDU/hduVector.h>
#include <GL/glut.h>
```

**3** Add the appropriate library module and set the library path, as shown below in "Object/Library modules."

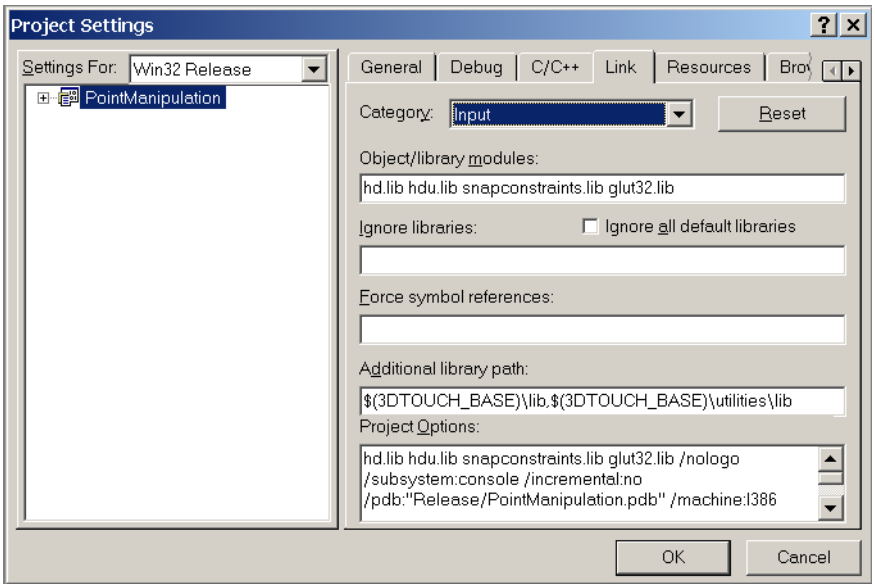


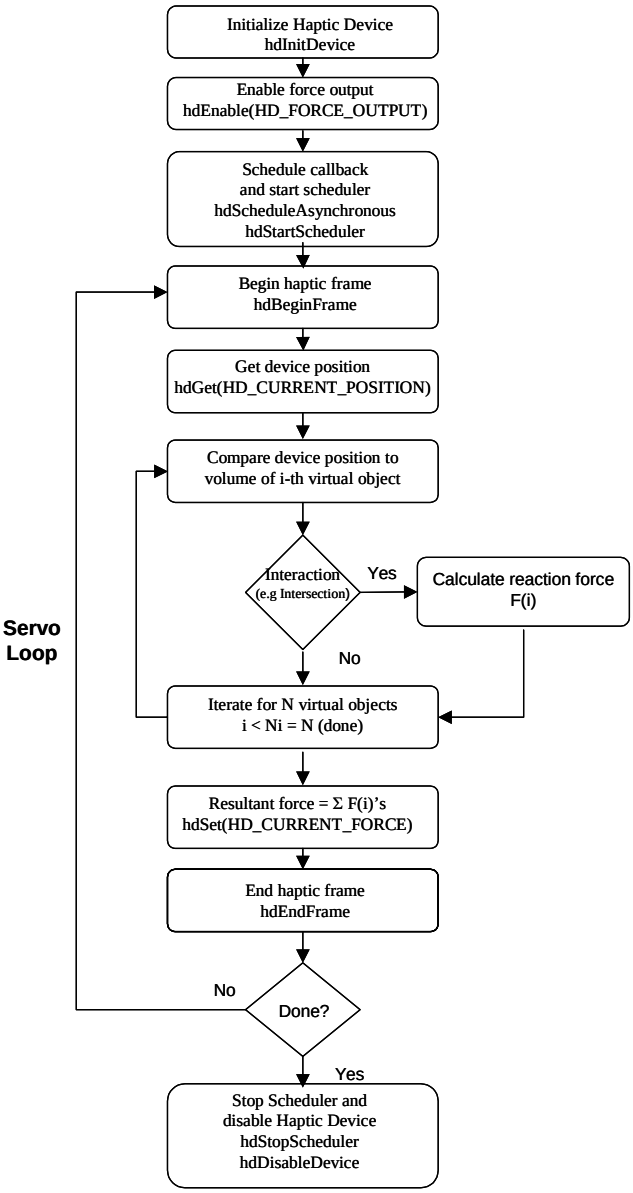

**FIGURE 3-3.** Set path to the library module

All builds should include hd.lib. Debug builds should add the debug utility library hdud.lib, and release builds should add the release library (hdu.lib).

| | |
|---|---|
| **`hd.lib`** | Release version of HDAPI |
| **`hdu.lib`** | Release version of HDU library (HD Utilities) |
| **`hdud.lib`** | Debug version of HDU library (HD Utilities) |

# Design of Typical HDAPI Program

The following diagram shows a typical flow chart of an HDAPI program for rendering virtual objects.



# 4

# HDAPI Programming

This chapter contains the following sections:

# Haptic Device Operations

Haptic device operations include all operations that are associated with getting and setting state. Device operations should exclusively be performed within the servo loop by use of the scheduler callbacks. Directly making calls to getting state, starting and ending frames, enabling and disabling capabilities, etc. within the application is not thread safe and will typically result in an error. These calls can be made safely in the application, but only when the scheduler is not running.

## Initialization

Both the device and scheduler require initialization before use. Devices are identified by their name, which is a readable string found in the “PHANToM Configurations” control panel under “PHANToM”. Typically, if there is only one device, that will be named “Default PHANToM.” The first calls in an HDAPI application then are usually for device initialization:

```
HHD hHD = hdInitDevice(HD_DEFAULT_DEVICE);
```

Devices are initialized with forces set to off for safety. The next command in initialization is to enable forces:

```
hdEnable(HD_FORCE_OUTPUT);
```

Note that the forces do not actually become enabled, however, until the scheduler is started.

```
hdStartScheduler();
```

If multiple devices are used, each needs to be initialized separately. However, there is only one scheduler so it just needs to be started once. **hdStartScheduler()** starts the scheduler and should be the last call in initialization. Asynchronous calls should be scheduled before this so that they begin executing as soon as the scheduler is turned on.

### Current Device

The HDAPI has a concept of a current device, which is the device against which all device-specific calls are made.

```
hdMakeCurrentDevice(hHD);
```

If multiple devices are used, the devices need to take turns being current in order to have operations targeted at them. For the case of a single device, **hdMakeCurrentDevice()** does not ever need to be called.

### Device Capabilities

Some device features are capabilities that can be toggled on or off. With the exception of HD_FORCE_OUTPUT, capabilities should not be changed unless the developer is well aware and intentional about the outcome, since most capabilities are related to device and user safety. Capabilities are set using **hdEnable()** and **hdDisable()**, and the current setting of a capability can be queried using **hdIsEnabled()**.

```
if (!hdIsEnabled(HD_FORCE_OUTPUT))
{
    hdEnable(HD_FORCE_OUTPUT);
}
```

As with all calls involving device state, enable and disable calls should be made from the scheduler thread through a scheduler callback.

## Haptic Frames

Haptic frames define a scope within which the device state is guaranteed to be consistent. Frames are bracketed by **hdBeginFrame()** and **hdEndFrame()** statements. At the start of the frame, the device state is updated and stored for use in that frame so that all state queries in the frame reflects a snapshot of that data. At the end of the frame, new state such as forces is written out to the device. Calls to get last information such as last position yield information from the previous frame.

Most haptics operations should be run within a frame. Calling operations within a frame ensures consistency for the data being used because state remains the same within the frame. Getting state outside a frame typically returns the state from the last frame. Setting state outside a frame typically results in an error.

Each scheduler tick should ordinarily have up to only one frame per device. Frames for different devices can be nested. The developer can disable the one frame per tick per device limit by disabling the HD_ONE_FRAME_LIMIT capability, but this is not generally recommended because some devices may not function properly when more than one frame is used per scheduler tick.

Frames can be interwoven when used with multiple devices. However, note that each call to **hdBeginFrame()** makes the associated device active.

```
HHD id1, id2;
…
hdBeginFrame(id1);
hdBeginFrame(id2);
...
hdEndFrame(id1);
hdEndFrame(id2);
```

An alternate way to manage device state is to call **hlCheckEvents()** instead of using **hlBegin/EndFrame()**. In addition to checking for events, **hlCheckEvents()** updates the device state and last information just as using hlBegin/EndFrame does. For example, consider a scene with a static sphere and a graphics cursor representation such as a typical Hello Sphere application. The traditional paradigm of managing the scene is to call **hlBegin/EndFrame()** periodically and respecify the sphere each time. This also updates the device state, so that the position of the graphics cursor is maintained. An alternative approach is to just call **hlCheckEvents()** periodically. This updates the device state so that the position of the graphics cursor can be maintained, so it saves the application from having to respecify the shape once it's created.

## Scheduler Operations

The scheduler allows for calls to be run in the scheduler thread. Since the device needs to send force updates at a very high rate—typically 1000Hz— the scheduler manages a high priority thread. If the developer needs to make queries or change state, he should do so within this loop; otherwise, since state is constantly changing, it is typically unsafe for the application to query or set state. For example, the user should not be querying data that is being changed at scheduler rates; variables should not be shared between the two threads. The user should access variables that are modified by the haptics thread only by using a scheduler callback.

A callback’s prototype is of the form:

```
HDCallbackCode HDCALLBACK DeviceStateCallback
              (void *pUserData);
```

The return value can be either:

HD_CALLBACK_DONE or HD_CALLBACK_CONTINUE.

Callbacks can be set to run either once or multiple times, depending on the callback's return value. If the return value requests for the callback to continue, it is rescheduled and run again during the next scheduler tick. Otherwise it is taken off the scheduler and considered complete, and control is returned to the calling thread in the case of synchronous operations.

```
// client data declaration
struct DeviceDisplayState
{
    HDdouble position[3];
```

```
    HDdouble force[3];
}

// usage of the above client data, within a simple callback.
HDCallbackCode HDCALLBACK DeviceStateCallback
                              (void *pUserData)
{
   DeviceDisplayState *pDisplayState =
      static_cast<DeviceDisplayState *>(pUserData);

   hdGetDoublev(HD_CURRENT_POSITION,
                pDisplayState->position);

   hdGetDoublev(HD_CURRENT_FORCE,
                pDisplayState->force);

  // execute this only once
  return HD_CALLBACK_DONE;
}
```

Scheduler calls are either of two varieties -- asynchronous and synchronous. Synchronous calls only return after they are completed, so the application thread waits for a synchronous call before continuing. Asynchronous calls return immediately after being scheduled.

**Synchronous Calls** Synchronous calls are primarily used for getting a snapshot of the state of the scheduler for the application. For example, if the application needs to query position or button state, or any other variable or state that the scheduler is changing, it should do so using a synchronous call. As an example:

```
// get the current position of end-effector
DeviceDisplayState state;
hdScheduleSynchronous(DeviceStateCallback,
                      &state,
                      HD_MIN_SCHEDULER_PRIORITY);
```

**Asynchronous Calls** Asynchronous calls are often the best mechanism for managing the haptics loop. For example, an asynchronous callback can persist in the scheduler to represent a haptics effect: during each iteration, the callback applies the effect to the device. As an example:

```
HDCallbackCode HDCALLBACK CoulombCallback(void *data)
{
    HHD hHD = hdGetCurrentDevice();
    hdBeginFrame(hHD);
    HDdouble pos[3];

    //retrieve the position of the end-effector.
    hdGetDoublev(HD_CURRENT_POSITION,pos);

    HDdouble force[3];
    // given the position, calculate a force
    forceField(pos, force);
```

```
    // set the force to the device
    hdSetDoublev(HD_CURRENT_FORCE, force);

    // flush the force
    hdEndFrame(hHD);

    // run at every servo loop tick.
    return HD_CALLBACK_CONTINUE;
}

hdScheduleAsynchronous(AForceSettingCallback,
                       (void*)0,
                       HD_DEFAULT_SCHEDULER_PRIORITY);
```

The asynchronous callback scheduling function returns a handle that can be used in the future to perform operations on the callback. These operations include unscheduling the callback—i.e. forcing it to terminate—or blocking until its completion (see **hdWaitForCompletion()** in the *Open Haptics API Reference*).

```
HDSchedulerHandle calibrationHandle =
   hdScheduleAsynchronous(aCallback,
                          (void*)0,
                          HD_MIN_SCHEDULER_PRIORITY);
hdStopScheduler();
hdUnschedule(calibrationHandle);
```

Callbacks are scheduled with a priority, which determines what order they are run in the scheduler. For every scheduler tick, each callback is always executed. The order the callbacks are executed depends on the priority; highest priority items are run before lowest. Operations with equal priority are executed in arbitrary order.

Only one scheduler thread ever runs, regardless of the number of devices. If there are multiple devices, they all share the same scheduler thread.

## State

### Get State

Device state and other information can be retrieved through the use of the **hdGet** family of functions, for example, **hdGetDoublev(), hdGetIntegerv()**. These all require a valid parameter type, and either a single return address or an array. It is the caller's responsibility to ensure that the size of the return array is as large as the number of return values for the function.

Not all functions support parameters of all argument types. If an invalid type is used, then an HD_INVALID_INPUT_TYPE error is generated. For example, HD_DEVICE_MODEL_TYPE requires a string and should only be called with **hdGetString()**.

CURRENT and LAST state refer to state that either exists in the frame in which the query was made, or the previous frame. If a call to CURRENT or LAST state is made outside a frame, then those are treated as if they were made within the previous frame; for example,. HD_CURRENT_FORCE will return the force that was set in the previous frame, and HD_LAST_FORCE will return the force that was set in the frame before that one.

Force output parameters such as HD_CURRENT_FORCE, HD_CURRENT_TORQUE and HD_CURRENT_MOTOR_DAC_VALUES will return whatever value the user set for each during the frame. The current state of these force output parameters is automatically set to zero at the beginning of each frame.

The following examples illustrate various functions to get state:

```
HDint buttonState;
HDstring vendor;
hduVector3Dd position;
HDfloat velocity[3];
HDdouble transform[16];

hdGetIntegerv(HD_CURRENT_BUTTONS,&buttonState);
hdGetString(HD_DEVICE_VENDOR,vendor);
hdGetDoublev(HD_CURRENT_POSITION,position);
hdGetFloatv(HD_CURRENT_VELOCITY,velocity);
hdGetDoublev(HD_LAST_ENDPOINT_TRANSFORM,transform);
```

Calls to getting state should generally be run within the scheduler thread and within a haptics frame.

## Set State

Setting certain parameters can change the characteristics of some safety behaviors or command forces to the device. Setting state should always be done within a begin/end frame. Mixing of Cartesian forces or torques with motor DAC values is presently not supported. For example, calling **hdSetDoublev()** on HD_CURRENT_FORCE and HD_CURRENT_MOTOR_DAC_VALUES will result in an error.

The caller is responsible for passing in the correct number of parameters. Not all parameters support all types; if an invalid type is used, then an HD_INVALID_INPUT_TYPE error is generated. See the *Open Haptics API Reference* for more information.

The following illustrates some typical uses of setting state:

```
HDdouble force[3] = {0.5, 0.0, 1.0};
hdSetDoublev(HD_CURRENT_FORCE,force);
HDfloat rampRate = .5;
hdSetFloatv(HD_FORCE_RAMPING_RATE,&rampRate);
```

Note that forces are not actually sent to the device until the end of the frame. Setting the same state twice will replace the first with the second. For example, if the developer wishes to accumulate several difference forces, he can either accumulate the resultant force in a private variable, or can use **hdGet()/hdSet()** repeatedly to accumulate the force in the HD_CURRENT_FORCE storage.

## Synchronization of State

The scheduler provides state synchronization capabilities between threads. For example, consider a custom state that needs to be updated at servo loop rates and is accessed and modified from another thread such as the graphics thread. One instance may be a dynamics simulation that updates the position of objects according to some equation of motion run in the servo loop thread. The positions will be changing frequently, and the graphics redraw functions will occasionally access those positions and use them to draw the objects in the scene. During the graphics redraw, the state needs to be consistent since the graphics thread runs at a considerably lower rate than the servo loop thread. For example, if the graphics thread were to access the instantaneous position of the object twice, even in a single short routine, it is likely that the two positions would be different.

In the example below, the position will likely have changed between the two times it is queried.

```
HDCallbackCode positionUpdateCallback(void *pUserData)
{
    hduVector3Dd *position = (hduVector3Dd *)pUserData;
    hdGetDoublev(HD_CURRENT_POSITION,*position);
    return HD_CALLBACK_CONTINUE;
}

void func()
{
   hduVector3Dd position;
   hdScheduleAsynchronous(positionUpdateCallback,
                          position,
                          HD_DEFAULT_SCHEDULER_PRIORITY);
   hduVector3Dd pos1 = position;
   hduVector3Dd pos2 = position;
   /* This assertion will likely fail. */
   assert(pos1 == pos2);
}
```

## Calibration Interface

Calibration allows the device to maintain an accurate idea of its physical position. For example, before calibration is called, a device might think that its arm is in the center of the workspace whereas the arm is actually off to one side. There are a few different methods for performing calibration:

### Types of Calibration

**Hardware reset of encoders:** In this form of calibration, the user manually places the unit into a reset position and calls the calibration routine. For the PHANTOM devices, the reset position is such that all links are orthogonal. This typically only needs to be performed once when the unit is plugged in and the calibration will persist until the unit is unplugged or another hardware reset is performed. Examples of devices that support hardware reset include all PHANTOM Premium models.

**Inkwell calibration:** In this form of calibration, the user puts the gimbal into a fixture, which constrains both its position and orientation, and calls the calibration routine. The scheduler typically needs to be running for this form of calibration to succeed. This calibration persists from session to session. Examples of inkwell calibration devices include the PHANTOM Omni.

**Auto calibration:** In auto calibration, the device internally uses mechanisms to update its calibration as the unit is moved around. This style also supports partial calibration where information about one or more axes is obtained such that calibration can be performed along those axes. This calibration persists from session to session. Examples of auto calibration devices include the PHANTOM Desktop.

### Querying Calibration

Since each device type has a different form of calibration, the type of calibration supported can be queried through getting HD_CALIBRATION_STYLE. For encoder reset and inkwell calibration, the user should be prompted to put the unit into the appropriate position and **hdUpdateCalibration()** should be called once. For auto calibration, the calibration should be periodically checked with **hdCheckCalibration()** and updated through **hdUpdateCalibration()** if the calibration check indicates that the calibration needs an update through the HD_CALIBRATION_NEEDS_UPDATE return value.

> **Note** Devices can also be calibrated through running PHANToM Test, a diagnostic utility installed with PDD and available from either Start>Sensable or within the <*SensAble install directory*>/PHANTOM Device Drivers.

### When to Calibrate

Since calibration may cause the device position to jump, calibration should ordinarily be performed with some force checking, or disabling of forces. Otherwise, for example, calibration might cause the device to jump into a position where a large force would be commanded in response (such as the inside of an object).

**Calling Calibration**

First, the user should choose a calibration style from a list of supported styles, because some devices may support multiple types of calibration.

