Haptic Device Abstraction Layer (HDAL)

Programmer's Guide

VERSION 3.0.27
March 30, 2010

Novint Technologies Incorporated
Albuquerque, NM USA

Copyright Notice

Trademarks

Warranties and Disclaimers

Novint Technologies does not warrant that this publication is error free. This publication could include technical or typographical errors or other inaccuracies. Novint 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@novint.com. You can also phone 1-866-298-4420.

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

Corporate Headquarters

Novint Technologies, Inc.
PO Box 66956
Albuquerque, NM 87193
Phone: 1-866-298-4420
E-mail: support@novint.com
Internet: http://www.novint.com

Preface

This manual is a guide to using the Haptic Device Abstraction Layer produced by Novint Technologies. It contains general descriptions and code samples for this SDK. This manual was current as of the release of the corresponding version of HDAL. For detailed information on all API functions, data structures, and constants, please see the HDAL API Reference.

Document Overview

Intended audience

The common attribute that should be shared by all readers of this document is a solid understanding of C++ programming. The examples are all given for the standard Microsoft IDEs Visual C++ 6.0 and Visual Studio 2005. Users of other tools will be expected to make their own conversions. Visual Studio 2003 is not included because a) Microsoft does not support it in Windows Vista, and b) it is straightforward to convert Visual C++ 6.0 projects and workspaces to Visual Studio 2003.

Beyond that, there are generally two subdivisions of the audience—those with a background in haptics and those without. Those with a solid understanding of principles of haptic programming may safely skim or skip Section 1, Introduction to Haptic Programming. It is expected that a large number of readers will have a solid background in graphical programming in the DirectX environment. These readers are cautioned to pay careful attention to all discussions of coordinate systems, as the XYZ coordinate system used by HDAL are not the same as those used by DirectX.

Typographic conventions

Format	Example	Description
Times New Roman (TNR)	This manual is a guide…	Normal text
`Courier New`	`#define VALUE 37`	Code snippets
Bold TNR	**WARNING**	High emphasis warnings, notes, etc.

Table of Contents

1 Introduction to Haptics Programming

1.1 History of haptics

The word "haptic" is rooted in Greek (*hapto*, to fasten or bind), and refers to the sensation of touch or feel using a special device that is either grasped, as a grip, or worn, as a glove. Such a device is often called a "haptic display device," in the same sense that an LCD is a visual display device. In this guide we will interchangeably use the terms haptic device, force feedback device, or touch device.

Haptic or force feedback systems originated in the 1940s for the manipulation of materials in harmful environments such as radioactive or underwater. The industry evolved from purely mechanical systems to include the use of electrical actuation and sensing, computers, and telecommunications technologies.

In the 1990s, commercial desktop haptic devices, costing thousands of dollars, have emerged, bringing high-fidelity, three-dimensional force feedback within the financial reach of many more users. Most recently, Novint Technologies has introduced the Novint Falcon. This device is priced for a consumer and enables realistic-force feedback for entertainment and training applications to users who could never afford the technology before.

1.2 Force feedback

We experience the world through our five senses: sight, hearing, touch, smell, and taste. Until now, most human-computer interaction was limited to the first two only. We could see and hear what was going on in the virtual world before us. Now we can feel it as well.

With the tip of your finger, and without any attention to what you feel with other parts of your body, reach out and touch something. Move your fingertip around the object, feeling its shape, texture, and resilience. The physical desktop holding your keyboard will probably feel flat, smooth, and hard. The cushion of the seat you are in will feel round, rough, and soft. These sensations give you a profound level of intimacy with the world around you. With good haptic programming, such a sensation can become available to you and the end users of the game or application you are developing. For the sake of simplicity, we will just use "application" from here on, recognizing that games are particular kinds of applications.

The role of the developer of haptic systems is to understand the haptic sensations that would be experienced by a real person if the virtual world of the application became real. (True, a real person would feel the real world through more than their fingertips, but an extraordinary amount of information is received even in so simple a way.) For instance, in the case of a video game, different weapons have different recoil effects when fired. On the other hand, getting shot by a .357 Magnum bullet should feel very different from getting run into by an NFL linebacker. Haptic programming is the art of representing

those sensations of forces as a three dimension function of time. (We will avoid mathematics as much as possible in this guide, but a little bit is unavoidable.)

Thus, an event in the application should initiate (or possibly terminate) a set of three functions $F_x(t)$, $F_y(t)$, and $F_z(t)$.