```
HDint supportedCalibrationStyles;

hdGetIntegerv(HD_CALIBRATION_STYLE,
              &supportedCalibrationStyles);
if (supportedCalibrationStyles &
    HD_CALIBRATION_ENCODER_RESET)
{
   calibrationStyleSupported = true;
}
if (supportedCalibrationStyles & HD_CALIBRATION_INKWELL)
{
   calibrationStyleInkwellSupported = true;
}
if (supportedCalibrationStyles & HD_CALIBRATION_AUTO)
{
   calibrationStyleAutoSupported = true;
}
```

Next, define callbacks that will check the calibration as follows:

```
HDCallbackCode CalibrationStatusCallback
                    (void *pUserData)
{
    HDenum *pStatus = (HDenum *) pUserData;

    hdBeginFrame(hdGetCurrentDevice());
    *pStatus = hdCheckCalibration();
    hdEndFrame(hdGetCurrentDevice());

    return HD_CALLBACK_DONE;
}
```

An example of a callback that will update the calibration (in cases where manual input is not required, i.e. calibration style is not inkwell) is:

```
HDCallbackCode UpdateCalibrationCallback
                (void *pUserData)
{
    HDenum *calibrationStyle = (HDint *) pUserData;

    if (hdCheckCalibration() ==
                      HD_CALIBRATION_NEEDS_UPDATE)
    {
        hdUpdateCalibration(*calibrationStyle);
    }

    return HD_CALLBACK_DONE;
}
```

## Error Reporting and Handling

Generated errors are put on an error stack and thus can be retrieved in reverse order, for example, most recent first. If the error stack is empty, then asking for an error will return one that has HD_SUCCESS as its error code. Error information contains three fields.

**1** An error code, from the definitions file.

**2** An internal error code. This is the raw error code generated by the device call, it is typically used to query for additional support from the device vendor.

**3** The device ID that generated the call, this is useful for error handling debugging in a system that contains multiple devices.

Errors are not always generated by the immediate call beforehand. For example, if the user asks for an error in his main application thread, that error might have come from an asynchronous call that was running in the scheduler thread.

Errors can occur from any number of different causes. Some are a result of programmatic fault, such as attempting to call a function with improper argument type. Others are device faults such as if a device cannot be initialized. Still others may result from usage patterns such as temperature and force errors. It is not necessary to check for errors after each call; however, the error stack should be queried periodically, particularly to allow the application to catch errors that need user notification such as temperature and device initialization failures.

As an example:

```
/* Check if an error occurred while attempting to render the force */
if (HD_DEVICE_ERROR(error = hdGetError()))
{
   if (hduIsForceError(&error))
   {
      bRenderForce = FALSE;
   }
   else if (hduIsSchedulerError(&error))
   {
       return HD_CALLBACK_DONE;
   }
}
```

## Cleanup

Before application exit, the scheduler should be stopped and all scheduler operations terminated. Callbacks can be terminated either through the application calling **hdUnschedule()**, or through the callback itself knowing to return HD_CALLBACK_DONE. Finally, the device should be disabled. The following shows a typical cleanup sequence:

```
hdStopScheduler();
hdUnschedule(scheduleCallbackHandle);
hdDisableDevice(hdGetCurrentDevice());
```

# 5

# HLAPI Overview

The HLAPI is a high-level C API for haptic rendering patterned after the OpenGL API for graphic rendering. The HLAPI allows programmers to specify geometric primitives such as triangles, lines and points along with haptic material properties such as stiffness and friction. The haptic rendering engine uses this information along with data read from the haptic device to calculate the appropriate forces to send to the haptic device.

Like OpenGL, HLAPI is based on a state machine. Most HLAPI commands modify the rendering state and rendering state may also be queried using the API. State includes information such as the current haptic material settings, transformations and rendering modes. In addition to state set by a user's program, the HLAPI state includes the state of the haptic device such as its position and orientation. The API also provides the ability to set event callback functions which the rendering engine will call whenever certain events, such as touching a shape or pressing the stylus button on the haptic device, occur.

This chapter includes the following sections:



## Generating Forces

There are three ways to generate haptic feedback using the HLAPI:

- **Shape rendering** allows users to specify geometric primitives which the rendering engine uses to automatically compute the appropriate reaction force to simulate touching the surfaces defined by the geometry. The HLAPI allows users to specify geometry using OpenGL commands as well as through custom shape callbacks.
- **Effect rendering** allows users to specify global force functions which are not easily defined by geometry. While shapes only generate forces when the haptic device is in contact with the shape geometry, effects may generate forces at any haptic device position. HLAPI includes a number of standard force effects such as viscosity and springs as well as the ability to specify custom force functions.

- **Direct proxy rendering** allows the user to set a desired position and orientation for the haptic device and the rendering engine will automatically send the appropriate forces to the haptic device to move it towards the desired position.

## Leveraging OpenGL

The primary mechanism for specifying the geometry of shapes with the HLAPI is to use OpenGL commands. This allows for a broad range of ways to specify geometry as well as for reuse of OpenGL code in already existing programs.

The HLAPI is able to haptically render geometry specified using the full range of OpenGL commands for specifying geometry. This includes primitives such as points, lines and polygons specified using **glBegin()** as well as geometry stored in display lists and vertex arrays. Capturing geometry from OpenGL is done in two different ways: depth buffer shapes and feedback buffer shapes.

With *depth buffer* shapes, OpenGL rendering commands are used to render geometry to the depth buffer. The HLAPI reads the image from the depth buffer and uses it to determine the appropriate geometry to be used to generate forces for the haptic device.

With *feedback buffer* shapes, geometry rendered using OpenGL is captured in the OpenGL feedback buffer. The HLAPI then reads the geometry out of the feedback buffer and computes the appropriate forces to send to the haptic device.

A program may also set transforms using OpenGL calls such **glTranslate(), glRotate()**, and **glScale()** and the HLAPI will apply the transforms set by these commands to geometric primitives for haptics rendering. The API does this by querying portions of the transform state from the OpenGL state machine.

## Proxy Rendering

Haptic rendering of geometry is done using the proxy method. The proxy (also known as the “SCP” or “god-object”) is a point which closely follows the position of the haptic device. The position of the proxy is constrained to the outside of the surfaces of all touchable shapes. The haptic rendering engine continually updates the position of the proxy, attempting to move it to match the haptic device position, but not allowing it to move inside any shapes. While the actual position of the haptic device may be inside a shape, the proxy will always be outside. When not touching a shape, the proxy will always be placed at the device position, but when in contact with a shape the haptic device will

penetrate the surface of the shape and the proxy will remain on the outside of the surface. The force sent to the haptic device is calculated by stretching a virtual spring-damper between the haptic device position and the proxy position.

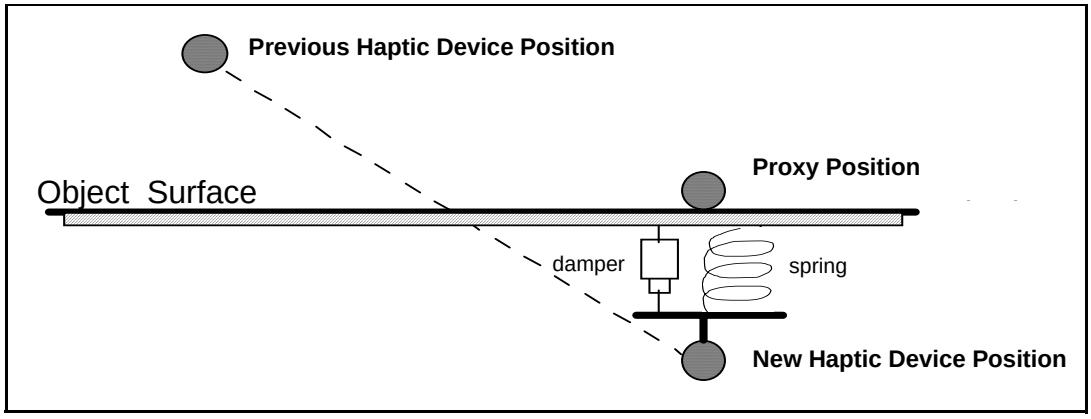



**FIGURE 5-4.** The Proxy

The HLAPI automatically maintains the appropriate proxy position for the geometry specified. Programs may query the current proxy position from API state in order to draw a 3D cursor or to know the point on a shape which the user is touching.

## Threading

Because haptic rendering requires more frequent updates than typical graphics applications, the HLAPI rendering engine creates, in addition to the main application thread, two additional threads that it uses for haptic rendering: the servo thread and the collision thread. The main application thread in a typical HLAPI program is referred to as the “client thread”. The client thread is the thread in which the HLAPI rendering context is created and in which HLAPI functions are called by client programs. Typical users of the HLAPI will write code that runs in their client thread and will not need to know about the servo or collision threads, although there are cases where advanced users will want to write code that runs in one of these threads.

**Servo Thread**

The servo thread handles direct communication with the haptic device. It reads the device position and orientation and updates the force sent to the device at a high rate (usually 1000 hz). This thread runs at an elevated priority in order to maintain stable haptic rendering. The servo thread is similar to the servo thread in a typical HDAPI program although unlike with HDAPI, HLAPI hides the servo thread from the user (with the exception of custom force effects).

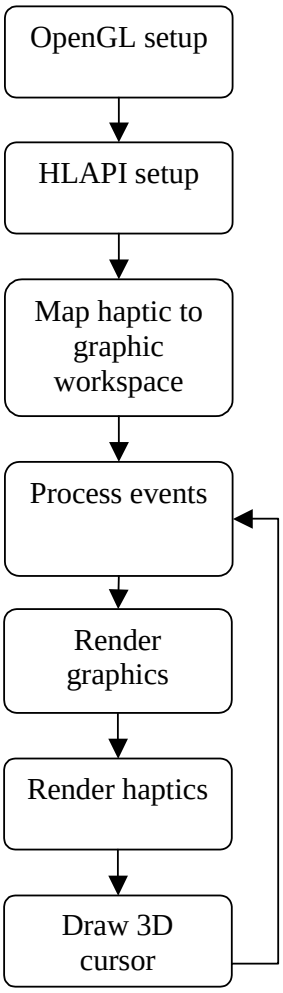
**Collision Thread**

The collision thread is responsible for determining which geometric primitives are in contact with the proxy. It runs at a rate of 100 hz which is slower than the servo thread but faster than the client thread. The collision thread finds which of the shapes specified in the client thread are in contact with the proxy and generates simple local approximations of those shapes. These local approximations are sent to the servo thread which uses them to

update the force to the haptic device. Using simple local features allows the servo thread to maintain a high update rate even if the number of geometric primitives provided from the client thread is high.

## Design of Typical HLAPI Program

A typical HLAPI program has the following structure:




First, the program sets up OpenGL by creating a graphics rendering context and tying it to a window. Then it initializes the HLAPI by creating a haptics rendering context and tying it to a haptic device. Then the program specifies how the physical coordinates of the haptic device should be mapped into the coordinate space used by the graphics. This mapping is used by the HLAPI to map geometry specified in the graphics space to the physical workspace of the haptic device. Next, the application renders the scene graphics

using OpenGL. Then the program processes any events generated by the haptics rendering engine such as contact with a shape or a click of the stylus button. Then the haptics are rendered, usually by executing nearly the same code as for rendering the graphics, but capturing the geometry as a depth or feedback buffer shape. In addition to rendering scene geometry, a 3D cursor is rendered at the proxy position reported by the HLAPI. Finally, the rendering loop continues by rendering the graphics again.

# 6

# HLAPI Programming

This chapter contains the following sections:

## Device Setup

Device setup for HLAPI is similar to HDAPI. The first step is to initialize the device by name using **hdInitDevice()**. The device name is a readable string label that can be modified in the PHANTOM Configuration control panel.

```
hHD = hdInitDevice(HD_DEFAULT_DEVICE);
if (HD_DEVICE_ERROR(hdGetError()))
{
   exit(-1);
}
```

The second step is to create a context for the initialized device using **hlCreateContext()**. The context maintains the state that persists between frame intervals and is used for haptic rendering.

```
hHLRC = hlCreateContext(hHD);
```

The third step is to make the context current by calling **hlMakeCurrent()**.

```
hlMakeCurrent(hHLRC);
```

## Rendering Contexts

In HLAPI, all commands require that there be an active rendering context. The rendering context contains the current haptic rendering state and serves as a target for all HLAPI commands.

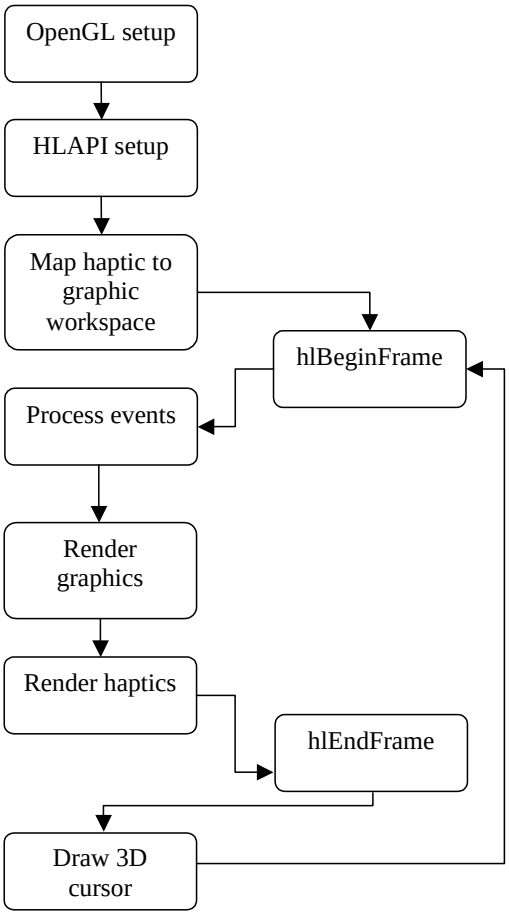
Calling **hlMakeCurrent()** on a rendering context, sets the context as the active context for the current thread. Like in OpenGL, all HLAPI commands made from this thread will operate on the active context. For this thread, it is possible to use multiple rendering contexts in the same thread by making additional calls to **hlMakeCurrent()** to switch the active context. It is also possible to make calls to the same rendering context from multiple threads, but as in OpenGL, the HLAPI routines are not thread safe, so only one thread may make calls to the shared rendering context at a time. To ensure thread safety, a rendering context should only be active in one thread at a time. This can be done by using a *critical section* or *mutex* to synchronize calls to **hlMakeCurrent()** for the shared context.

## Haptic Frames

All haptic rendering commands in the HLAPI must be used inside a haptic frame. A haptic frame is bracketed at the start and end by calls to **hlBeginFrame()** and **hlEndFrame()** respectively. Explicitly marking the beginning and end of the haptic frame allows the API to properly synchronize changes to the state and to the rendering engine.

In a typical program, there will be one haptic rendering frame for each graphic rendering frame. Note that this is very different from a typical HDAPI program in which the haptics framerate is 1000 hz and the graphics framerate is much slower (usually 30-60 hz). In HLAPI programs, often the haptics and graphics are updated one right after the other, or even simultaneously. Generally, the haptics and graphics rendering calls occur in the same thread since they both access the same geometry to render. However, there is no requirement that haptics and graphics framerates match nor that the haptics and graphics be updated in the same thread.

In a typical program, **hlBeginFrame()** is called at the top of the rendering loop, so that any objects in the scene that depend on the haptic device or proxy state have the most current data. **hlEndFrame()** is called at the end of the rendering loop to flush the changes to the haptic rendering engine at the same time that the graphics are flushed so that two will be in synch.




At the start of the haptic frame, **hlBeginFrame()** samples the current haptic rendering state from the haptic rendering thread. **hlEndFrame()** will commit the rendered haptic frame and will synchronously resolve any dynamic changes by updating the proxy position.

In addition to updating haptic rendering state available to the client thread, **hlBeginFrame()** also updates the world coordinate reference frame used by the haptic rendering engine. By default, **hlBeginFrame()** samples the current GL_MODELVIEW_MATRIX from OpenGL to provide a world coordinate space for the entire haptic frame. All positions, vectors and transforms queried through **hlGet*()** or **hlCacheGet*()** in the client or collision threads will be transformed into that world coordinate space. Typically, the GL_MODELVIEW_MATRIX contains just the world to view transform at the beginning of a render pass.

All HLAPI commands that query haptic device or proxy state and are called between the same begin/end frame pair will report the same results. For example, multiple calls to query the haptic device position during a single frame will all report the same exact position, the position at the time **hlBeginFrame()** was called, even if the actual position of the haptic device has changed since the start of the frame. This is done to avoid problems where, for example, during a frame, the program moves multiple objects in a scene by the amount that the haptic device moved. In that situation, reporting different haptic device movements at different times during the frame would cause the objects to be moved out of synch.

At the end of the haptic frame, all changes made to the state, such as the specification of shapes and force effects, are flushed to the haptic rendering engine. The **hlEndFrame()** call is, therefore, similar to doing a **glFlush()** followed by swapping the buffers in a double buffered OpenGL program. This allows a program to make multiple changes to the scene being rendered during a frame and have the changes all occur simultaneously at the end of the frame.

## Rendering Shapes

Shape rendering in HLAPI is used to render surfaces and solid objects. Shapes may be created out of multiple geometric primitives such as lines, points and polygons. Custom shape types may also be created by specifying callback functions. Shape rendering is done using the proxy method as described in “Proxy Rendering” on page 5-2.

### Begin/End Shape

Shape geometry is specified using OpenGL commands bracketed by calls to **hlBeginShape()** and **hlEndShape()**. HLAPI captures the geometry specified by the OpenGL commands, and uses this geometry to perform haptic rendering. For example, the following code renders a 1x2 rectangle in the XY plane as an HLAPI shape:

```
// start the haptic shape
hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId);

glBegin(GL_POLYGON);
glVertex3f(0, 0, 0);
glVertex3f(1, 0, 0);
glVertex3f(1, 2, 0);
glVertex3f(0, 2, 0);
```

```
glEnd();

// end the haptic shape
hlEndShape();
```

## Shape Identifiers

Every shape must have a unique integer identifier. The rendering engine uses this identifier to detect changes to a shape from frame to frame so that it may render the correct forces for the shape as it changes. Before rendering a shape, allocate a new shape identifier using the routine **hlGenShapes().**

```
HLuint myShapeId;
myShapeId = hlGenShapes(1);
```

This identifier should be passed to **hlBeginShape(),** every time your shape is rendered. When you no longer need to render the shape, you should free the shape identifier by calling **hlDeleteShapes()**, so that others may use it.

## Shape Types

There are two different ways that the HLAPI captures geometry from OpenGL commands: using the depth buffer and using the feedback buffer. When rendering a shape, you must specify which method to use for your shape by passing either HL_SHAPE_DEPTH_BUFFER or HL_SHAPE_FEEDBACK_BUFFER.

## Depth Buffer

Depth buffer shapes use the OpenGL depth buffer to capture shape geometry. While the feedback buffer shape stores points, line segments and polygons to use for haptic rendering, the depth buffer shape does haptic rendering using an image read from the depth buffer. When **hlEndShape()** is called, the API reads an image from the OpenGL depth buffer. This image is then passed to the collision thread and is used for collisions with the proxy. Any OpenGL commands that modify the depth buffer will be captured as part of the shape and rendered haptically. This includes any routines that generate polygons or other primitives that modify the depth buffer as well as any shaders or textures that modify the OpenGL depth buffer.

Since the depth buffer does not store geometric primitives, it cannot be used to render points and lines using the HL_CONSTRAINT touch model. It can however, be used to render polygons and polygon meshes as constraints.

Because rendering to the depth buffer turns the 3D geometry into an image, it is important that the image be rendered using the correct viewpoint. You will only be able to feel the portions of the geometry that are viewable from the viewpoint used to render the image. This means that you cannot feel the backside of an object or any undercuts. By default, depth buffer shapes are rendered using the current OpenGL viewing parameters. In general, this is the same view that is used for graphics rendering. In this case, using a

depth buffer shape, you will only be able to feel the portions of the shape that you can see. However, if you enable the haptic camera view optimization, the HLAPI will automatically adjust the OpenGL viewing parameters based on the motion and mapping of the haptic device in the scene. This will enable you to feel portions of the shape, even if they are not visible on the screen. This works well for most shapes although there may be noticeable discontinuities when feeling shapes with deep, narrow grooves or tunnels. For such shapes, it is better to use a feedback buffer shape. Note that by default the haptic camera view is disabled. To enable it, call:

```
hlEnable(HL_HAPTIC_CAMERA_VIEW);
```

Unlike with feedback buffer shapes, depth buffer shapes may be rendered once per frame for both haptics and graphics. If haptic camera view is disabled, the depth buffer image needed for haptics is the same that is generated for the graphics. You can combine the graphics and haptics rendering by simply bracketing your existing graphics rendering code with an **hlBeginShape()** and an **hlEndShape()**:

```
hlBeginFrame();

// clear color and depth buffers for new frame
// haptic rendering requires a clear depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// combined graphic and haptic rendering pass
hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId);
drawMyShape();
hlEndShape();

// swap buffers to show graphics results on screen
glutSwapBuffers();

// flush haptics changes
hlEndFrame();
```

If you are using the haptic camera view optimization, the above approach will not work, since HLAPI will change the viewing parameters based on the motion and mapping of the haptic device. This will cause the graphics to be rendered from this modified view. When using haptic camera view, or when you want your haptics and graphics rendering routines to be different, you will need to render the haptics and graphics separately as you would with a feedback buffer shape. The following code snippet shows how such a program would be structured:

```
hlBeginFrame();

// clear color and depth buffers for new frame
// haptic rendering requires a clear depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// graphic rendering pass
drawMyShapeGraphic();

// swap buffers to show graphics results on screen
```

```
// do this before rendering haptics so we don't
// get haptic camera view rendered to screen
glutSwapBuffers();

// haptic rendering pass - clear the depth buffer first
// so that you don't mix in the depth buffer image
// graphics rendering
glClear(GL_DEPTH_BUFFER_BIT);
hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId);
drawMyShapeHaptic();
hlEndShape();

// flush haptics changes
hlEndFrame();
```

Note the placement of the calls to **glClear()** and to **glutSwapBuffers()**. Because a depth buffer shape does draw to the color and depth buffer (unlike a feedback buffer shape), you have to be careful to not render the view from the haptic camera to the screen and also to clear the depth buffer before rendering the haptics so that the depth image from the graphics does not get mixed in with that of the haptics.

## Feedback Buffer

Feedback buffer shapes use the OpenGL feedback buffer to capture geometric primitives for haptic rendering. When you begin a feedback buffer shape, by calling **hlBeginShape(),** HLAPI automatically allocates a feedback buffer and sets the OpenGL rendering mode to feedback mode. When in feedback buffer mode, rather than rendering geometry to the screen, the geometric primitives that would be rendered are saved into the feedback buffer. All OpenGL commands that generate points, lines and polygons will be captured. Other OpenGL commands, such as those that set textures and materials, will be ignored. When **hlEndShape()** is called, the primitives written to the feedback buffer are saved by the haptic rendering engine and used for force computations in the haptic rendering threads.

When using the feedback buffer shapes, you should use the **hlHinti()** command with HL_SHAPE_FEEDBACK_BUFFER_VERTICES to tell the API the number of vertices that will be rendered. HLAPI uses this information to allocate memory for the feedback buffer. OpenGL requires that sufficient memory be allocated prior to rendering the geometry. If not enough memory is allocated, some geometry will be lost and HL_OUT_OF_MEMORY error will be set. It is therefore better to over allocate than it is to under allocate. If no hint value is specified, HLAPI will allocate space for 65536 vertices.

The following code snippet shows how to render a rectangle using a feedback buffer shape:

```
hlHinti(HL_SHAPE_FEEDBACK_BUFFER_VERTICES, 4);

hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);

glBegin(GL_POLYGON);
glVertex3f(0, 0, 0);
glVertex3f(1, 0, 0);
glVertex3f(1, 2, 0);
glVertex3f(0, 2, 0);
glEnd();

hlEndShape();
```

The OpenGL rendering commands that may be used are not limited to direct calls to **glBegin(), glEnd()** and **glVertex()**. You may call any routines that generate geometric primitives, such as calling display lists, vertex arrays or glu NURB rendering functions. In many cases the same OpenGL calls can be made to render a shape for both graphics and haptics. In this case you can simply call your rendering routine twice, once for graphics rendering and once for haptics:

```
// graphic rendering pass
drawMyShape();

// haptic rendering pass
hlHinti(HL_SHAPE_FEEDBACK_BUFFER_VERTICES, nVertices);
hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);
drawMyShape();
hlEndShape();
```

While the feedback buffer shape will capture lines and points, they will only be used for haptic rendering when the touch model is set to HL_CONSTRAINT. Since the haptic device proxy is modeled as a single point, it may be constrained to points, lines and polygons, but it may only contact polygons.

When creating a feedback buffer shape, it is important not to change the OpenGL culling settings. Specifically, you should not call **glCullFace()**, or **glEnable() / glDisable()** with GL_CULL_FACE in between the **hlBeginShape()** and **hlEndShape()** calls. If you do, you may not be able to feel parts of your shape. For example, if you have back face culling enabled you will not be able to feel any of the back facing polygons of your shape since OpenGL will cull them out before they can be saved in the feedback buffer. While these faces are not visible in graphics rendering, you may reach around the back of a shape with the haptic device to touch an area that you cannot see. This is an easy mistake in an HLAPI program since you often share rendering routines between the haptics and the graphics and it is easy to forget that for graphics rendering, you may have changed the culling settings. The HLAPI changes the OpenGL culling state in **hlBeginShape()** and restores the previous value in **hlEndShape()** so you do not need to change the setting yourself.

The HLAPI expects that polygon meshes be reasonably well behaved. Specifically, if a mesh has even very small gaps between adjacent triangles, this will allow the proxy to pass through the mesh giving the user the impression that they have fallen through the object. In addition, meshes which are self-intersecting, non-manifold or meshes where polygons have inconsistent winding orders may cause fall through or other haptic artifacts.

## Optimizing Shape Rendering

In graphics rendering, many optimizations, such as view frustum culling and back face culling, work by rendering only the geometry that is actually viewable. In haptics rendering, performance may be optimized by rendering only geometry that is actually touchable. This can be done by considering only geometry that is near the current proxy position. The HLAPI provides a number of ways to optimize haptic rendering: adaptive viewport, haptic camera view, and culling with spatial partitions. Each is described in detail below.

## Adaptive Viewport

When using depth buffer shapes, performance may be improved by enabling the adaptive viewport optimization. This optimization limits the region of the depth buffer that is read into memory to be used for haptic rendering to the area near the current proxy position. The performance improvement will depend on the speed at which the graphics card is able to read the depth image from the on board memory of the graphics accelerator. On many graphics accelerators, reading data from the depth buffer can be very costly.

To turn on the adaptive viewport, make the following call before calling **hlBeginShape()**:

```
hlEnable(HL_ADAPTIVE_VIEWPORT);
```

Once this call is made, all newly created depth buffer shapes will use the adaptive viewport. To turn off the adaptive viewport call:

```
hlDisable(HL_ADAPTIVE_VIEWPORT);
```

In order to use the adaptive viewport, the scene must be redrawn regularly when the haptic device is moving, otherwise the haptic device may leave the region of the scene covered by the portion of the depth image that was copied. The portion of the depth image read is refreshed every time the shape is drawn. For normal use of the haptic device, redrawing the graphics at a normal 30-60hz framerate is sufficient when using adaptive viewport, however in applications where the user moves the haptic device very quickly, you may notice discontinuities in the force output.

## Haptic Camera View

When the haptic camera view is enabled, HLAPI will automatically modify the viewing parameters used when rendering a depth buffer or feedback buffer shape so that only a subset of the geometry near the proxy position will be rendered. When the haptic camera view is enabled, HLAPI modifies the OpenGL view frustum so that only the shape geometry near the proxy position is rendered.

For feedback buffer shapes, this can dramatically increase performance by reducing the number of geometric primitives considered for haptic rendering. The improvement will depend on the density of the geometry in the region around the proxy, since denser geometry will lead to a larger number of primitives in the haptic view frustum.

For depth buffer shapes, this offers less of a performance improvement, since once the primitives have been rendered to the depth buffer, the actual haptic rendering of a depth buffer image is not dependent on the number of primitives. That said, there is some performance benefit to considering only the geometry near the proxy when generating a depth buffer image. In addition, when using haptic camera view, HLAPI generates a depth buffer image that is a subset of the full depth buffer used by the graphics window so as with adaptive viewport, less data is read back from the depth buffer. If haptic camera view is enabled, the adaptive viewport setting is ignored. In addition, for depth buffer shapes, using haptic camera view allows you to feel parts of the geometry that are not viewable from the graphics view.

To turn on the haptic camera view, make the following call before calling **hlBeginShape()**:

```
hlEnable(HL_HAPTIC_CAMERA_VIEW);
```

Once this call is made, all newly created depth buffer and feedback buffer shapes will use the haptic camera view. To turn off the haptic camera view call:

```
hlDisable(HL_HAPTIC_CAMERA_VIEW);
```

As with the adaptive viewport, in order to use the haptic camera view, the scene must be redrawn regularly when the haptic device is moving. Each time the shape is rendered, only the geometry near the proxy is captured, so if the haptic device moves far enough away from the proxy position at the time the shape was specified, there may not be any recorded geometry near the device position. This can lead to the shape not feeling correct and in some cases to the haptic device “kicking” when new geometry near the device position is finally recorded. For normal use of the haptic device, redrawing the graphics at a normal 30-60hz framerate is sufficient when using haptic camera, however for applications where the user moves the haptic device very quickly you may not want to use haptic camera view.

## Culling with Spatial Partitions

When rendering shapes with very large numbers of primitives additional culling based on the haptic camera view is recommended. While the haptic camera view will cull out primitives which are not near the proxy, with a large enough number of primitives, this culling itself can become prohibitively expensive and lead to low framerates. By using a spatial partition such as a BSP tree, octree or hierarchical bounding spheres, large groups of primitives may be culled at once. HLAPI does not provide a spatial partition as part of the standard API because building an efficient partition is dependent on the data structures and types of data specific to each application. The HapticViewer sample application that comes with HLAPI provides a simple example of spatial partitioning for haptic rendering using a binary tree.

To perform haptic culling using a spatial partition, first you need to determine the region in space that the haptic camera view will consider for haptic rendering. This is simply the OpenGL view frustum that HLAPI sets as part of the call to **hlBeginShape()** when haptic camera view is enabled. This can be queried using the OpenGL **glGet()** functions. The viewing frustum used by the haptic camera view will always be based on an orthographic projection, so it will always be a box, however it will not always be axis aligned. Once you know the frustum box to consider, then you use your spatial partition to find the subset of geometry which is inside or partially inside this box. Finally, this subset of geometry should be drawn using OpenGL.

The following code snippet shows how to do this. For a full example, see the HapticViewer sample application.

```
hlEnable(HL_HAPTIC_CAMERA_VIEW);

hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, myShapeId);

// get frustum box from OpenGL
hduVector3Dd frustumCorners[8];
getFrustum(frustumCorners);

// render only the geometry in the frustum
spatialPartition->renderOnlyInBox(frustumCorners);

hlEndShape();
```

The **getFrustum()** involves reading the viewing parameters from OpenGL and using them to reconstruct the view frustum:

```
void getFrustum(hduVector3Dd* frustum)
{
   // get OpenGL matrices
   GLdouble projection[16];
   GLdouble modelview[16];
   glGetDoublev(GL_PROJECTION_MATRIX, projection);
   glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
   // invert modelview matrix to get model to eyetransform
   hduMatrix eyeTmodel =
                hduMatrix(modelview).getInverse();
```

```
    // invert projection matrix to clip to eye transform
    hduMatrix clipTeye =
                hduMatrix(projection).getInverse();

    // compose the two together to get clip to model
    // transform
    hduMatrix clipTmodel = clipTeye.multRight(eyeTmodel);

    // Compute the edges of the frustum by transforming
    // canonical clip coordinates for the corners
    // into eye space.
    frustum[0] = hduVector3Dd(-1, -1, -1) * clipTmodel;
    frustum[1] = hduVector3Dd(-1, -1, 1) * clipTmodel;
    frustum[2] = hduVector3Dd( 1, -1, -1) * clipTmodel;
    frustum[3] = hduVector3Dd( 1, -1, 1) * clipTmodel;
    frustum[4] = hduVector3Dd( 1, 1, -1) * clipTmodel;
    frustum[5] = hduVector3Dd( 1, 1, 1) * clipTmodel;
    frustum[6] = hduVector3Dd(-1, 1, -1) * clipTmodel;
    frustum[7] = hduVector3Dd(-1, 1, 1) * clipTmodel;
}
```

The routine renderOnlyInBox involves using the spatial partition to efficiently determine which primitives are in the box with the corners specified and render them. The HapticViewer sample shows an example of how this is implemented using a binary tree.

## Which Shape Type Should I Use?

For large numbers of primitives, depth buffer shapes are more efficient and use less memory since the haptic rendering engine only needs to use the depth image for haptic rendering. The complexity of haptic rendering on a depth image is independent of the number of primitives used to generate the image. At the same time, for small numbers of primitives, feedback buffer shapes are more efficient and use less memory due to the overhead required to generate and store the depth image.

Depth buffer shapes are less accurate than feedback buffer shapes although in nearly all applications, the difference in accuracy is undetectable. With depth buffer shapes, the geometry is transformed into a 2D depth image before being rendered haptically, and this is a “lossy transformation,” particularly when the shape has undercuts that are not visible from the camera position. The haptic camera view attempts to minimize the undercuts by choosing an appropriate camera position, however in some cases no camera position can eliminate all undercuts. This is particularly difficult on shapes with deep grooves or narrow tunnels. Such shapes are best rendered as feedback buffer shapes.

If you are rendering lines and points to be used as constraints, you must use a feedback buffer shape since depth buffer shapes cannot capture points and lines.

Also note, DepthBufferShape with Haptic Camera View and HL_FRONT_AND_BACK touchable faces is not fully supported for dynamically changing shapes. It will work as long as you remain in contact with the shape. This will be fixed in a future release when we add support for multiple camera views.

# Mapping Haptic Device to Graphics Scene

All applications will have to determine an appropriate mapping of the haptic workspace to the graphics scene. Defining a mapping between the workspace and the graphic scene will describe how movement of the physical device translates to movement in the graphic scene.

## The Haptic Workspace

The haptic workspace is the physical space reachable by the haptic device. The dimensions of the haptic workspace can be obtained by calling **hlGetDoublev()** with HL_WORKSPACE. The values returned are in millimeters.

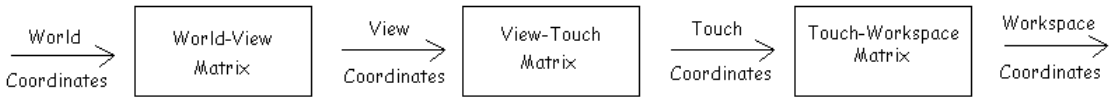
```
HLdouble workspaceDims[6];
hlGetDoublev(HL_WORKSPACE, workspaceDims);
```

Most applications will choose to use the entire workspace. Those applications that desire to use a subset of the workspace can set the usable workspace by calling **hlWorkspace().** The following call sets the usable workspace to be a box 160mm in X, 150 mm in Y, and 60 mm in Z. The workspace coordinates are such that positive Z is towards the user.