1.3 Continuous or discrete

In the real world, distance, force, and time are continuous. That is, given any two different times, it is possible to find a third value of time between the two. The same is true of force and position. Such is not the case in the world of computers. Given limitations on word length and sampling rates, all the elements of haptic calculations are discrete. That is, it is possible to have two values so close together that the computer cannot tell the difference between them. This is a subject that haptics theorists have spent much time and energy on. Fortunately for you, the programmer, the primary concern is the discrete nature of time. Human tactile sensors are not as fine-grained as computers can be with respect to position and force. The next section discusses the issue of time sampling in detail.

1.4 Importance of sample rate

Our three physical receptors so important to virtual reality, sight, hearing, and feel, vary widely as to their ability to perceive the effect of sample rate. Hearing is most acute in this regard. It is child's play for humans to hear the difference between two sounds with frequency 10,000 Hz and 11,000 Hz. On the other extreme, our sight is so slow that movies achieve the effect of smooth motion by presenting us little snapshots 24 times a second, knowing that we will just blur them together into smooth motion. Feel is somewhere in the middle. Research has shown that something in the range of 500 Hz to 1000 Hz is necessary to achieve the sensation of smoothness in touch.

What does this mean to the programmer? In the world of sound, you know (even if you haven't thought about it) that to produce a credible sound, it has to be packaged ahead of time (think in terms of a .wav file) and handed off to a special processor (sound card) to get the desired effect. In the world of graphics, you know (and you really should have thought about this) that you have roughly .04 seconds to do the calculations needed to hand off a hugely complex set of commands and data to a graphics processor so that the next frame can be supplied to the user soon enough for the visual effect to be achieved.

And so it is with haptics calculations: you have approximately a millisecond, with everything else going on in your virtual calculations, to calculate the forces needed to provide the user with the sensations needed for a credible haptic effect. If you have a serious graphic workload and a complex haptic calculation competing for CPU time, you have to give serious consideration to system design. Fortunately, with today's graphics processors and with some experience in haptics calculation, very complex interactions are being implemented with satisfying results.

1.5 Force models

Haptics calculation is basic applied physics in action. There are only two physics relationships needed to provide a huge percentage of haptic effects.

F = mA	basic mass times acceleration
F = kX + cV	basic damped spring.

We will consider these one at a time.

1.5.1 Mass-acceleration

If you are modeling the effect of, say, shooting a basketball, the primary force calculation is based on the mass of the basketball and the force required to produce a desired acceleration. (The purpose of this discussion is limited to how one might calculate forces; there are other simulation issues such as how to determine whether the ball is still in contact with the hand, which we will not discuss here.) Here is one approach to solving the problem:

Initial conditions:

P	Position at start of simulation
V	Velocity at start; probably zero, indicating the ball is stationary.
T	Time at start of simulation

For each sample point:

```
p =  position at this sample
t = time at this sample
dt = t – T          Delta time
v = (p – P)/dt      Current Velocity
a = (v – V)/dt      Acceleration
F = m * a           Force
// Now get ready for next sample point
T = t
V = v
P = p
```

1.5.2 Spring-mass-damper model

Another extremely useful model found throughout haptic simulations is the spring-mass-damper model. This model finds application in very easily visualized examples, such as a ball bouncing at the end of a rubber band, to not so easily visualized examples, such as a section of human flesh used in a medical training simulation. In the latter case, the flesh could well be modeled as a huge collection of very tiny spring-mass-damper elements connected to one another in a 3D mesh configuration. Between those two extremes lie many opportunities to apply this model to achieve credible haptic effects.

The basic model is shown in Figure 1. Following time-honored convention, the mass of the object is labeled m, distance x (increasing to the right), the spring constant of the spring K, and the viscosity of the damper c. The shaded rectangle on the left of the model is an immovable object.

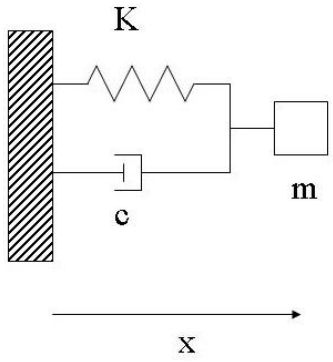


Figure 1
Basic Spring-Mass-Damper Model

In "steady state, that is, with zero velocity and zero acceleration, the force exerted on the mass by the spring is $F_s = -Kx$. The stiffer the spring (higher K), the more force is exerted for a given deflection x, and for any spring, the greater the deflection x, the greater the force. Once again, we are here ignoring boundary conditions, such as stretching the spring too far. Note the sign on the force: if we increase x by moving the mass to the right, the spring exerts a counteracting force increasing to the left.

The effect of the damper is to slow down the motion of the mass. It produces a force opposite to the velocity and proportional to it: $F_d = -cv$.