```
hlWorkspace(-80, -80, -70, 80, 80, 20);
// left, bottom, back, right, top, front
```

## Matrix Stacks

HLAPI provides two matrix stacks to define the mapping from the haptic workspace to the graphics scene. These matrix stacks are HL_VIEWTOUCH_MATRIX and HL_TOUCHWORKSPACE_MATRIX. The matrices are 4x4. The mapping from the haptic workspace to the graphic scene is defined as follows:




- World coordinates are the global frame of reference for the graphics scene.
- View coordinates are the local coordinates of the camera (eye coordinates).
- Touch coordinates are the parent coordinate system of the workspace. Touch coordinates represent the basic mapping of the workspace to view coordinates independent of further workspace transformation.
- Workspace coordinates are the local coordinates of the haptic device.
- World-view matrix defines the transformation to the camera coordinate frame. The world-view matrix is captured when **hlBeginFrame()** is called.

- View-touch matrix defines the rotation and translation of the haptic workspace relative to view coordinates independent of the workspace mapping. The view-touch matrix will be the identity for most applications.
- Touch-workspace matrix defines the mapping of the workspace to view coordinates. The mapping will contain a scale to map the workspace and a translation to orient the workspace to the target mapping in view coordinates.

The Touch-Workspace and View-Touch matrix stacks function in much the same way as matrix stacks in OpenGL. The HLAPI maintains a current matrix stack and all matrix functions affect the current matrix. Set the current matrix by calling **hlMatrixMode()** with either HL_VIEWTOUCH or HL_TOUCHWORKSPACE.

Functions that affect the current matrix stack are **hlPushMatrix(), hlPopMatrix(), hlLoadMatrix(), hlMultMatrix(), hlOrtho()** and several convenience routines in hlu.

Call **hlGetDoublev()** with HL_VIEWTOUCH_MATRIX or HL_TOUCHWORKSPACE_MATRIX to retrieve the top of a matrix stack.

**Touch-workspace Matrix**

The purpose of the touch-workspace matrix is to define the mapping between the workspace and view coordinates. The matrix will contain a scale to match the size of the workspace with the target in view coordinates and a translation to orient the closest part of the workspace with the closest part of the target of the mapping. Most applications will only modify the touch-workspace matrix leaving the view-touch matrix as the identity matrix. The touch-workspace matrix functions much like the projection matrix in OpenGL.

**View-touch Matrix**

The purpose of the View-Touch matrix is to orient the mapped workspace to the target in view coordinates. For example an application may want to map the X dimension of the workspace (longest dimension) with the Z dimension of the view. To do so, manipulate the View-Touch matrix independent of the Touch-Workspace matrix.

## Mapping

Most applications will map the workspace to view coordinates such that everything that is visible is also touchable. Other applications may only be interested in a portion of the viewing volume, a collection of objects, or even just a subset of a single object. The HLAPI provides both convenience routines for the most common mappings as well as a flexible mechanism for advanced users. All routines depend on the current state of workspace dimensions (HL_WORKSPACE).

**Basic Mapping**

When defining the workspace mapping, the physical dimensions of the workspace must be considered. The majority of haptic devices will have a physical workspace that is not uniform in all dimensions. For example the PHANTOM Omni device workspace has dimensions 160w x 120h x 70d mm.

Directly mapping such a workspace would require a non-uniform scale matrix and would result in non-uniform movement of the proxy in the scene. To define a uniform mapping you will have to determine how best to fit the haptic workspace about the area of interest. Fortunately the HLAPI provides convenience routines to define the most common workspace mappings.

Most applications will have a uniform mapping of the haptic workspace to the viewable scene. **hluFitWorkspace()** will define a uniform workspace mapping such that workspace completely encloses the view volume. Call **hluFitWorkspace()** with the projection matrix that defines the viewing volume:

```
hlMatrixMode(HL_TOUCHWORKSPACE);
hluFitWorkspace(projectionMatrix);
```

The constructed matrix is multiplied with top of the current matrix stack.

Applications may only be interested in a portion of the scene. This may be defined as a portion of the viewing volume, a single object or a collection of objects. Call **hluFitWorkspaceBox()** with the desired extents and a matrix that transforms the extents into view coordinates. If the extents are in view coordinates, the matrix will be the identity. If the extents represent the bounding box of an object, the matrix will be the model-view matrix used to draw the object.

```
HLdouble minPoint[3], maxPoint[3];
hluFitWorkspaceBox(modelMatrix, minPoint, maxPoint);
```

**hluFitWorkspaceBox()** will define a uniform mapping of the haptic workspace such that the workspace encloses a bounding box defined by minPoint and maxPoint where modelMatrix is the matrix that transforms the points defined by minPoint and maxPoint to view coordinates.

**Advanced Mapping**

### Uniform vs. Non-Uniform Mapping

Both **hluFitWorkspace()** and **hluFitWorkspaceBox()** define a uniform mapping of the workspace to view coordinates. While this allows for uniform proxy movement in the scene, it does give up a portion of the haptic workspace to allow for a uniform scale. If your application is such that using the entire haptic workspace is more important than uniform proxy movement or if non-uniform proxy movement will be imperceptible for your application, you may want to use a non-uniform workspace scale.

**hluFitWorkpaceNonUniform()** and **hluFitWorkspaceBoxNonUniform()** are the functional non-uniform equivalents of **hluFitWorkspace()** and **hluFitWorkspaceBox()**.

**Touch-Workspace Matrix** As stated, the touch-workspace matrix defines the basic mapping between the workspace and view coordinates. Although the hlu functions are flexible, application developers may require more control when defining the mapping. You may consider generating an intermediary projection matrix solely for the purpose of workspace mapping. Instead, **hlOrtho()** provides more direct manipulation of the workspace mapping. **hlOrtho()** defines a non-uniform mapping such that the haptic workspace will be a fit box defined in view coordinates. The following call will map the workspace to a box in view coordinates centered about the origin 20 units on a side:

```
hlOrtho(-10.0, 10,0, -10.0, 10.0, -10.0, 10.0);
// left, bottom, near, right, top far
```

**View-Touch Matrix** The view-touch matrix provides further control of the workspace-view coordinate mapping. The view-touch matrix can be modified independently of the touch-workspace matrix to affect change in workspace mapping. An application that

desires the optimal workspace mapping based on a new view-touch matrix must refit the workspace after the view-touch matrix changes. For example, an application may want to map the X-axis of the device (longest dimension) to the Z-axis in view coordinates. The function **hluFeelFrom()** provides the mechanism to translate and orient the workspace to the view. The following code will reorient the haptic workspace:

```
// workspace looking from right
HLdouble handx = 1, handy = 0, handz = 0;

// at scene origin
HLdouble centerx = 0, centery = 0, centerz = 0;

// up vector
HLdouble upx = 0; upy = 1;upz = 0;

hluFeelFrom(handx, handy, handz, centerx, centery,
            centerz, upx, upy, upz);

hluFitWorkpace(projectionMatrix):
```

## Drawing a 3D Cursor

The application user will often need to visualize the proxy position relative to scene objects in order to interact with the virtual environment. A 3D cursor is the graphic representation of the proxy in the scene. To draw a 3D cursor you will need to get the 3D world coordinate position of the proxy and determine a size for the 3D cursor. The proxy position can be obtained by calling **hlGetDoublev()** with HL_PROXY_POSITION:

```
Hldouble proxyPosition[3];
hlGetDoublev(HL_PROXY_POSITION, proxyPosition);
```

You will need to provide a scale for the 3D cursor as it will be represented by a 3D object in the scene. The function **hluScreenToModelScale()** will return the scale to use when drawing the 3D cursor such that it will occupy a single screen pixel when the cursor is at the near plane of the viewing frustum.

For 3D cursor objects that are not rotation invariant, you can either obtain the cursor rotation in world coordinates by calling **hlGetDoublev()** with HL_PROXY_ROTATION or the cursor transformation from the haptic workspace to world coordinates by calling **hlGetDoublev()** with HL_PROXY_TRANSFORM.

The following code snippet shows drawing a 3D cursor:

```
// cursor size in pixels at the near plane
#define CURSOR_SCALE_PIXELS 20

GLdouble modelview[16];
GLdouble projection[16];
GLint viewport[4];

glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
```

```
glGetDoublev(GL_PROJECTION_MATRIX, projection);
glGetIntegerv(GL_VIEWPORT, viewport);

glPushMatrix();

// get proxy position in world coordinates
HLdouble proxyPosition[3];
hlGetDoublev(HL_PROXY_POSITION, proxyPosition);

// transform to draw cursor in world coordinates
glTranslatef(proxyPosition[0], proxyPosition[1],
  proxyPosition[2]);

// compute cursor scale
HLdouble gCursorScale;
gCursorScale = hluScreenToModelScale(modelview, projection,
viewport);
gCursorScale *= CURSOR_SIZE_PIXELS;

glScaled(gCursorScale, gCursorScale, gCursorScale);

drawSphere();

glPopMatrix();
```

## Material Properties

Material properties control the tactile properties of the surface. This is analogous to visual properties—visually, material properties include such identifiers as color, glow, specular highlights; haptically, material properties include such identifiers as stiffness and friction.

Material properties are specified using **hlMaterial()**. They can be applied to either the front, back, or both faces of an object.

### Stiffness

Stiffness is set by calling **hlMaterial()** with the HL_STIFFNESS property.

```
hlMaterialf(HL_FRONT_AND_BACK, HL_STIFFNESS, 0.7);
```

Stiffness defines how hard an object feels. Mathematically, stiffness determines the rate that the resistance force increases as the device attempts to penetrate the surface. Forces are generated using the Hooke's Law equation F=kx, where “k” is the stiffness, or spring constant and “x” is the vector representing penetration depth. A higher stiffness value thus will result in greater resistance when the device pushes against the surface.

Real world hard surfaces, which contain high stiffness, include metal or glass. Soft surfaces, with low stiffness, include Jell-O® and rubber.

Stiffness may be any value between 0 and 1, where 0 represents a surface with no resistance and 1 represents the stiffest surface the haptic device is capable of rendering stably.

Setting stiffness too high may cause instability. The stiffness force is intended to resist penetration into the object, but an exceptionally high stiffness may cause the device to kick or buzz. For example, if the stiffness is set to an unreasonably high number, then the device would experience a strong kick in the opposite direction whenever it even lightly touched the surface.

## Damping

Damping is set by calling **hlMaterial()** with the HL_DAMPING property.

```
hlMaterialf(HL_FRONT_AND_BACK, HL_DAMPING, 0.1);
```

Damping adds a velocity-dependent property to an object. Damping is governed by the equation F=kv, where “k” is the damping coefficient and “v” is the velocity of the device.

One real world example of an object with high damping is corn syrup. The more forceful your contact, the more resistance the corn syrup provides.

Damping may range from 0 to 1 where 0 means no damping and 1 means the most damping the haptic device is capable of rendering.

Setting damping too high can cause instability and oscillation. The purpose of damping is to provide some retardation to the device's velocity; however, if the value is too high, then the reaction force could instead send the device in the opposite direction of its current motion. Then in the next iteration, the damping force would again send the device in the opposite direction, etc. This oscillation will manifest as buzzing.

## Friction

Friction is set by calling **hlMaterial()** with either the HL_STATIC_FRICTION or HL_DYNAMIC_FRICTION property.

```
hlMaterialf(HL_FRONT_AND_BACK, HL_STATIC_FRICTION, 0.2);
hlMaterialf(HL_FRONT_AND_BACK, HL_DYNAMIC_FRICTION,
            0.3);
```

Friction provides resistance to lateral motion on an object. Friction is classified into two categories: static and dynamic (sometimes also designated “stick-slip friction”). Static friction is the friction experienced when the device initially begins motion starting from rest along the surface. Dynamic friction is the friction experienced as the device is moving along the surface.

For example, if you put your finger on ice, then slide your finger along the surface, the ice feels initially sticky, then feels smooth after your finger has started to move. Ice is an example of an object that has high static friction but low dynamic friction. Rubber, on the other hand, has both high static and dynamic friction. Steel can have both low static and dynamic friction.

### Popthrough

Popthrough is set by calling **hlMaterial()** with the HL_POPTHROUGH property.

```
hlMaterialf(HL_FRONT_AND_BACK, HL_POPTHROUGH, 0.5);
```

Popthrough defines how hard the device must push against the surface of an object before it pops through (or “pushes through”) to the other side. A value of 0 or false turns popthrough off, so that the surface does not allow penetration. A positive value controls the popthrough threshold, where a higher value means that the device must push harder before popping through. The popthrough value roughly corresponds to a ratio of the device's maximum nominal force threshold.

This is not an attribute that has a real-world physical counterpart because surfaces in real life do not allow an object to push through them without themselves being destroyed. A fictitous example might be a sheet of paper that a user pushes against until he tears a hole through it, where that hole then restores itself instantaneously such that the user ends up on the other side of the paper and the paper itself is unchanged.

Popthrough has many types of useful application. An application might set a surface to be touchable from the front and back, then set a popthrough threshold such that the user can push through the surface to feel its back side, then pop out of the surface to feel its front side. For example, the user may want to at times interact with either the outside or inside surface of a sphere without having to explicitly toggle which side is touchable. Another application might have some infinite plane that's used as a guide, where the user can get through the plane by pushing with enough force on it. The alternative would be to force the user to explicitly turn off the plane whenever he was interested in interacting with the space below it.

Setting the popthrough value too low may make it difficult to touch the surface without pushing through to the other side. Setting the popthrough value too high may mean the user will experience a maximum force error before reaching the popthrough value.

## Surface Constraints

By default, shapes in the HLAPI are rendered so that the proxy may not pass through them, giving the impression of a solid object. This is referred to as contact rendering mode. The API also supports a constraint rendering mode where the proxy is constrained to the surface of a shape giving the impression of a magnetic object to which the haptic device position sticks.

To render a shape as a constraint, use the function **hlTouchModel()** with the HL_CONSTRAINT parameter. Once the touch model is set to constraint, all newly specified shapes will be rendered as constraints. To render other shapes as standard contact shapes, call **hlTouchModel()** again with HL_CONTACT parameter.

## Snap Distance

Snap distance is set by calling **hlTouchModel()** with the HL_SNAP_DISTANCE property.

```
hlTouchModel(HL_CONSTRAINT);
hlTouchModelf(HL_SNAP_DISTANCE, 1.5);
```

Objects that behave as constraints will force the device to their surface whenever the device is within a certain proximity of their surface. This proximity is the snap distance. Beyond the proximity, those constraints will be inactive and thus provide no force contributions. Once the device enters the proximity and for as long as it stays within that distance, the object will confine it to its surface. It is said that the device is “stuck” to the surface of the constraint.

A simple example of a constraint is a line constraint. Whenever the device is within the snap distance of the line, the line projects the proxy position to itself and confines it there. Forces are generated proportional to the distance between the device position and the nearest point on the line. Once that distance exceeds the snap distance, the proxy is freed and the line constraint no longer operates on the device.

The following code snippet renders a triangle in contact mode with its edges rendered as constraint line segments. The effect is that you can slide along the triangle and the haptic device snaps onto an edge when you get near it.

```
hlBeginFrame();
hduVector3Dd triangleVerts[3];
triangleVerts[0].set(0, -50, 0);
triangleVerts[1].set(-50, 0, 0);
triangleVerts[2].set(50, 0, 0);

// draw edges as constraint
hlTouchModelf(HL_SNAP_DISTANCE, 1.5);
hlTouchModel(HL_CONSTRAINT);
hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, edgesShapeId);
glBegin(GL_LINE_LOOP);
for (int i = 0; i < 3; ++i)
   glVertex3dv(triangleVerts[i]);
glEnd();
hlEndShape();
// draw face as contact
hlTouchModel(HL_CONTACT);
hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, faceShapeId);
glBegin(GL_TRIANGLES);
glNormal3f(0, 0, 1);
for (int i = 0; i < 3; ++i)
   glVertex3dv(triangleVerts[i]);
glEnd();
hlEndShape();

hlEndFrame();
```

## Combining Constraints

When multiple constraints are specified and the proxy is within the snap distance of more than one constraint, HLAPI computes the constrained proxy position using the following rules:

**If the constraints do not overlap or intersect:** The proxy will be constrained to the geometry, within the specified snap distance, that is closest to the current proxy. This means that once the proxy is constrained to a primitive, it will remain constrained to that primitive, and will not become constrained to any other primitive until the distance between the primitive and the haptic device position becomes greater than the snap distance for that primitive.

**If the constraints overlap or intersect:** **If the overlapping or intersecting constraints are of different dimensionality,** such as a three dimensional surface and a one dimensional line or a two dimensional surface and a one dimensional point, HLAPI will constrain the proxy first to the primitive of higher dimension and then if the newly constrained proxy is within snap distance of the lower dimension constraint, it will further constrain the proxy to that constraint. This allows for you to move freely along a higher dimensional constraint such as a surface or a line and then be snapped to a lower dimensional constraint within the higher dimensional constraint.

**If the overlapping or intersecting constraints are of the same dimensionality** the proxy will be constrained to whichever primitive is closest to the current proxy. If two constraints are of equal distance to the proxy, the proxy will be constrained to whichever is closest to the haptic device position. The proxy is therefore allowed to move from one constraint to another if both constraints intersect. This allows you to build compound constraints out of groups of primitives of the same dimension that feel continuous such as curve constraints made up of connected line segments or surfaces made up of connected triangles.

## Pushing and Popping Attributes

Attributes can be pushed and popped off the attribute stack the same

way that matrices are handled with **hlPushMatrix** and **hlPopMatrix**.