The two forces add together to produce the total force on the mass:
$F = F_s + F_d = -Kx - cv$.

If we are simulating the force on a mass that we are trying to move, we can calculate the force as follows:

Initial conditions:

P	Position at start of simulation
V	Velocity at start; probably zero, indicating the ball is stationary.
T	Time at start of simulation

For each sample point:

p = position at this sample
t = time at this sample

$dt = t - T$	Delta time
$v = (p - P)/dt$	Current Velocity
$F_s = Kp$	Spring Force
$F_d = cv$	Damper Force
$F = F_s + F_d$	Total force.

```
// Now get ready for next sample point
T = t
V = v
P = p
```

Most real simulations are somewhat more complicated than this, and for reasons beyond just the fact that all the dynamic variables are in three dimensions. For example, in both cases, we have not touched on the effects of gravity. However, this should be enough to get the newcomer to haptics simulations started. See the examples for more complete implementations.

1.6 Other reading

For more detailed information on developing programs for haptic application, the following may be helfpul:

- *Force and Touch Feedback for Virtual Reality*, Grigore C. Burdea, John Wiley & Sons, 1996
- *Stable haptic virtual reality application development platform* (Ph. D. dissertation, Eric Acosta, Texas Tech University, 2006), http://esr.lib.ttu.edu/handle/2346/1404

2 HDAL Overview

2.1 Objectives of HDAL

The primary objective of HDAL is to provide a uniform interface to all supported device types. The application programmer is relieved of the responsibility to know how each device is initialized, how its data is retrieved and delivered, and by what means the haptic force calculations are to be initiated. Novint has experience with a wide variety of modern 3D input, with and without force feedback, and has created this uniform interface based on years of experience with haptic applications programming.

The API provides means for selecting a device, initializing it, reading its state (position, buttons, etc.), issuing force commands to it, and shutting it down properly. This includes implementing a callback function that HDAL will cause to be executed at a 1 KHz rate, to achieve the required haptic fidelity.

2.2 Layered architecture

HDAL is implemented as a series of layers as indicated in Figure 2. The application consists of two primary components (for the purpose of understanding HDAL): a graphics simulation component and a haptic simulation component. The key objective of the programmer is to assure that these two components are synchronized to one another well enough that there is close correlation between what the user sees and hears (in the graphics simulation component) and what the user feels (in the haptic simulation component).

The haptic component communicates with HDAL through the HDAL API, and the primary element of that communication, after initialization, is via the callback function.

This function is called from within HDAL once every servo "tick," approximately 1,000 times per second. Within this function, the user reads the servo position, calculates the forces appropriate to the application, and sends those forces to the device.

The abstraction layer manages a family of drivers, one for each type of device known to HDAL. The primary driver is for the Novint Falcon; others will be available in the future. Each device type is supplied by its manufacturer with its own SDK; the HDAL Driver layer knows how to communicate with the associated device type SDK well enough to provide all the functionality provided by HDAL to the application. The specific device type SDK, in turn, has the responsibility to communicate with the actual device. In the case of the Novint Falcon, this communication is via USB..

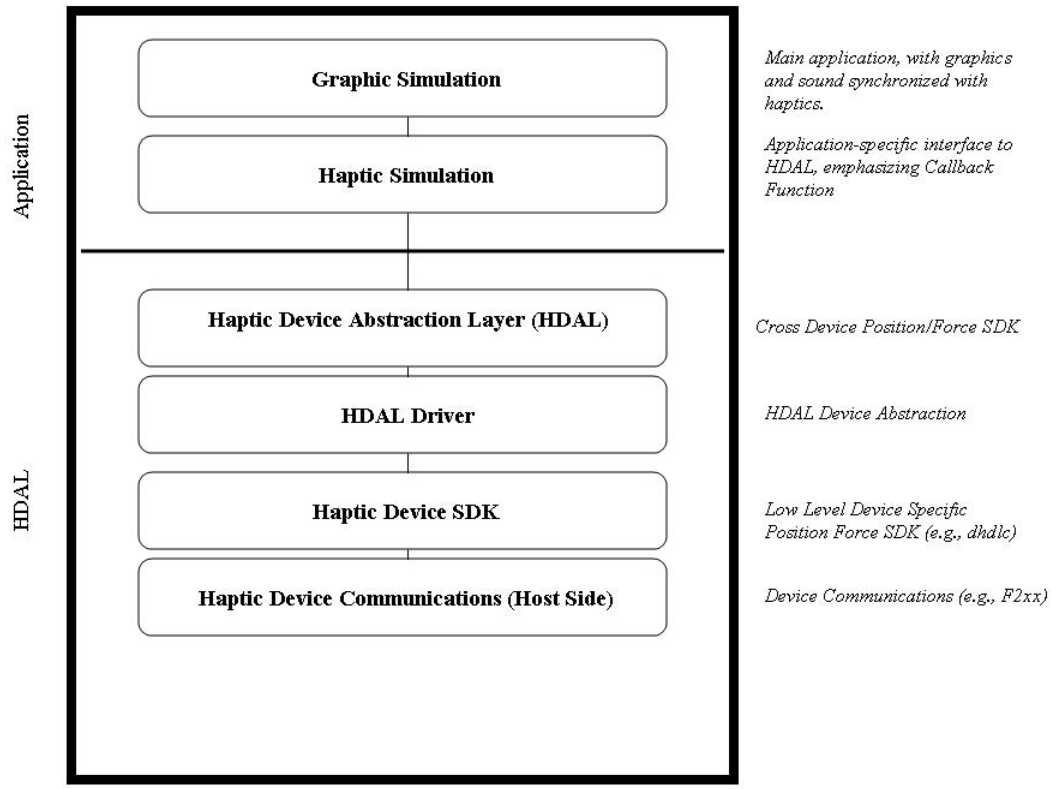


Figure 2
HDAL Layers

3 Installation

These instructions describe the installation of the HDAL SDK itself. It is presumed that you have already installed the drivers for the device you are interested in. If that device is the Novint Falcon, this was the Falcon installer CD that was shipped with your Falcon.

3.1 Installer instructions

Simply run the SDK installer. You may accept all the defaults for a normal installation, or redirect the SDK to a folder of your choice.

3.2 Environment variables

Once the installation is complete, check two environment variables

- NOVINT_DEVICE_SUPPORT
  Should point to the folder where you installed the SDK
- PATH
  Should begin with ".;XXXX\bin;", where XXXX has the same value as NOVINT_DEVICE_SUPPORT has. Note the leading "."; this allows you to put trial versions of DLLs into the application's folder for testing without disturbing the normal DLLs used for all other applications.

4 Building an application using HDAL

4.1 IDE Setup

The first step to building an HDAL-based application is to set up the development environment to find all the components. The two items of concern are the header files and the library files. This section assumes you are using one of the Microsoft Visual Studio products. These instructions are for Visual C++ 6.0, but are easily adapted to Visual Studio 2003 and Visual Studio 2005. The example projects in the Example folder are based on the assumption that they are installed in a specific relationship with respect to the previously installed drivers. These setup instructions are for a more general case, where you may have a project installed anywhere. The only assumption is that you do have the Falcon drivers installed and the NOVINT_DEVICE_SUPPORT environment variable exists and contains the path to the HDAL folder in that installation.

4.1.1 Global search setup

This procedure sets up the IDE for all development projects.

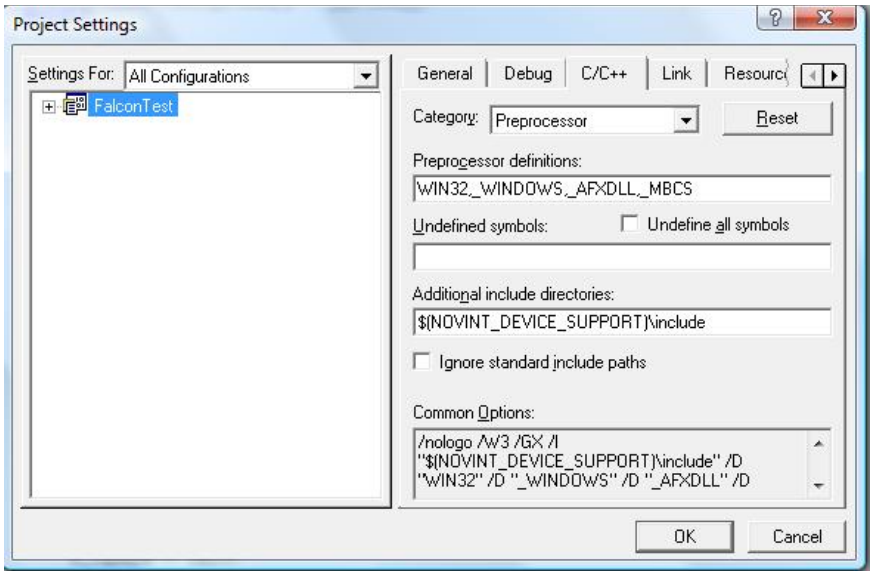
- In Tools->Options, select the Directories tab, and in the "Show directories for:" drop down box, select "Include files." Double-click on an empty slot in the directories list, click the "…" button and browse to the HDAL\include folder in the Falcon driver installation folder. Click ok.
- Still in the same dialog, select "Library files" and repeat the process, this time navigating to the HDAL\lib folder.