```
hlPushAttrib(HL_TOUCH_BIT);
hlPopAttrib();
```

Pushing and popping attributes is handled much the same way as in openGL, where the developer can specify which attributes to push. The developer could push all attributes, but typically he'll only be interested in saving a subset of information. A typical example is if the user wants to save off the material properties of the state; he can then push those attributes onto the stack, define new parameters for whatever object he's creating, then pop the attribute stack to restore the previous values. **hlPopAttribute** restores only whatever attributes were pushed by the last **hlPushAttrib**.

See the *Open Haptics API Reference* for a list of valid attributes. The developer can push an individual set of attributes or OR the attribute types together to push multiple ones.

## Effects

Effects provide a way to render forces to the haptic device to simulate arbitrary sensations. Force effects are typically used to generate ambient sensations, like drag, inertia or gravity. These sensations are ambient because they apply throughout the workspace. Force effects can also be used to generate transient sensations, like an impulse. Force effects can be started and stopped or triggered in response to events, like touching a shape or pressing a button on the haptic device. Unlike shape rendering, effects will persist until stopped or until the duration has elapsed for trigger effects.

There are a number of built-in effect types. HLAPI presently supports the following generic force effect types: constant, spring, viscous, and friction. In addition, the force effect facility supports a callback effect type, which allows for custom force effect rendering.

Effects can either be persistent or run for a predefined period of time. This behavior is controlled by the HL function used to invoke the effect. A persistent effect is invoked using **hlStartEffect()**. The force effect will continue to render until **hlStopEffect()** is

called with the corresponding effect identifier. Effect identifiers are allocated using **hlGenEffects()** and are deallocated using **hlDeleteEffects()**. The following example demonstrates starting and stopping a force effect:

```
/* Start an ambient friction effect */
HLuint friction = hlGenEffects(1);
hlBeginFrame();
hlEffectd(HL_EFFECT_PROPERTY_GAIN, 0.2);
hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0.5);
hlStartEffect(HL_EFFECT_FRICTION, friction);
hlEndFrame();

/* Execute main loop of program */
/* Stop the ambient friction effect */
hlBeginFrame();
hlStopEffect(friction);
hlEndFrame();

hlDeleteEffects(friction, 1);
```

Effects can also be invoked as temporal effects, such that they render for a set period of time. This is controlled by invoking the effect using the HL function **hlTriggerEffect()**. Triggering an effect differs from the start/stop approach since it doesn't require an effect identifier and will automatically stop when the effect duration has elapsed. The following example demonstrates triggering a force effect:

```
static const HDdouble direction[3] = { 0, 0, 1 };
static const HDdouble duration = 100; /* ms */
// Trigger an impulse by commanding a force with a
// direction and magnitude for a small duration
hlEffectd(HL_EFFECT_PROPERTY_DURATION, duration);
hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 1.0);
hlEffectdv(HL_EFFECT_PROPERTY_DIRECTION, direction);
hlTriggerEffect(HL_EFFECT_CONSTANT);
```

Effects can be controlled by one or more of the following properties:

- HL_EFFECT_PROPERTY_GAIN
- HL_EFFECT_PROPERTY_MAGNITUDE
- HL_EFFECT_PROPERTY_FREQUENCY
- HL_EFFECT_PROPERTY_DURATION
- HL_EFFECT_PROPERTY_POSITION
- HL_EFFECT_PROPERTY_DIRECTION

These properties can be set by using the **hlEffect*()** functions, which take as arguments the property type and either a scalar or vector, depending on the property. Effect properties that will be referenced by an effect are sampled from the HL context state at the time **hlStartEffect()** or **hlTriggerEffect()** is called. Please refer to the *Open Haptics API Reference* for information about the use of each property for the built-in effects.

HLAPI allows for multiple force effects to be rendered at the same time. All of the effects will be combined to produce one force to be rendered to the device. Also note that the built-in effects utilize the proxy position as input, since this allows for stable effect force

rendering while feeling shapes. Therefore, it is possible to contact a shape while an ambient friction effect is active or to be constrained to a shape while resisting a spring effect.

Effects can be updated in two ways. First, the effect can be stopped, new effect parameters specified via **hlEffect** commands, an then started again.

```
hlStopEffect(frictionEffect)
hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 1.0)
hlStartEffect(FrictionEffect);
```

Effect parameters can be updated in place while the effect is running. This is accomplished through **hlUpdateEffect()**, which takes as an argument an effect that is already running or defined. The current effect state is copied into the specified effect, i.e. the same way the current effect state is applied when the developer call **hlStartEffect()**.

```
hlEffect(HL_EFFECT_PROPERTY_MAGNITURE, 1.0);
hlUpdateEffect(frictioEffect);
```

# Events

The HLAPI allows client programs to be informed via callback functions when various events occur during haptic rendering. You may pass a pointer to a function in your program to the API along with the name of the event you are interested in. That function will be called when the event occurs. The events you can subscribe to include touching a shape, motion of the haptic device and pushing the button on the stylus of the haptic device.

## Event Callbacks

To set a callback for an event, use the function **hlAddEventCallback()** as follows:

```
hlAddEventCallback(HL_EVENT_TOUCH, HL_OBJECT_ANY,
                   HL_CLIENT_THREAD,
                   &touchShapeCallback, NULL);
```

This tells HLAPI to call the function **touchShapeCallback()** when any shape is touched. You may register as many callbacks as you want for the same event. The callback function should be defined as follows:

```
void HLCALLBACK touchShapeCallback(HLenum event,
                                   HLuint object,
                                   HLenum thread,
                                   HLcache *cache,
                                   void *userdata)
{
   hduVector3Dd proxy;
   std::cout << "Touched a Shape" << std::endl;
   hlCacheGetDoublev(cache, HL_PROXY_POSITION, proxy);
   std::cout << "Proxy position is: " << proxy <<
                std::endl;
}
```

This simple callback prints out the position of the proxy when the touch event occurred.

The cache argument that is passed to the callback function may be used with **hlCacheGet*()** functions in order to query the cached haptic rendering state from the time the event occurs. Events may occur in the haptic rendering thread and be handled asynchronously in the client thread. This can lead to delay between the time the actual event occurred and when the callback functioned is invoked. As a result, it is often useful to have access to a snapshot of the haptic rendering state from the point in time when the event occurred.

For client thread events, in order to have your callback function called, you must call the function **hlCheckEvents()** on a regular basis. Most programs will call this function once during each haptic rendering frame, although it may be called at any time from the client thread. **hlCheckEvents()** will call the callback functions for all registered events which have occurred since the last time **hlCheckEvents()** was called. Note that collision thread events will dispatch automatically, provided a valid haptic device handle is assigned to the context.

## Types of Events

The HLAPI allows you to register callbacks for the following events:

**Touch and Untouch**

Touch events occur when the user touches a shape with the haptic device. Specifically, a touch event occurs when the haptic rendering engine determines that the proxy comes in contact with a shape that was rendered in the last frame. Touch events are only reported when the proxy first comes in contact with the shape; they are NOT continually reported every frame that the proxy stays in contact with the shape.

Similarly, untouch events occur when the user stops touching a shape with the haptic device. Specifically, an untouch event occurs when the proxy ceases to be in contact with a shape.

Touch and untouch event callbacks may be registered for a specific shape or they may be registered for any shape. If you pass the id of a shape as the second argument to **hlAddEventCallback()**, the callback will only be invoked when that shape is touched or untouched. If you pass HL_OBJECT_ANY instead, then the callback will be invoked when any shape is touched/untouched.

**Motion**

Motion events are triggered when the proxy position or orientation changes. The amount of motion required to trigger the event may be changed by setting the state variables HL_EVENT_MOTION_LINEAR_TOLERANCE and HL_EVENT_MOTION_ANGULAR_TOLERANCE. These values may be set using the function **hlEventd()** and they may be queried using the function **hlGetDoublev()**. Motion events are triggered when the distance between the current proxy position and the position of the proxy the last time the motion event was triggered exceeds one of these thresholds. The motion event can be registered for the collision thread to allow for a faster sampling rate of proxy motion. To subscribe to a motion use HL_EVENT_MOTION.

**.Button**

Button events are triggered when the user presses one of the buttons on the haptic device. Separate events are triggered for button down and button up. For haptic devices with two buttons, there are separate button up and button down events for each of the two buttons. For haptic devices with one button, or for the first button on a two button device, subscribe to button events using HL_EVENT1_BUTTON_DOWN and HL_EVENT1_BUTTON_UP. For the second button on a two button device use HL_EVENT2_BUTTON_DOWN and HL_EVENT2_BUTTON_DOWN.

**Calibration**

Special events are triggered when the haptic device requires calibration data or a calibration update. See “Calibration” on page 6-27.

**Events for Specific Shapes**

By passing a shape identifier to **hlAddEventCallback()** for motion and button events, your callback will only be called if the event occurs while the shape with that identifier is being touched. If you wish your callback to be called on any shape or even when no shape is being touched, pass HL_OBJECT_ANY as the shape identifier. By specifying a shape id for a button event, your callback will be invoked when clicking the button on that shape. This can be used for example to implement the selection of shapes in the scene. By specifying a shape id for a motion callback, the callback will be invoked when dragging the haptic device along the surface of that shape. This can be used, for example, to implement painting on a shape.

## Events and Threading

Most of the time event callbacks are executed in the client thread but there are cases where you may need to have an event callback executed in one of the haptic rendering threads.

**Client Thread Events**

To have your callback invoked in the client thread, pass HL_CLIENT_THREAD as the third argument to **hlAddEventCallback()**. Callback functions registered to be called in the client thread will be invoked when your program calls **hlCheckEvents()**. Be aware that you may only specify new shapes and effects as part of an event callback if it is called in between an **hlBeginFrame()** and an **hlEndFrame()**. Although it is legal to call hlCheckEvents outside of a begin/end frame pair, one way to ensure that it is safe to specify shapes and effects in your client thread event callback is to call **hlCheckEvents()** in between the **hlBeginFrame()** and **hlEndFrame()** calls.

**Collision Thread Events**

Because events occur in the haptic rendering threads, there is generally some latency between the occurrence of the event and when the registered event callbacks are invoked in the client thread. While this latency is not noticeable for changes to graphic rendering, it can be noticeable for certain changes to the haptic rendering. For example, if you wish to

place a haptic constraint at the current position of the proxy when a button event occurs, by the time the client thread callback is invoked the proxy will have moved from the position that it had at the time the event occurred. Placing the constraint at this time can cause a haptic jolt or kick since the position of the proxy may have moved away from the location of the constraint. Other examples where client thread callbacks may not be sufficient include drawing and painting programs which require more frequently updated position data than the client thread updates can provide running at the 30-60hz graphics rate. By subscribing to client thread motion callbacks, you would receive position data at the 100hz rate of the collision thread.

Be aware that most of the HLAPI routines are not thread safe so you must be careful which API routines you call in your collision thread event callback. You may not call **hlBeginFrame()** nor may you specify any shapes or effects in a collision thread callback. The only API routines you may call are **hlCacheGet()** in order to query the cached state at the time of the event and the **hlProxy()** routines to directly set the position of the proxy as well **hlEnable()** and **hlDisable()** with the HL_PROXY_RESOLUTION argument. The ability to disable proxy rendering and directly set the proxy position is useful in the case where you wish to place a constraint at the proxy position. By disabling proxy resolution, you can fix the proxy at the same position until you are able to place the constraint in the client thread. This prevents the haptic kick that could occur if the proxy were allowed to move before the constraint was placed. The shape manipulation sample program (see the <*OpenHaptics install directory*>/examples) provides an example of how to do this.

## Calibration

Calibration is a necessary step in using a haptic device in order to accurately obtain positions and accurately render forces. The calibration requirements vary from device to device, but HLAPI but HLAPI abstracts this by offering a convenient event callback interface coupled with a catchall function for updating calibration.

Adding calibration support to a program involves registering for two primary event callbacks to monitor the calibration state:

- HL_EVENT_CALIBRATION_INPUT, and
- HL_EVENT_CALIBRATION_UPDATE

The HL_EVENT_CALIBRATION_INPUT event notifies the program that the device requires calibration input from the user. The input required from the user is particular to the haptic device being used. For instance, for the PHANTOM Omni haptic device, the user should be prompted to place the stylus into the flashing inkwell. Once calibration input has been obtained, a secondary event named HL_EVENT_CALIBRATION_UPDATE will be reported. This notifies the program that the device is ready for calibration update as soon as the program is ready to command it. The program can then choose to either perform the calibration or to defer it. The most typical reason for deferring calibration is if the user is in the middle of performing an operation that would be interrupted if the position of the device were to instantaneously change.

A calibration update can be invoked by calling the **hlUpdateCalibration()** function. This flushes the calibration through HLAPI, and will reset the haptic rendering pipeline to avoid force rendering artifacts due to the discontinuity in device position.

The following is an example of handling the calibration events:

```
/* Register event callbacks for calibration */
hlAddEventCallback(HL_EVENT_CALIBRATION_UPDATE,
                  HL_OBJECT_ANY, HL_CLIENT_THREAD,
                   &calibrationCallback, NULL);
hlAddEventCallback(HL_EVENT_CALIBRATION_INPUT,
                   HL_OBJECT_ANY, HL_CLIENT_THREAD,
                   &calibrationCallback, NULL);

/* This function will be invoked when a calibration event occurs. This
handler notifies the user to either provide calibration input or that
calibration is about to be updated. This callback will be invoked as a
result of calling hlCheckEvents() within the main loop of the program.
*/
void HLCALLBACK calibrationCallback(HLenum event,
                   HLuint object,
                   HLenum thread,
                   HLcache *cache,
                   void *userdata)
{
  if (event == HL_EVENT_CALIBRATION_UPDATE)
  {
    std::cout << "Device requires calibration update..." << std::endl;
    hlUpdateCalibration();
  }
  else if (event == HL_EVENT_CALIBRATION_INPUT)
  {
    std::cout << "Device requires calibration input..."
   << std::endl;
  }
}
```

## Dynamic Objects

Objects which move or change shape dynamically as you touch them present unique challenges for haptic rendering. The basic proxy rendering algorithm assumes that the shape you are touching is static while the proxy and haptic device move. The algorithm assumes that the current proxy is always constrained to be outside of the object. It then is able to move the proxy towards the haptic device position while keeping it outside the surface. When a shape changes or moves, the proxy may cross the surface and end up inside the object. When this happens, it feels as if the haptic device fell through the surface.

The HLAPI haptic rendering algorithms are able to solve this problem if the rendering engine can detect when a shape has moved or changed. When the rendering engine knows that a particular shape has moved or changed, it is able to detect that the proxy has crossed

the surface and is then able to move the proxy back out of the dynamic shape. Since the calculations involved in updating the proxy position are complex, the haptic rendering engine will only apply them to shapes that it recognizes as dynamic.

If an object's position, orientation or scale changes but the shape of the object remains the same, the HLAPI will automatically recognize that the object has changed and will correctly update the proxy. HLAPI will determine if the transform of a shape has changed since the last haptic frame by reading the OpenGL matrix stacks when **hlBeginShape()** is called. It compares the matrices that it reads with those that it read for the same shape during the last frame. In order for this to work correctly, any OpenGL transformation commands that will be changing the shapes transform from frame to frame must be made before the call to **hlBeginShape()**, otherwise HLAPI will not detect that the transformation has changed. For example, the following code will not allow HLAPI to detect the transform change:

```
// draws snowman made up of three spheres
// at position specified
void drawSnowman(const double* pos)
{
   hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId);

   glPushMatrix();
   // will cause slipthrough if pos changes from frame
   // to frame!!!
   glTranslatef(pos[0], pos[1], pos[2]);

   glutSolidSphere(2.5, 30, 30);
   glTranslatef(0, 2.5, 0);
   glutSolidSphere(2, 30, 30);
   glTranslatef(0, 2, 0);
   glutSolidSphere(1, 30, 30);

   glPopMatrix();
   hlEndShape();
}
```

To fix the problem simply move the transform above the call to **hlBeginShape()**:

```
// draws snowman made up of three spheres
// at position specified
void drawSnowman(const double* pos)
{
   glPushMatrix();

   // ok to do this before hlBeginShape
   glTranslatef(pos[0], pos[1], pos[2]);

   hlBeginShape(HL_SHAPE_DEPTH_BUFFER, myShapeId);

   glutSolidSphere(2.5, 30, 30);
   glTranslatef(0, 2.5, 0); // ok, won't change
                            // between frames
   glutSolidSphere(2, 30, 30);
```

```
    glTranslatef(0, 2, 0);// ok, won't change
                          // between frames
    glutSolidSphere(1, 30, 30);

    glPopMatrix();
    hlEndShape();
}
```

For shapes where the transformation does not change but the geometry itself does, the API cannot detect the change so you must inform the API that the shape has changed. This is the case with surfaces which deform. To tell the HLAPI that the geometry of the shape will change, set the dynamic geometry hint to true before drawing the shape by calling:

```
hlHintb(HL_SHAPE_DYNAMIC_SURFACE_CHANGE, HL_TRUE);
```

After specifying the dynamic shape, it is a good idea to turn the hint off before drawing any non-dynamic shapes to avoid the extra overhead of the dynamic proxy updates for shapes that are not really dynamic.

The dynamic shape hint is not supported when using depth buffer shapes that are touchable from both the front and back faces in conjunction with the haptic camera view. In this situation, HLAPI is not able to correctly move the proxy to the dynamic surface.

## Direct Proxy Rendering

Rather than specifying shapes which will influence the position of the proxy, HLAPI allows you set the proxy position and orientation directly. In order to directly set the proxy, you must first disable proxy resolution by calling:

```
hlDisable(HL_PROXY_RESOLUTION);
```

This will stop the haptic rendering engine from updating the proxy position itself. The proxy position will remain at its current location until it is explicitly moved by the user. HLAPI will ignore any shapes that are specified when proxy resolution is disabled. To set the proxy position or orientation call:

```
HLdouble positionVector[3] = {1.0, 2.0, 3.0};
hlProxydv(HL_PROXY_POSITION, positionVector);