4.1.2 Project search setup

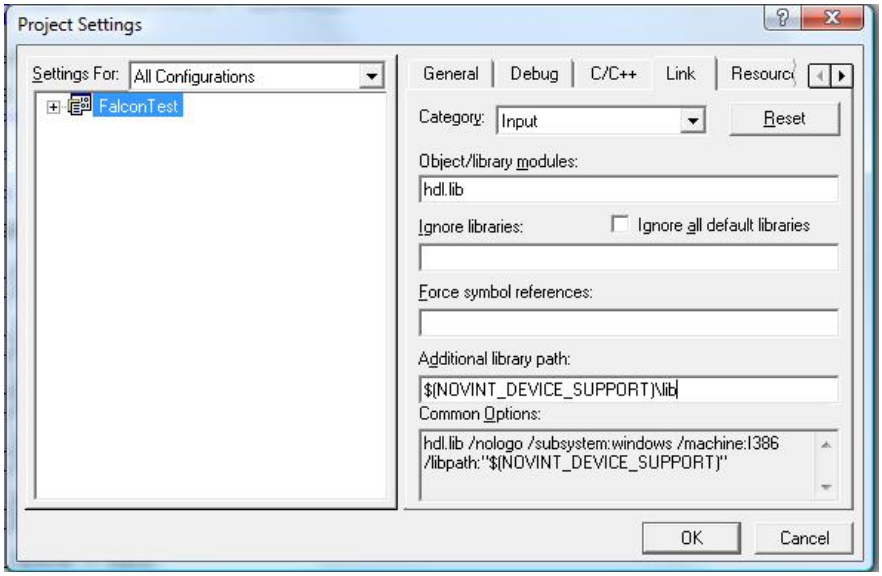
This procedure must be followed on each project that uses HDAL.

- In Project->Settings, select the C/C++ tab. In the "Settings for:" list box select "All Configurations". In the "Category:" list box, select "Preprocessor". In the "Additional include directories" add "$(NOVINT_DEVICE_SUPPORT)\include".
- Still in the same dialog, select the Link tab, Category "Input", and add "$(NOVINT_DEVICE_SUPPORT)\lib" to the "Additional library path" field.

Figure 3 shows how this would look in Visual C++ 6.0, and Figure 4 shows how it would look in Visual Studio 2005.