HLdouble orientationQuaternion[4] = { 0.7071, 0, 0, 0.7071};
hlProxydv(HL_PROXY_ROTATION, orientationQuaternion);
```

Direct proxy setting can be used if you wish to perform your own collision detection between the proxy and the geometry in the scene. The SimpleRigidBody sample program shows an example of this. It uses direct proxy rendering to implement virtual coupling between the haptic device and a body in a dynamic situation. It implements a basic rigid body simulation and sets the proxy position and orientation to follow one of the bodies in the simulation. In addition, a spring force from the haptic device position and orientation is applied to the body in the simulation. The result is that the body in the simulation acts as

the proxy. It moves to follow the haptic device position but its motion is restricted by collisions with other bodies in the simulation and these restrictions are fed back to the user by the fact that the proxy position follows the body.

Direct proxy setting is also useful in collision thread events. If you place a new surface or constraint at the proxy position in a client thread event callback, you will get force discontinuities. This happens because of the latency between the haptic rendering and client threads. However, you cannot create new shapes in a collision thread event handler. Instead, you can disable proxy resolution in the collision thread event callback which will lock the proxy in place until the next client thread frame at which point you can create the shape. The ShapeManipulation sample program uses this technique when placing the axis snap constraint (see the <*OpenHaptics Install Directory*>/examples).

## Multiple Devices

An HLAPI program can support concurrent use of multiple haptic devices. Each haptic rendering context will generate forces for only one device, so to use multiple devices, you must use a separate rendering context for each device. During program startup, first initialize ALL of the haptic devices using the HDAPI **hdInitDevice()** function. Then, create a haptic rendering context for each device instance. It is important to note that the scheduler is automatically started when a haptic rendering context is created with a valid device handle. Furthermore, haptic devices initialized after the scheduler is started will not participate in the scheduler unless the scheduler is cycled off and then on. Therefore, the recommended initialization order is to initialize all devices prior to creating their respective haptic rendering contexts.

Haptic rendering for multiple devices and contexts involves executing a separate rendering pass for each context. You must employ the use of the **hlMakeCurrent()** function in order to target rendering operations at a particular context and device. In general, this is the optimal approach, since the geometry required for each haptic rendering context is particular to the location of the respective proxy. It may also be desirable to begin a haptic frame and check for events from all contexts so that input and events can be processed for all device interactions prior to rendering new shapes and effects for each of the devices.

The HelloSphereDual sample program demonstrates concurrent use of two device instances in an HLAPI program.

# Extending HLAPI

This section contains the following topics:



## Custom Shapes

Custom shapes allow developers to extend the OpenHaptics API to create user-defined surfaces and constraints. Custom shapes are completely described through two routines: *intersect* and *closest feature*. These are specified using the **hlCallback()** routine with either the HL_SHAPE_INTERSECT_LS or HL_SHAPE_CLOSEST_FEATURE parameter. Intersect should be defined for contact shapes, closest point for constraint shapes.

## Intersect Callback

The intersect callback routine should determine if a given line segment between and including a start and end point crosses the custom shape's surface. If so, it should return the intersection point, normal at the intersection point, and face.

A shape defines its intersection callback routine through the following procedures:

```
hlCallback(HL_SHAPE_INTERSECT_LS,
     (HLcallbackProc) intersectSurface, (void *)data);

bool intersectSurface(const HLdouble startPt[3],
                      const HLdouble endPt[3],
                      HLdouble intersectionPt[3],
                      HLdouble intersectionNormal[3],
                      HLenum *face,
                      void *userdata);
```

The intersection point should determine the closest feature on the surface to the given point. That point should lie exactly on the surface. Failure to return a correct intersection point can result in slipthroughs or kicking.

The intersection normal should point directly away from the surface at the point of contact. Failure to return a correct intersection normal can result in unnatural stickiness when feeling the surface.

The face should return the sidedness of the intersection as either HL_FRONT or HL_BACK. For example, if the shape is a plane that was intersected from above the plane, face should be set to HL_FRONT and the intersection normal is the plane's normal. If that plane was intersected from below, then face should be set to HL_BACK and the intersection normal is the inverse of the plane's normal. Failure to return the correct face can result in slipthrough.

The userdata is for any additional data that the user needs to pass into the intersection routine.

The intersection routine for a sphere with radius r might look like this:

**1** Check that the start point is outside or on the sphere, i.e. that the distance between itself and the center of the sphere is greater than or equal to the radius of the sphere.

**2** Check that the end point is inside or on the sphere, i.e. that the distance between itself and the center of the sphere is less than or equal to the radius of the sphere.

**3** Determine if and where the segment intersects with the sphere. This can be done using a binary search, quadratic formula, or other techniques.

**4** If the segment intersects, calculate the normal at the intersection point. For a sphere, this is the direction of a vector from the center of the sphere to the intersection point. Normalize the vector so that it is of unit length.

**5** Set the face as HL_FRONT. If the sphere is two-sided, then repeat this process with the start and end points reversed and return HL_BACK if an intersection is detected. Reverse the intersection normal, so that it points from the intersection point to the center of the sphere.

## Closest Feature Callback

The closest feature routine should return the closest point on the surface to the query point, as well as one or more local features that approximate the surface in the vicinity of that point.

A shape defines its closest feature callback routine through the following procedures:

```
hlCallback(HL_SHAPE_CLOSEST_FEATURES,
     (HLcallbackProc) closestSurfaceFeatures,
     (void*) data);

bool closestSurfaceFeatures(const HLdouble queryPt[3],
              const HLdouble targetPt[3],
              HLgeom *geom,
              HLdouble closestPt[3],
              void *userdata);
```

The closest point should represent the point on the surface closest to the query point. Failure to return the correct closest point may result in buzzing while the surface is being used as a constraint.

The closest feature should also generate at least one local feature geometry. These are used by the API to actually feel the surface; i.e. the surface is completely represented by its local features to the API. For example, a typical local feature is a plane that passes through the intersection point and whose normal is perpendicular to the surface. If the user is touching an edge, then closestSurfaceFeatures may generate two planes - one for each wall of the edge. Failure to generate correct local features may result in slipthrough or other instability.

The userdata is for any additional data that the user needs to pass into the routine.

The closest feature routine for a sphere with radius r might look like this:

**1** Project the point onto the sphere's surface. For a sphere, this involves creating a vector from the center of the sphere to the query point, scaling it so that its length is the radius of the sphere, and then adding that vector to the center of the sphere.

**2** Generate a plane whose normal points away from the sphere at the closest point. For the sphere, this is just the vector from the center of the sphere through the closest point, normalized.

## Custom Effects

HLAPI allows for custom force effect using a callback function interface. There are three main callback functions that need to be registered for proper execution of a custom force effect. They are:

- HL_EFFECT_START
- HL_EFFECT_STOP
- HL_EFFECT_COMPUTE_FORCE

These callback functions will be invoked in the servo loop thread as entry points for starting, stopping or computing a force. These callback functions are registered using **hlCallback()**, which requires the callback type, callback function pointer, and user data as arguments. The following example shows how to register and start a custom callback effect.

```
HLuint effect = hlGenEffects(1);

/* Provide some data to be used by the callback functions */
MyEffectData myData;

hlBeginFrame();
hlCallback(HL_EFFECT_COMPUTE_FORCE, (HLcallbackProc)
      computeForceCB, &myData);
hlCallback(HL_EFFECT_START, (HLcallbackProc) startEffectCB,
      (void*)&myData);
hlCallback(HL_EFFECT_STOP, (HLcallbackProc) stopEffectCB,
      (void*)&myData);
hlStartEffect(HL_EFFECT_CALLBACK, effect);
hlEndFrame();

/* Execute main loop and stop the force effect some time later */

hlBeginFrame();
hlStopEffect(effect);
hlEndFrame();

hlDeleteEffects(effect, 1);
```

The callback functions bound to HL_EFFECT_START and HL_EFFECT_STOP will be invoked in the servo loop thread each time the effect is to be started or stopped respectively. Effects may be started and stopped during execution in response to events in the system that would disrupt the continuity of the effect force rendering, for instance, when calibration is updated or if a force error occurs. These routines should be used for initializing or cleaning up simulation state used by the force effect, such as the initial proxy position. The proxy position can be obtained from the HLcache object, which is queried using **hlCacheGet*()**. The HLcache provides access to state from the haptic rendering pipeline that may be useful for rendering the custom force effect. In addition, an HLcache container can be used to gain access to haptic rendering state from the pipeline,

such as the current proxy position. Unlike events, the HLcache container provided to custom force effects contains data in workspace coordinates, so that it can readily be used in force effect computations.”

```
/******************************************************
 Servo loop thread callback called when the effect is started
******************************************************/
void HLCALLBACK startEffectCB(HLcache *cache,
void *userdata)
{
    MyEffectData *pData = (MyEffectData *) userdata;

    fprintf(stdout, "Custom effect started\n");

    /* Initialize the anchor point to be at the proxy
       position */
    hlCacheGetDoublev(cache, HL_PROXY_POSITION,
                      pData->m_anchorPos);
}


/******************************************************
 Servo loop thread callback called when the effect is stopped
******************************************************/
void HLCALLBACK stopEffectCB(HLcache *cache, void *userdata)
{
  fprintf(stdout, "Custom effect stopped\n");
}
```

The callback function bound to HL_EFFECT_COMPUTE_FORCE is responsible for computing the effect force. A callback effect can either generate a force or it can modify the existing force computed by the haptic rendering pipeline. The current force from the pipeline is provided as the first parameter of the force callback function. In addition, an HLcache object can be used to gain access to haptic rendering state from the pipeline, such as the proxy position. Furthermore, HDAPI accessors can be used to query state and properties about the haptic device, such as device position, velocity, nominal max stiffness, nominal max force, etc. The following example code demonstrates computing a sawtooth drag effect, where an anchor point is moved whenever a spring force threshold is exceeded.

```
/******************************************************
 Servo loop thread callback for computing a force effect
******************************************************/
void HLCALLBACK computeForceCB(HDdouble force[3],
               HLcache *cache,
               void *userdata)
{
  MyEffectData *pData = (MyEffectData *) userdata;

   /* Get the current proxy position from the state cache.
      Note: the effect state cache for effects is
      maintained in workspace coordinates, so its data can
      be used without transformation for computing forces.*/
  hduVector3Dd currentPos;
  hlCacheGetDoublev(cache, HL_PROXY_POSITION, currentPos);

   /* Use HDAPI to access the nominal max stiffness for the haptic
      device */
   HDdouble kStiffness;
   hdGetDoublev(HD_NOMINAL_MAX_STIFFNESS, &kStiffness);

   /* Compute a spring force between the current proxy
      position and the position of the drag point. */
   hduVector3Dd myForce = kStiffness *
         (pData->m_anchorPos - currentPos);

  /* Update the drag point if force exceeds the drag
      threshold */
  static const HDdouble kDragThreshold = 1.0;
  if (myForce.magnitude() > kDragThreshold)
  {
    pData->m_dragPos = currentPos;
  }

  /* Accumulate our computed force with the current force
      from the haptic rendering pipeline. */
  force[0] += myForce[0];
  force[1] += myForce[1];
  force[2] += myForce[2];
}
```

Since callback effects are executed in the servo loop thread, it is important not to modify the user data from outside the servo loop thread. If a change needs to be made to user data in use by the callback effect, then it is recommended to use the **hdScheduleSynchronous()** callback mechanism from HDAPI for modifying that state.

## Integrating HDAPI and HLAPI

The HDAPI scheduler can be used alongside HLAPI to control the haptic device at runtime. One particularly useful technique is to schedule two asynchronous callbacks, one with max scheduler priority and the other with min scheduler priority. Since HLAPI allows for **hdBeginFrame() / hdEndFrame()** calls to be nested, this can allow a client to take ownership of the overall HDAPI frame boundaries without disturbing HLAPI. This enables an HDAPI program to utilize HLAPI for its proxy rendering and impedance control facilities yet still have ultimate control over the forces that get commanded to the haptic device, since the last **hdEndFrame()** will actually commit the force to the haptic device.

When using HLAPI and HDAPI together, error handling should be deferred to HLAPI's **hlGetError()** once the haptic rendering context has been created. HLAPI propagates errors from HDAPI, so that they can be handled through the same code path.

# 7

# Utilities

This chapter includes information about some of the utilities shipped with the OpenHaptics toolkit, including the following sections:

# Vector/Matrix Math

The HD utilities include basic 3D vector and matrix math. Vector utilities include common operations such as dot products and cross products, as well as basic algebra. Matrix operations include basic transformations.

## Vector Utilities

The <HDU/hduVector.h> header exposes a simple API for common vector operations in three dimensional space. A brief description of the functions follows:

**Default constructor**

```
hduVector3Dd vec1;
vec1.set(1.0, 1.0, 1.0);
```

**Constructor from three values**

```
hduVector3Dd vec2(2.0, 3.0, 4.0);
```

**Constructor from an array**

```
HDdouble x[3] = {1.0, 2.0, 3.0};
hduVector3Dd xvec = hduVector3Dd(x);
```

**Assignment**

```
hduVector3Dd vec3 = hduVector3Dd(2.0, 3.0, 4.0);
```

**Usual operations:**

```
vec3 = vec2 + 4.0* vec1;
```

**Magnitude:**

```
HDdouble magn = vec3. magnitude();
```

**Dot product:**

```
HDdouble  dprod = dotProduct(vec1, vec2);
```

**Cross product:**

```
hduVector3Dd vec4 = crossProduct(vec1, vec2);
```

**Normalize:**

```
vec4.normalize();
```

## Matrix Utilities

The <HDU/hduMatrix.h> header exposes a simple API for common matrix operations. A brief description of the functions follows:

**Default constructor**

```
hduMatrix mat1;  // the identity matrix by default

HDdouble a[4][4] = {
                    {a1,a2,a3,a4},
                    {a5,a6,a7,a8},
                    {a9,a10,a11,a12},
                    {a13,a14,a15,a16}
                   };
mat1.set(a);
```

**Constructor from sixteen values**

```
hduMatrix mat(a1,a2,a3,a4,a5,a6,a7,a8,
              a9,a10,a11,a12,a13,a14,a15,a16);
```

**Constructor from an array**

```
HDdouble a[4][4] =  {
                    {a1,a2,a3,a4},
                    {a5,a6,a7,a8},
                    {a9,a10,a11,a12},
                    {a13,a14,a15,a16}
                    };
hduMatrix mat2(a);
```

**Assignment**

```
hduMatrix mat3 = mat2;
```

**Get values**

```
double vals[4][4];
mat3.get(rotVals);
```

**Usual operations**

```
mat3 = mat2 + 4.0 * mat1;
```

**Invert**

```
mat3 = mat2.getInverse();
```

**Transpose:**

```
mat3 = mat2.transpose();
```

**Create a rotation**

```
hduMatrix rot;
rot = createRotation(vec1, 30.0*DEGTORAD);
HDdouble rotVals[4][4];
rot.get(rotVals);
glMultMatrixd((double*)rotVals);
```

The HDAPI defines both a float and double vector although other types can be created by the developer using the same methodologies.

hduVector3Dd and hduMatrix can both be used in **glGetDoublev()** functions. For example, the following are legal:

```
hduVector3Dd vector;
HDdouble array[3];
hdGetDoublev(HD_CURRENT_POSITION,vector);
hdGetDoublev(HD_CURRENT_POSITION,array);
```

They may also be used in the HLAPI **hdGetDoublev()** and **hdSetDoublev()** functions.

## Workspace to Camera Mapping

One of the challenging aspects of graphics programming are transformations among coordinate systems. The haptic device introduces yet another coordinate system, which we will refer to as “Workspace System.” Both the HLAPI and the HDAPI provide a generic way to query the dimensions of the Workspace System as shown below:

Using HDAPI:

```
HDdouble aUsableWorkspace[6];
HDdouble aMaxWorkspace[6];
hdGetDoublev(HD_USABLE_WORKSPACE_DIMENSIONS,
             aUsableWorkspace);
hdGetDoublev(HD_MAX_WORKSPACE_DIMENSIONS,
             aMaxWorkspace);
```

The HD_MAX_WORKSPACE_DIMENSIONS option returns the maximum extents of the haptic device workspace. However, due to the mechanical properties of the device, it is not guaranteed that forces can be reliably rendered in all that space.

However, using the HD_USABLE_WORKSPACE_DIMENSIONS results in a parallelepiped where it is guaranteed that forces are rendered reliably. It is clear that this, in general, is a subset of the max reachable space.

HLAPI also provides a generic way to query the dimensions of the Workspace System, as shown below:

```
HLdouble workspaceDims[6];
HLdouble maxWorkspaceDims[6];
hlGetDoublev(HL_WORKSPACE, maxWorkspaceDims);
hlGetDoublev(HL_MAX_WORKSPACE_DIMENSIONS, workspaceDims);
```

The HL_WORKSPACE dimension option returns current extents of the haptic workspace. HL_WORKSPACE is set-able by the user. By default it is set to HL_MAX_WORKSPACE_DIMENSIONS. HL_WORKSPACE is similar in function to GL_VIEWPORT in OpenGL.

The HL_MAX_WORKSPACE_DIMENSIONS option returns the maximum extents of the haptic workspace. HL_MAX_WORKSPACE_DIMENSIONS is similar in function to GL_MAX_VIEWPORT_DIMS.

From the graphics side, you assign a geometric primitive's vertices in an arbitrary coordinate system (the “world” system), and in general, transform those coordinates with a modelview matrix, to convert to what is called the eye system.

In the eye coordinate system certain points and vectors have a very simple form. For example, the eye is placed in the origin of the “eye” coordinate system, and the view vector points in the negative-z direction. In this coordinate system you then define a projection matrix, which defines a view frustum. You can define an orthographic (also called parallel) projection, where the view frustum is a parallelepiped. Alternatively, you can define a perspective projection, where the view frustum is a cut prism. In defining both projections, you have to define a near and far plane, which correspond to the range of z-coordinates in the eye space that are enclosed in the corresponding frusta.

A problem that programmers often face in integrating haptics with graphics is how to “align” the workspace system with the view frustum. Such a mapping is needed in order for the user to see what he/she feels (and vice versa). There are a few options for you to consider. An important question is which subset of the view frustum is “touchable?” You may want to allow the user to touch only within a range of z-values (in eye coordinates). In general, you will usually want to map a subset of the visible depth range onto the workspace z axis.

As a part of both the HD Utilities library (HDU) and the HL Utilities library (HLU) we provide functions that can be used to facilitate this mapping. We also provide source code, which can be used as a starting point for programmers who want to modify the details of the mapping. The following example shows the generation of a matrix using the HDU library used to render the end-effector position.

```
HDdouble workspacemodel[16];
HDdouble screenTworkspace;
GLdouble modelview[16];
GLdouble projection[16];
GLint viewport[4];

glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
glGetDoublev(GL_PROJECTION_MATRIX, projection);
hduMapWorkspaceModel(modelview, projection,
                     workspacemodel);
```

The hduMapWorkspaceModel call will assign the workspacemodel matrix. You can subsequently apply this matrix in the OpenGL matrix stack (as a last operation):

```
glMultMatrixd(workspacemodel);
```

Doing so will allow you to directly display the end-effector position on the screen, without transforming the end-effector coordinates.

The HLU library provides a similar function to facilitate mapping between the model and the workspace. hluModelToWorkspaceTransform generates a matrix that transforms from model to workspace coordinates.

```
HLdouble modelview[16];
HLdouble viewtouch[16];
HLdouble touchworkspace[16];
HLdouble modelworkspace[16];

glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
hlGetDoublev(HL_VIEWTOUCH_MATRIX, viewtouch);
hlGetDoublev(HL_TOUCHWORKSPACE_MATRIX, touchworkspace);
hluModeltoWorkspaceTransform(modelview, viewtouch,
                             touchworkspace,
                             modelworkspace);
```

Matrix storage is compatible with OpenGL column major 4X4 format. An hduMatrix can be constructed from the matrix linear array read from OpenGL. This array can also be accessed using the array cast operator for hduMatrix. For example, the above HDU example could have used:

```
hduMatrix workspacemodel;
hduMatrix modelview;
hduMatrix projection;
```

## Snap Constraints

The Snap Constraint library provides classes that can be used to implement simple constraints. The basic architecture includes a basic SnapConstraint class and derived PointConstraint, LineConstraint, PlaneConstraint classes. Furthermore, we provide a CompositeConstraint class, also derived from SnapConstraint, which allows the user to easily combine multiple constraints.

The basic functionality is provided by SnapConstraint::testConstraint function, which, for a given test point it calculates the proxy position a point that respects the constraint, and is as close as possible to the test point. The prototype of this function is:

```
virtual double testConstraint(const hduVector3Dd &testPt,
                              hduVector3Dd &proxyPt);
```

## C++ Haptic Device Wrapper

The Haptic Device wrapper provides a convenient encapsulation of common state and event synchronization between the haptic and graphic threads. The implementation allows for event callbacks to be registered for a number of common state transitions, like button presses, making and losing collision contact and device errors. Callback functions can be registered for invocation in both the haptic and graphics threads. The haptic thread detects the event, such as a button press, and then the event can be handled in either the haptic thread, graphics thread or both. In addition, state is managed such that a snapshot of state is provided along with each event. This is very useful when programming interactions that have haptic and graphic components, since it's important to be dealing with the exact same event related state in both threads. For instance, the position of the haptic device when the button is pressed is an important piece of state that should be consistent for both threads, or else the haptic response and the graphic response to the button press will be spatially out of sync.

The Haptic Device Wrapper is structured as two implementations of the IHapticDevice interface. Derived classes HapticDeviceHT and HapticDeviceGT encapsulate state management for the haptic thread and graphics thread respectively.

An example setup scenario is provided below:

```
/* Create the IHapticDevice instances for the haptic and graphic threads */
m_pHapticDeviceHT = IHapticDevice::create(
    IHapticDevice::HAPTIC_THREAD_INTERFACE, m_hHD);
m_pHapticDeviceGT = IHapticDevice::create(
    IHapticDevice::GRAPHIC_THREAD_INTERFACE, m_hHD);

/* Setup a callback for button 1 down and up for the graphics thread */
m_pHapticDeviceGT->setCallback(
    IHapticDevice::BUTTON_1_DOWN, button1EventCallbackGT, this);
m_pHapticDeviceGT->setCallback(
    IHapticDevice::BUTTON_1_UP, button1EventCallbackGT, this);
```

In order for state and/or an event to propagate from the device, through the haptics thread and into the graphics thread, the IHapticDevice::beginUpdate and IHapticDevice::endUpdate methods need to be called as part of a main loop in both threads. The HapticDeviceHT instance can be updated within the haptic thread using an asynchronous callback scheduled with the HDAPI scheduler. The HapticDeviceGT instance can be updated in the graphics thread (or application thread) at the appropriate place where other events are dispatched. For instance, this might be appropriate as part of the rendering loop, idle processing or whatever periodic callback mechanism is available.

The following update call with the graphics thread will synchronize state and events from the haptics thread to the graphics thread.

```
/* Capture the latest state from the haptics thread. */
m_pHapticDeviceGT->beginUpdate(m_pHapticDeviceHT);
m_pHapticDeviceGT->endUpdate(m_pHapticDeviceHT);
```

Calling the above update routines will flush pending events from the haptics thread to the graphics thread and dispatch them. In addition, calling m_pHapticDeviceGT->**getCurrentState()** or m_pHapticDeviceGT->**getLastState()** will provide a snapshot of useful haptic rendering related state that is maintained by the HapticDevice class.

Below is an example of an IHapticDevice event handler for a button press.

```
/******************************************************************
 This handler gets called in the graphics thread whenever a button
press is detected. Initiate a manipulation at the button press
location.
******************************************************************/
void HapticDeviceManager::button1EventCallbackGT(
    IHapticDevice::EventType event,
    const IHapticDevice::IHapticDeviceState * const pState,
    void *pUserData)
{
    HapticDeviceManager *pThis = static_cast<HapticDeviceManager *>
                        (pUserData);

    if (event == IHapticDevice::BUTTON_1_DOWN)
    {
        assert(!pThis->isManipulating());
        pThis->startManipulating(pState->getPosition());
    }
    else if (event == IHapticDevice::BUTTON_1_UP)
    {
        assert(pThis->isManipulating());
        pThis->stopManipulating(pState->getPosition());
    }
}
```

Please refer to the PointSnapping or PointManipulation examples within the HD graphics examples directory for more information.

## hduError

The hduError provides some extra error handling utilities. Functions of interest include:

- **hduPrintError(FILE *stream, const HDErrorInfo *error, const char *message)**, which pretty-prints extended error information.
- **HDboolean hduIsForceError(const HDErrorInfo *error)**, which is a convenience function that allows for easy determination on whether one has encountered a force error.
- **hduIsSchedulerError()**, which can test for errors related to the scheduler.

## hduRecord

hduRecord is a logging tool used for recording data at servo loop rates. The utility gathers data for a specified number of servo loop ticks, then writes the data to the specified file. By default, hduRecord captures the device position, velocity, and force for each servo loop tick. A callback can also be specified to add additional data in the form of a string. This callback is called every tick and its contents appended to the data for that tick. For example, the callback could append transform information, error state, etc.

For example, here is a typical use of hduRecord to log the force, position, velocity, and gimbal angles for 5000 ticks.

```
char *recordCallback(void *pUserData)
{
    hduVector3Dd gimbalAngles;
    hdGetDoublev(HD_CURRENT_GIMBAL_ANGLES,gimbalAngles);
    char *c = new char[200];
    sprintf(c,"%lf %lf %lf", gimbalAngles[0],
                             gimbalAngles[1],
                             gimbalAngles[2]);
    return c;
}

FILE *pFile =
          fopen("c:\\temp\\recordServoLoopData.txt","w");
hdStartRecord(pFile,recordCallback,NULL,5000);
```

## Haptic Mouse

The haptic mouse utility library can be used alongside HLAPI to emulate 2D mouse input. This allows your haptically enabled program to obviate the need for a 2D mouse, since the haptic device can now be used for selecting items from menus, toolbars, and general interaction with a windows GUI. Introducing the haptic mouse to a program that's already using HLAPI only requires a few additional lines of code!

### Setup

The haptic mouse utility can either share an existing haptic rendering context, or it can be used stand-alone with a dedicated haptic rendering context. The haptic mouse also requires one main window, which is used for mapping from the viewport to the screen as well as control over transitioning the mouse between its active and inactive state. The haptic mouse also requires the viewing transforms from OpenGL for mapping from the 3D coordinates of the model space to the 2D coordinates of the viewport. This allows the absolute 3D position of the haptic device to be mapped to the 2D screen to control the mouse cursor and makes for a seamless transition between moving the 3D and 2D cursor on the screen. Lastly, a render function for the haptic mouse must be called periodically within an HL frame scope so that the haptic mouse can render haptics, such as the ambient

friction effect when the haptic mouse is active. The program must also periodically call **hlCheckEvents()** for the haptic rendering context, since the haptic mouse registers a client thread motion event, which is used for monitoring transitions.

The haptic mouse minimally requires 4 function calls to be added to any HLAPI program.

```
HLboolean hmInitializeMouse(HHLRC hHLRC,
                            const char *pClassName,
                            const char *pWindowName);
```

Call this function to initialize the haptic mouse utility. The haptic mouse utility is a singleton, so it can only be used to emulate one mouse cursor. The utility requires access to an initialized haptic rendering context and the main top-level window for the application. The haptic mouse utilizes the HLAPI haptic rendering context for monitoring collision thread motion events and button presses. It also utilizes HLAPI for rendering a simple haptic effect while the mouse is active. Additionally, the haptic mouse requires access to the top-level window for the application for mapping to the screen and providing additional state used for transitioning. For Win32, the class name and window name can be obtained using the following two Win32 functions: **GetClassName()** and **GetWindowText()**. The benefit of an approach like this is that it's readily portable, particularly when used with platform independent toolkits like GLUT and GLUI.

```
void hmShutdownMouse();
```

Call this function as part of application shutdown, before the haptic rendering context is deleted.

```
void hmSetMouseTransforms(const GLdouble modelMatrix[16],
                          const GLdouble projMatrix[16],
                          const GLint viewport[4]);
```

Provide haptic mouse with the 3D transforms used by the main view of the application. These should be the same transforms used by the view in which the 3D haptic cursor is displayed. These transforms provide a mapping from the world coordinate space of the application to window coordinates of the viewport. These transforms should be provided to the haptic mouse utility whenever the view is reshaped, the projection transform changes or the camera's modelview transform changes.

```
void hmRenderMouseScene();
```

The haptic mouse utility needs to be called within the **hlBeginFrame()** / **hlEndFrame()** scope of the haptic rendering context. This allows it to perform its own haptic rendering, such as starting or stopping an ambient friction force effect. The haptic mouse also utilizes this call as the primary hook for transitioning between active and inactive state.

## Haptic Mouse Transitioning

The default behavior of the haptic mouse is to automatically transition from inactive to active whenever the 3D cursor leaves the viewport. As soon as the 3D cursor moves outside of the viewport, a 2D mouse cursor will appear at that location and proceed to track the motion of the haptic device. When the 2D cursor moves back into the viewport, the haptic mouse will automatically discontinue mouse emulation and deactivate itself. The active state of the haptic mouse can always be queried using the **hmIsMouseActive()** function. This can be used, for instance, to skip haptic rendering of other shapes and effects in the scene, since they might interfere with the movement of the haptic mouse. Transitioning will only occur when the main window is in the foreground. This allows you to bring another application to the foreground and move over the region of the screen containing the original viewport without transitioning. In addition, transitions will be avoided while the mouse is captured. This prevents undesirable transitions while dragging with the mouse, invoking a drop down menu, or other interactions that temporarily capture mouse input. The mouse transition behavior can be modified using the **hmEnable() and hmDisable()** calls with the following two mode enums:

- HM_MOUSE_INSIDE_VIEWPORT, and
- HM_MOUSE_OUTSIDE_VIEWPORT

These modes control whether the haptic mouse is allowed inside or outside of the viewport respectively. As mentioned above, the default behavior is that HM_MOUSE_INSIDE_VIEWPORT is disabled and HM_MOUSE_OUTSIDE_VIEWPORT is enabled.

## Customized Workspace Mapping

In some cases, it is desirable to change the workspace allocation to enable sufficient motion outside of the viewport. This is achieved by modifying the workspace used by the haptic rendering context via a call to **hlWorkspace()** prior to setting the TOUCHWORKSPACE transform of the context. The easiest way to think of this allocation is that the size and position of the viewport relative to the parent window should match the size and positioning of the context's workspace relative to the haptic device's overall workspace in X and Y. Determining new coordinates for the context's workspace involves the following steps:

1. Save off the initial workspace
2. Use the 2D window coordinates of the viewport within the main window to compute normalized coordinates.
3. Apply the normalized coordinates of the viewport to the initial workspace to determine the corresponding coordinates within the workspace. You should only need to compute the update workspace coordinates for the X and Y dimensions, since we want to preserve the Z dimension of the original workspace.
4. Call **hlWorkspace()** with these modified workspace coordinates and update the TOUCHWORKSPACE transform using one of the HLU fit workspace routines.

## Additional resources

Please refer to the HapticMaterials example (see the <install directory>/examples) for a basic use case of the haptic mouse. Also, the full source code for the haptic mouse utility is provided in the utilities directory of the OpenHaptics toolkit. Feel free to tailor the utility to the needs of your application. For instance, it is possible to render extruded geometry that corresponds to 2D controls in the window or planes aligned with the sides of the view volume and window. This makes the haptic mouse an even more potent replacement for your plain old windows mouse.

# 8

# Deploying OpenHaptics Applications

The Open Haptics toolkit exposes a programmatic interface to the Haptic Device (HD) abstraction layer runtime. Developers can use the toolkit to haptically enable their applications, and can distribute the runtime to third parties that run these applications (provided that they have purchased the right to do so). It is therefore important for an SDK developer to understand the run-time configuration of the underlying library, as well as the API that the SDK provides.

## Run-time configuration

The toolkit installation installs Dynamically Linked Libraries (DLLs) in the Windows System Directory (e.g. C:\Windows\System32). Those libraries are the PHANToMIOLib42.dll, hl.dll, and the hd.dll. The PHANToMIOLib42.dll is a private low-level library that is statically linked with the hl.dll and hd.dll, and therefore needs to be present on the target system. For more information see the *OpenHaptics Installation Guide*.

## Deployment Licensing

Deployment licenses work by requiring the application writer to supply vendor and application strings along with a password. The password is generated by SensAble and is validated at run-time by OpenHaptics.

The following APIs are used

```
/* Licensing */
HLAPI HLboolean HLAPIENTRY hlDeploymentLicense(const char* vendorName,
                                               const char* applicationName,
                                               const char* password);

HDAPI HDboolean HDAPIENTRY hdDeploymentLicense(const char* vendorName,
                                               const char* applicationName,
                                               const char* password);
```

Sample Code:

```
<<<<< Start code sample

char*  vendorName = "ABC Software, Inc.";  /*  Vendor name */
char*  appName    = "ABC Haptics";         /*  Vendor application name */

char*  hdPassword   =".........."; /* supplied by Sensable*/
char*  hdPassword   =".........."; /* supplied by Sensable*/
HDboolean  hdLicenseStatus;
HLboolean  hlLicenseStatus;

hdLicenseStatus = hdDeploymentLicense(vendorName,
                                      appName,
                                      hdPassword);
if(hdLicenseStatus == false)
{
  /*  handle HD license failure */
}


hlLicenseStatus = hlDeploymentLicense(vendorName,
                                      appName,
                                      hdPassword);

                                      hdPassword);
if(hdLicenseStatus == false)
{
  /*  handle HL license failure  */
}

>>>>> End   code sample
```

See the HL Console Deployment Example program for a complete code sample. A guide to all the installed Source Code Examples can be found in <*OpenHaptics Install directory*>/doc.

# 9

# Troubleshooting

This chapter contains troubleshooting information about the following topics:

## Device Initialization

If the device has trouble initializing, use the “PHANToM Test” application to ensure that the device is properly hooked up and functioning. Check that the string name used to identify the device is spelled correctly.

PHANToM Test is installed with the PHANTOM Device Driver and can be found in the <*SensAble install directory*>/PHANTOM Device Drivers.

## Frames

Each scheduler tick should contain at most one frame per device, when HD_ONE_FRAME_LIMIT is enabled. Frames need to be properly paired with an **hdBeginFrame()** and **hdEndFrame()** of an active device handle. Frame errors include:

- Setting state for a device, outside a frame for that device.
- Having more than one frame for a particular device within a single scheduler tick, if HD_ONE_FRAME_LIMIT is enabled.
- Mismatched begin/end for the same device.

Multiple frames can be enabled by turning off HD_ONE_FRAME_LIMIT. This is not recommended, however, since resulting behavior is likely to be device-dependent. Generally only one call to set motor/torques should be issued per tick, and all haptics operations should be encapsulated by the single begin/end. If in doubt, a simple way to accomplish this is to schedule a begin as the first operation that happens in a tick, and an end as the last, as shown in the following:

```
HDCallbackCode HDCALLBACK beginFrameCallback(void *)
{
    hdBeginFrame(hdGetCurrentDevice());
    return HD_CALLBACK_CONTINUE;
}
HDCallbackCode HDCALLBACK endFrameCallback(void *)
{
    hdEndFrame(hdGetCurrentDevice());
    return HD_CALLBACK_CONTINUE;
}

hdScheduleAsynchronous(beginFrameCallback,
                       (void*)0,
                       HD_MAX_SCHEDULER_PRIORITY);
hdScheduleAsynchronous(endFrameCallback,
                       (void*)0,
                       HD_MIN_SCHEDULER_PRIORITY);
```

## Thread Safety

Since the HDAPI runs a fast, high priority thread, care must be taken to ensure that variable and state access is thread-safe. One common design error when using the HDAPI is to share variables between the application and scheduler thread. Since the scheduler thread is running at a much faster rate than the application thread, it is possible that changes in state in the servo loop thread may not be caught by the application thread, or the state may be changing while the application thread is processing it.

For example, suppose the user creates an application that waits for a button press and then terminates. The application creates a scheduler operation that sets a button state variable to true when the button is pressed, and the user waits for a change in state.