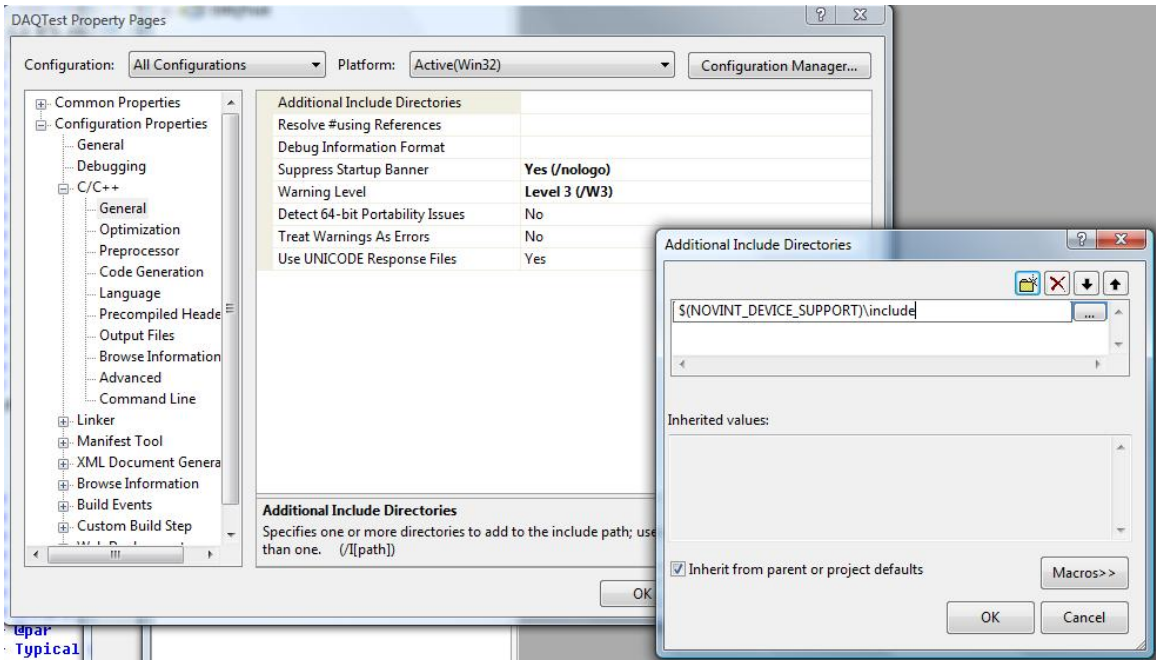


(a) Include path setup, Visual C++ 6.0

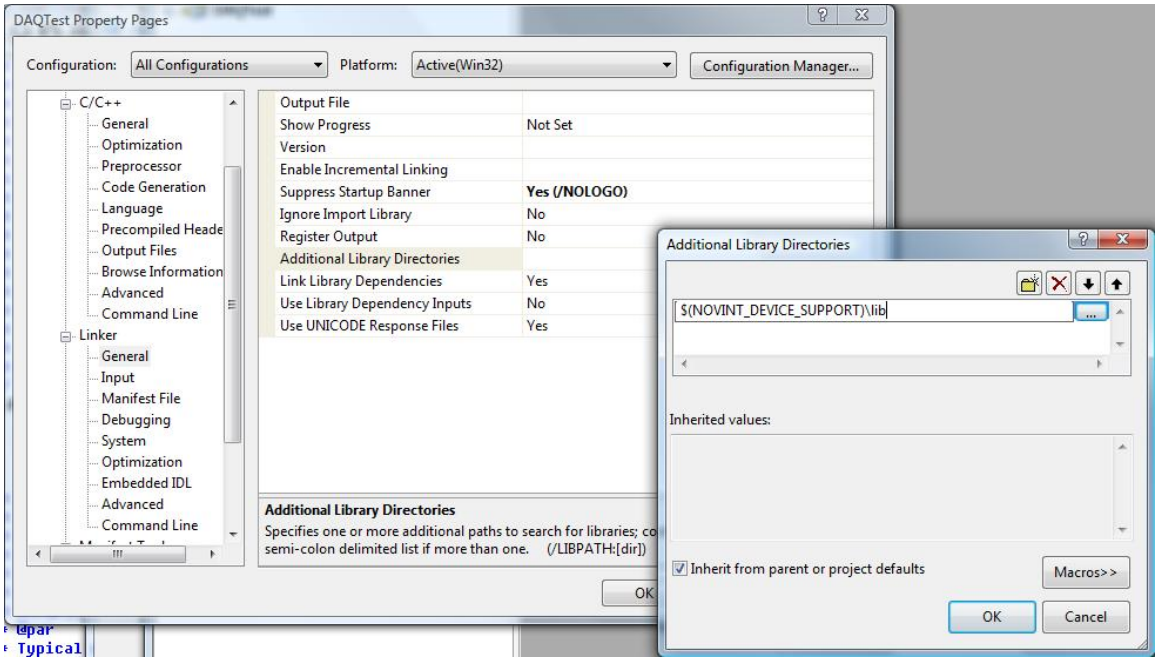


(b) Library path setup, Visual C++ 6.0

Figure 3: Search path setup, Visual C++ 6.0



(a) Include path setup, Visual Studio 2005



(b) Library path setup, Visual Studio 2005

Figure 4, Search path setup, Visual Studio 2005

4.1.3 Library file

No matter which search setup method is used, the “hdl.lib” file must be added to the “Object/library modules” list in the Link tab’s “Input” screen. Even though other DLLs

are required, only hdl.lib is required; the other DLLs are programmatically loaded by hdl.dll.

4.2 The reference point

Several HDAL API functions embed the string "Tool" in their names. This is a generic term indicating the interface between the user and the device. There are various names applied to what is here referred to as Tool: grip, end effector, device, and instrument. All have their use in various contexts; Tool is used here for brevity and general usefulness.

In HDAL, Tool simply refers to the reference point of position (hdlToolPosition), the point at which force is applied (hdlSetToolForce), or the user input scheme (hdlToolButton, hdlToolButtons).

4.3 The callback functions

4.3.1 Contact Callback

Once HDAL is properly initialized, it spawns a thread (called a "servo thread") that manages the communications to the device and calls functions defined by the application programmer. In standard parlance, these functioneare referred to as "callback" functions because the servo thread "calls back" to the application via these functiona. It is in these functions that the forces are calculated and supplied to the servo thread and data is exchanged between the application and the servo thread without the normal conflict issues that multithreaded applications usually must consider. The callback function prototype is

```
HDLServoOpExitCode HDLServoOp (void *pParam);
```

The following is, of course, a matter of system architecture and design, but it has been found to be useful to attach this function to a class, say HapticClass, as a friend function, and then pParam can be a pointer to the class. The following illustrates this.

```
// In HapticClass.h file:
class HapticClass
{ {
    friend HDLServoOpExitCode ContactCB(void *data);
    …
    HDLOpHandle m_hContactCB;
}

// In HapticClass.cpp file:
    // As part of initialization procedure…
    // See below for 'bNonBlocking' usage
    m_hContactCB = hdlCreateServoOp(ContactCB, this, bNonBlocking);
    …

HDLServoOpExitCode ContactCB(void *data)
{ {
    HapticClass *l_pHapticData = static_cast< HapticClass * >( data );
    // Now l_pHapticData points to the HapticClass object containing its data.
    …

    return HDL_SERVOOP_CONTINUE;
}
```

The HDL_SERVOOP_CONTINUE return value tells the servo thread to keep going.

4.3.2 Synchronization Callback

Frequently applications are designed with other threads that need access to haptic data. HDAL provides for this without requiring the application to deal with arcane issues of thread safety, especially since the application has no access to the HDAL servo thread itself. The best way to explain this feature is to illustrate it. We start by adding another friend function to HapticClass and a function to initiate the data transfer:

```
// In HapticClass.h file:
class HapticClass
{ {
    friend HDLServoOpExitCode ContactCB(void *data);
    friend HDLServoOpExitCode GetStateCB(void *data);
    …
public:
    void synchFromServo();     // Called from application thread

private:
    void synch();              // Called from GetStateCB (a friend function)
    double m_positionXApp;     // Application-visible data
    double m_positionXServo;   // Servo-visible data
}

// In HapticClass.cpp file:
// Callback to sync data from haptics thread to client (graphics) thread.
HDLServoOpExitCode GetStateCB(void *pUsrData)
{ {
    HapticClass *l_pHapticData = static_cast< HapticClass * >(pUsrData);
    l_pHapticData->synch();

    return HDL_SERVOOP_EXIT;  // Don't call this again until I ask for it.
}

void HapticClass::synch()
{ {
    // Here, data is copied thread-safely between data that is seen only
    // by the servo thread and data that is seen only by the application
    // thread.
    m_positionXApp = m_positionXServo;
    …
}

// synchFromServo() is called from the application thread when it wants to exchange
// data with the HapticClass object.  HDAL manages the thread synchronization
// on behalf of the application.
// See below for explanation of 'bBlocking'.
void HapticClass::synchFromServo()
{ {
    hdlCreateServoOp(GetStateCB, this, bBlocking);
}
```

Note that in the above, m_positionXApp is not typical of the complete information required for most simulations; this is just enough to illustrate how HDAL provides thread safety for the application.

The process is initiated from the application by its calling syncFromServo(). The only direct result of syncFromServo() is that a call to GetStateCB(…) is scheduled within HDAL to be run in the next servo thread cycle. As described below, the bBlocking parameter forces syncFromServo() to wait until that thread cycle is complete; if the

function were to return immediately, there would be no guarantee that the data transfer between the application and the servo thread would be complete and that the data would be valid. When the next servo thread cycle is executed, GetContactCB(…) is called, with a pointer to the HapticClass object (passed via the second parameter in the hdlCreateServoOp() call). Since GetStateCB(…) is a friend to HapticClass, it can call the synch() function as illustrated. synch() safely copies data between variables that are used only by the servo thread and variables that are used only by the application thread.

4.4 Initializing a device

The following code represents a normal initialization sequence. Sequence is important. In particular, hdlCreateServoOp must not be called before hdlInitNamedDevice, and if hdlStart is not called until after hdlCreateServoOp, the device will not operate. There is currently no error reporting to indicate a violation of these rules, but such errors will be added in the future. To avoid a future problem, follow this sequence.

```
...
#include <hdl/hdl.h>
#include <hdlu/hdlu.h>
...

// Initialize the device
// Passing 0 initializes the default device based on the
// [DEFAULT] section of HDAL.INI.   The names of other sections of HDAL.INI
// could be passed instead, allowing run-time control of different devices
// or the same device with different parameters.  See HDAL.INI for details.
HDLDeviceHandle hHandle = hdlInitNamedDevice((const char*)0);

// Now that the device is fully initialized, start the servo thread.
// Failing to do this will result in a non-funtional haptics application.
hdlStart();

// Make the device current.  All subsequent calls will
// be directed towards the current device.
hdlMakeCurrent(hHandle);

// Set up callback function
m_servoOp = hdlCreateServoOp(ContactCB, this, false);
```

4.4.1 Blocking

hdlCreateServoOp is called with two options, blocking and non-blocking. The third parameter, bBlocking (boolean), specifies which type of servo operation to schedule.