```
/* This is a scheduler callback that will run every servo loop tick
and update the value stored in the button state variable */
HDCallbackCode HDCALLBACK queryButtonStateCB(void *userdata)
{
   HDint *pButtonStatePtr = (HDint *) userdata;
   hlGetIntegerv(HD_CURRENT_BUTTONS, pButtonStatePtr);
   return HD_CALLBACK_CONTINUE;
}

HDint nCurrentButtonState = 0;
HDint nLastButtonState = 0;

HDSchedulerHandle hHandle = hdScheduleAsynchronous(
                            queryButtonCB,
                            &nCurrentButtonState,
                            HD_DEFAULT_SCHEDULER_PRIORITY);


/* Here's an example of INCORRECTLY monitoring button press
transitions */
while (1)
{
    if ((nCurrentButtonState & HD_DEVICE_BUTTON_1) != 0 &&
        (nLastButtonState & HD_DEVICE_BUTTON_1) == 0)
    {
        /* Do something when the button is depressed */
    }

    nLastButtonState = nCurrentButtonState;

    /* Now sleep or do some other work */
    Sleep(1000);
}
```

There are two problems with this example:

First, the button state variable is being shared between the button callback and the application thread. This is not thread safe because the application might query the button state while it is being changed in the servo loop thread.

Second, since the application thread is periodically (relative to servo loop rates) querying the button state, it is possible for the application to completely miss a state change. Consider, for example, that the user might be able to press the button and then release between when the application is checking button states.

The correct methodology is to use an asynchronous call to check for button state transitions. Once a transition is detected, that information can be logged for whenever the user queries it. The user should then use a synchronous call to query that information so that it is thread safe. An example of a thread-safe application that exits after a button state transition is detected follows.

```
HDboolean gButtonDownOccurred = FALSE;

HDCallbackCode HDCALLBACK monitorButtonStateCB(void *userdata)
{
    HDint nButtons, nLastButtons;

    hdGetIntegerv(HD_CURRENT_BUTTONS, &nButtons);
    hdGetIntegerv(HD_LAST_BUTTONS, &nLastButtons);

    if ((nButtons & HD_DEVICE_BUTTON_1) != 0 &&
        (nLastButtons & HD_DEVICE_BUTTON_1) == 0)
    {
        gButtonDownOccurred = TRUE;
    }

    return HD_CALLBACK_CONTINUE;
}

HDCallbackCode HDCALLBACK queryButtonStateCB(void *userdata)
{
    HDboolean *pButtonDown = (HDboolean *) userdata;

    *pButtonDown = gButtonDownOccurred;

    /* We sampled the button down, so clear the flag */
    gButtonDownOccurred = FALSE;

    return HD_CALLBACK_DONE;
}

/* Here's an example of CORRECTLY monitoring button press
transitions*/

HDSchedulerHandle hHandle = hdScheduleAsynchronous(
    queryButtonCB, &monitorButtonStateCB,
    HD_DEFAULT_SCHEDULER_PRIORITY);
```

```
while (1)
{
    HDboolean bButtonDown;
    hdScheduleSynchronous(queryButtonStateCB, &bButtonDown,
        HD_DEFAULT_SCHEDULER_PRIORITY);

    if (bButtonDown)
    {
        /* Do something when the button is depressed */
    }

    /* Now sleep or do some other work */
    Sleep(1000);
}
```

## Race Conditions

Race conditions are a type of threading error that occurs in environments where an operation needs to happen in a specific sequence but that order is not guaranteed. This is particularly an issue in multi threaded environments if two threads are accessing the same state or depending on results from each other. Race conditions are sometimes difficult to diagnose because they may happen only intermittently. Following guidelines for thread safety is the best way to prevent race conditions. Ensure that different threads are not executing commands that are order-dependent, and that data is shared safely between the scheduler and application.

## Calibration

Calibration of the haptic device must be performed in order to obtain accurate 3D positional data from the device as well as to accurately render forces and torques. If the motion of the haptic device on the screen doesn't seem to match the physical motion of the device, or if the forces don't seem to correspond to the contacted geometry, then it is likely that calibration is at fault. It is best to ensure the device is properly calibrated by running the “PHANToM Test” program, located in the “SensAble\PHANTOM Device Drivers” directory. Calibration will persist from session to session, but may be invalidated inadvertently if the unit is unplugged or the control panel configuration is changed.

Both HDAPI and HLAPI provide facilities for managing calibration at runtime. It is advised to not programmatically change calibration until the program is in a safe state to do so. For instance, the user may be in the middle of performing a haptic manipulation that would be interrupted if the calibration instantaneously changed. When changing calibration with HDAPI, it is safest to ensure that either force output is off, via a call to **hdDisable(HD_FORCE_OUTPUT)** or that force ramping is enabled, via a call to **hdEnable(HD_FORCE_RAMPING)**. Either of these features will mitigate the possibility of unintentional forces being generated due to the instantaneous change in

device calibration. HLAPI provides its own handling for calibration changes, which automatically resets the haptic rendering pipeline following a call to **hlUpdateCalibration()**.

## Buzzing

Buzzing refers to high frequency changes in force direction. For example, the following scheduler operation will simulate buzzing along the X axis:

```
HDCallbackCode buzzCallback(void *data)
{
    static int forceDirection = 1;
    forceDirection *= -1;
    hdBeginFrame(hdGetCurrentDevice());
    hdSetDoublev(HD_CURRENT_FORCE,
                 hduVector3Dd(forceDirection*3,0,0));
    hdEndFrame(hdGetCurrentDevice());
    return HD_CALLBACK_CONTINUE;
}
```

Buzzing is sometimes only audible, but depending on its magnitude and frequency it may also be felt. Unintentional buzzing can be caused by any variety of force artifacts. Often it is a result of the device attempting to reach some position but never getting there in a stable fashion. For example, in a standard gravity well implementation, the device is attracted to the center of the well when in its proximity:

```
hduVector3Dd position;
hdGetDoublev(HD_CURRENT_POSITION,position);
hduVector3Dd gravityWellCenter(0,0,0);
const float k = 0.6;
hduVector3Dd forceVector =
                        (gravityWellCenter-position)*k;
```

As k is increased, the gravity well becomes progressively unstable. Although it will always attract the device towards its center, it is likely to overshoot. If the force magnitude is consistently so high that the device is constantly overshooting the center, then buzzing will occur as the device continues to try to settle into the center.

Buzzing for shape based collisions can also occur if the k value, or stiffness, is set too high. Recall that shape based collisions generally use the F=kx formula, where k is the stiffness and x is the force vector representing the penetration distance into the object. If k is large, then the device may kick the user out of the shape as soon as the user even lightly comes in contact with the shape. If the user is applying some constant force towards the object, then the device can get into an unstable situation where in every iteration it is either giving the user a hard push out of the object or doing nothing. This will manifest as buzzing or kicking.

For example, the following plane implementation would cause buzzing or kicking because as soon as the user touches the plane, a relatively large force kicks him off of it:

```
hduVector3Dd position;
hdGetDoublev(HD_CURRENT_POSITION,position);
if (position[1] < 0)
{
    hdSetDoublev(HD_CURRENT_FORCE,hduVector3Dd(0,5,0));
}
```

Buzzing therefore often suggests that the current force calculation is causing the device to not be able to settle into a stable position.

Buzzing can also be caused by unstable computation of the position used for the position control. For example, there may be situations where the device attempts to get to a location but cannot settle on it, and instead reaches a metastable state. Consider the following force effect example. This creates a drag between the current position and a lagging object that attempts to minimize the distance between itself and the device position. Since the step size is large, the object will continue to oscillate around the device position while never reaching it, and the user will experience some humming or buzzing:

```
HDCallbackCode draggerCallback(void *)
{
    static hduVector3Dd lastPosition(0,0,0);
    hduVector3Dd position;
    hdGetDoublev(HD_CURRENT_POSITION,position);
    hduVector3Dd lastToCurrent = position-lastPosition;
    lastToCurrent.normalize();
    lastPosition += lastToCurrent;
    hdSetDoublev(HD_CURRENT_FORCE,lastPosition-position);
    return HD_CALLBACK_CONTINUE;
}
```

The correct way to implement this example is:

```
HDCallbackCode draggerCallback(void *)
{
    static hduVector3Dd lastPosition;
    const HDdouble stepSize = 1.0;
    hduVector3Dd position;
    hdGetDoublev(HD_CURRENT_POSITION,position);
    hduVector3Dd lastToCurrent = position-lastPosition;
    if (lastToCurrent.magnitude() > stepSize)
    {
        lastToCurrent.normalize();
        lastPosition += lastToCurrent*stepSize;
    }
    else
    {
        lastPosition = position;
    }
    hdSetDoublev(HD_CURRENT_FORCE,lastPosition-position);
}
```

There are several ways of debugging and mitigating buzzing. Each is described in further detail in the following pages.

- Check that the scheduler is not being overloaded.
- Scale down forces
- Check the position of the device
- Check force calculations
- Add damping/smoothing
- Detect and abort

**Check that the scheduler is not being overloaded.**

The scheduler should be running at a minimum of 1000 Hz for stability. It is possible on some devices to run as low as 500 Hz without force artifacts, but this is not recommended. Check that scheduler operations are not exceeding their allotted time. **hdGetSchedulerTimeStamp()** can be used to time how long the scheduler operations are taking; for example, schedule a callback that happens near the end of the scheduler tick and checks if the time has exceeded 1ms. In general, scheduler callbacks should take significantly less time than that allotted by the scheduler rate. The scheduler itself consumes some small amount of fixed processing time for its own internal operations per tick, and the application also needs to have some time for whatever functionality it is performing. If the scheduler operations are cumulatively taking too much time, consider amortizing some operations or having them run at a lower frequency than once every tick. Scheduler operations should not take more than the servo loop time slice to complete.

The following illustrates a method for checking that the scheduler is not being overloaded:

```
HDCallbackCode dutyCycleCallback(void *data);
{
    double timeElapsed = hdGetSchedulerTimeStamp();
    if (timeElapsed > .001)
    {
        assert(false && "Scheduler has exceeded 1ms.");
    }
    return HD_CALLBACK_CONTINUE;
}

main()
{
hdScheduleAsynchronous(dutyCycleCallback,
                       (void*)0,
                       HD_MIN_SCHEDULER_PRIORITY);
}
```

Buzzing can also be mitigated by running the scheduler at higher rates. Typically, the more frequent the force calculations are being updated, the smaller the amplitude of the buzzing.

**Scale down forces**

Scaling down the magnitude of forces will decrease the amplitude of the buzzing. This can be done globally or per effect, for example if only one force effect is problematic. Most forces are modeled using Hooke's Law, F=kx, so turning down k effectively diminishes the force contribution. As a general guideline, k should be kept below the HD_NOMINAL_MAX_STIFFNESS value. This value is device-dependant so one way to model forces would be to scale down the overall force based on the max stiffness value compared to the one used when the application was created.

For example, suppose an application was originally developed on the PHANTOM Omni, which has a nominal max stiffness of .5:

```
// Nominal max stiffness of device used in development
// of this application.
const float originalK = .5;
float nominalK;
hdGetFloatv(HD_NOMINAL_MAX_STIFFNESS,&nominalK);
hduVector3Dd outputForce = getOutputForce();
// Original output force.
hdSetDoublev(HD_CURRENT_FORCE,
             outputForce * nominalK/originalK);
```

If the k value is below or within the nominal max stiffness recommendation but buzzing is still occurring, then the buzzing can be mitigated by further scaling down until it becomes imperceptible. However, this is generally only recommended as a last resort, because buzzing within normal stiffness may be indicative of an underlying problem in the force calculations. Scaling down forces also tends to lead to less compelling feel.

Some force effects may not use Hooke's law but instead a non-linear model. In these situations, nominal max stiffness may not be a useful guide for determining how to scale forces.

**Check the position of the device**

Devices that use armatures are typically not uniformly stable across their physical workspace.. Buzzing may occur because some characteristic force is being applied to the device when the armature is nearly extended, but when that same force is applied closer to the center of the workspace it may not cause buzzing. If possible, operate the device close to the center of its workspace. One option is to check against HD_USABLE_WORKSPACE_DIMENSIONS to determine if the device is within its usable boundaries. If not, the application could scale down forces or shut them off completely.

**Check force calculations**

Use the **hduRecord** utility or variant to check that the forces being returned to the device are reasonable. For example, initial or “light” contact with an object should produce only a small amount of force. Recording or printouts could be started upon first contact with an object to inspect if the forces generated are sensible.

**Add damping/ smoothing**

Damping is an effective tool for smoothing velocity and thus can mitigate buzzing. A typical damping formula adds a retarding force proportional to the velocity of the device:

```
// Original output force.
hduVector3Dd outputForce = getOutputForce();
const HDfloat dampingK = .001;
hduVector3Dd velocity;
hdGetDoublev(HD_LAST_VELOCITY,velocity);
hduVector3Dd retardingForce = -velocity*dampingK;
hdSetDoublev(HD_CURRENT_FORCE,
             outputForce+retardingForce);
```

An alternative is to add drag by using a low pass filter mechanism for positioning. Instead of using the actual device position for calculations, force algorithms might instead use the average of the current position and history information.

```
hduVector3Dd actualPos, lastPos;
hdGetDoublev(HD_CURRENT_POSITION,actualPos);
hdGetDoublev(HD_LAST_POSITION,lastPos);
hduVector3Dd dragPos = (actualPos+lastPos)/2.0;
hduVector3Dd outputForce = doForceCalculation(dragPos);
```

As with scaling down forces, damping or smoothing should be seen as a last resort since it mitigates the problem versus addressing its root. Few real world objects use damping, so this solution will likely feel unnatural. In some cases, damping and smoothing can even lead to increased instability.

**Detect and abort**

As a last recourse, buzzing can be detected and made to cause an abort. For example, if an application buzzes only infrequently and the cause is non-preventable, then some safety mechanism can be introduced to signal an error or shut down forces if buzzing occurs. Since buzzing is simply rapid changes in velocity, one such buzzing detection mechanism might look at the history of device velocity and determine if the velocity direction is changing beyond a certain frequency and exceeds some arbitrary amplitude.

## Force Kicking

Kicking is caused by large discontinuities in force magnitude. For example, the following will cause kicking:

```
static int timer = 0;
timer = (timer + 1 ) % 1000;
if (timer < 500)
    hdSetDoublev(HD_CURRENT_FORCE,hduVector3Dd(5,0,0));
else
    hdSetDoublev(HD_CURRENT_FORCE,hduVector3Dd(-5,0,0));
```

On the other hand, gradually ramping up the force to that magnitude will mitigate this:

```
static int timer = 0;
timer = (timer + 1 ) % 1000;
{
    hdBeginFrame(hdGetCurrentDevice());
    hdSetDoublev(HD_CURRENT_FORCE,
                 hduVector3Dd(5*timer/1000.0,0,0));
    hdEndFrame(hdGetCurrentDevice());
}
return HD_CALLBACK_CONTINUE;
```

Unintentional kicking is usually a result of the application conveying an overly large instantaneous force to the device; in other words, a force magnitude that exceeds the physical capabilities of the device. Safety mechanisms such as enabling: HD_MAX_FORCE_CLAMPING, HD_SOFTWARE_FORCE_LIMIT, or HD_SOFTWARE_VELOCITY_LIMIT, HD_SOFTWARE_FORCE_IMPULSE_LIMIT can help catch these errors and either signal an error or put a ceiling on the amount of force that can be commanded.

Kicking can also result even in the absence of bugs in force calculations if the scheduler thread is suspended, such as in debugging. For example, consider a simple plane implementation:

```
hduVector3Dd position;
hdGetDoublev(HD_CURRENT_POSITION,position);
if (position[1] < 0)
{
   const HDfloat k = .5;
   hdSetDoublev(HD_CURRENT_FORCE,
                hduVector3Dd(0,
                             -position[1]*k,
                             0));
}
```

Ordinarily this plane works in a stable manner; whenever the user attempts to penetrate the plane, the plane applies some resistance force proportional to the penetration distance. However, suppose this application was run in debug mode and a breakpoint put inside the callback. At the breakpoint, the scheduler thread would be temporarily suspended; if the user moved the device significantly in the -Y direction, then continued execution of the program, the device would suddenly be instantaneously buried to a large degree and the ensuing resistance force could kick the device upward violently.

In general, extreme care must be taken if the scheduler thread is suspended such as during debugging. Hold onto the device or put it into a safe position (such as in an inkwell) while stepping through execution.

During development, it is important that the developer take care in commanding forces and handling the device during operation. The device has both software and hardware safety mechanisms for maximum force, but this does not guarantee that it will never give a substantial kick. In particular, applications typically assume the device is being held. Small forces that are felt when the user is holding the device may generate a large velocity if the device is free. To mitigate this, either artificially scale down forces when experimenting with force effects, or use HD_CURRENT_SAFETY_SWITCH to prevent forces from being commanded inadvertently.

It is strongly advised that the developer not disable safety routines such as HD_FORCE_RAMPING and HD_SOFTWARE_VELOCITY_LIMIT, and HD_SOFTWARE_FORCE_IMPULSE_LIMIT unless there is some compelling reason to do so. Those provide a level of safeguard for protection of the device and user.

## No Forces

If a device initializes but does not generate forces:

1. Check that the hardware is working correctly using PHANToM Test.
2. Use PHANToM Test to recalibrate the device.
3. Then check that a call is being made to enable HD_FORCE_OUTPUT or verify through **hdIsEnabled()** that force output is on.
4. If the application still does not generate forces, then check that setting HD_CURRENT_FORCE is not producing an error.

Sometimes devices may generate low or no forces because of error safety conditions. Whenever an HD_EXCEEDED error occurs, the device amplifiers will toggle and then ramp to normal output. In this case, those safety mechanisms can be disabled. A word of caution, safety conditions are usually triggered as a result of some error in force calculations that may cause the device to behave uncontrollably. Also, placing the PHANTOM Omni gimbal into the inkwell will cause forces to ram, so initially forces may appear low.

## Device Stuttering

Device stuttering is a sometimes audible condition where the motors toggle off and on because the servo loop is being stopped and started. This generally happens after a force or safety error, or if the servo loop is overloaded. Use **getSchedulerTimeStamp()** at the end of a scheduler tick to check that the scheduler is running at the desired rate versus being overloaded by operations that cannot all be completed within its normal time slice.

## Error Handling

Errors should be checked for occasionally. Note that errors may not correspond to the immediately previous call or even the same thread. There is only one error stack, so errors that occur in the scheduler thread will appear when queried by the application. As a result, an error may also in the stack well after the call that lead to it.

It is also important to note, in the case of multiple devices, the device ID that the error is associated with. If all else fails, note the internal device error code and contact the vendor for information on what might have generated that. The codes available in the API represent broad categories of errors where many underlying device errors may map to the same code if they are from similar causes.

# Index