Non-blocking:
The servo operation is scheduled in the HDAL servo loop and execution is returned to the calling thread immediately.

Blocking:
The servo operation is scheduled in the HDAL servo loop and execution of the calling thread is suspended until the servo operation runs and returns. The most common use of a blocking servo operation is to synchronize device data between the application and HDAL servo threads.

Note: HDAL supports only one scheduled blocking operation at a time. This means that if your application has multiple threads, you must make sure that no two threads are trying to schedule overlapping blocking operations.

4.4.2 Initialization with Interruptibility

Normally, a successful initialization prevents another application from achieiving a successful initialization with the same device. Two functions were introduced in Version 3.0.0 that make it possible for one user program to initialize a device in such a way that if another program uses a standard initialization call, the standard call will preempt the original program, requiring the first caller to uninitialize, yielding control to the second program. This was introduced to enable the implementation of F-Gen, the Falcon general mouse controller, which could then launch games implemented before the addition of these two function calls. Only one such "interruptible" caller can be active at any time; trying to "stack" such calls will result in unpredictable and probably undesirable behavior. Since only one such interruptible caller can be active, the user must disable F-Gen before using any application that takes advantage of this feature. To disable F-Gen, simply open the Services dialog, select FalconService, and click Stop. An alternative method is to type "net stop FalconService" in a standard Command Prompt console window.

Example usage (excerpted from F-Gen code):

```
    // Open Falcon, but allow it to be hijacked
    m_hdlHandle = hdlInitNamedDeviceEx("DEFAULT", NULL, interruptedCB, this);
    if (m_bIsRunning && (HDL_INVALID_HANDLE != m_hdlHandle))
    {
        hdlStart();
        ...
```

The callback function can be very simple:

```
// This is called if a legacy Falcon app calls hdlInitxxx while this app is running
HDLServoOpExitCode interruptedCB(void* pUserData)
{ {
    FalconService* fs = static_cast< FalconService* >( pUserData );
    fs->m_interrupted = true;
    return HDL_SERVOOP_EXIT;
}
```

Then the main loop of code can monitor `m_interrupted` and take cleanup and other action when it becomes true.

4.5 Scheduling

HDAL provides a servo loop and basic scheduling of servo operations.

Servo operations can either be blocking or non-blocking (see previous section). In both cases the servo operation is scheduled and will run during the next servo tick.

The lifetime of a servo operation is determined by the code returned by the servo callback function. A servo operation can either run at each servo tick (HDL_SERVOOP_CONTINUE) or run once and quit (HDL_SERVOOP_EXIT). This allows a continuous servo operation to destroy itself by returning HDL_SERVOOP_EXIT.

Function prototypes:

```
HDLOpHandle hdlCreateServoOp(
       HDLServoOp pServoOp,  /* function pointer callback */
       void* pParam,         /* parameter to callback     */
       bool bBlocking        /* is callback blocking/non-blocking */
       );

void hdlDestroyServoOp(
       HDLOpHandle hServoOp  /* handle to servo op */
       );
```

4.6 Graphics Integration

A complete simulation consists of both haptics and graphics, and in most cases sound also. The implementer is responsible not only for credible behavior of each of these, but also for careful synchronization among them. For instance, if the user is 'holding' a hammer and swings it to strike a nail, the sound of the strike, the graphic effect of the strike, and the feel of the strike should all occur simultaneously. However, studies and experience have shown that the psychophysiology of the human sensory systems gives more weight to vision than to touch when the two are in slight disagreement. The best illustration of this is in simulating an event involving a collision between rigid bodies, such as a hardened steel sword clanging against a hardened steel shield. In such a case, the electromechanical limitations of any haptic device mean that the extreme and near-instantaneous forces generated by such a collision cannot be generated.

The normal solution in such a case is to generate as much force as the device allows for as brief a time as the device allows, and graphically (where there are no such limits) instantaneously stop the sword at the shield's surface. Even though the device continues to travel just slightly past the boundary, the user accepts the visual input that says there is no penetration.

In the Examples folder are three versions of a "Hello World" level of basic haptic programming. In each a stationary cube is presented with a small spherical object that is under the direct control of the haptic input device. When the ball touches the cube, appropriate force is generated and supplied to the device, so that the user can feel the shape, size, location, and stiffness of the cube. In all three examples, the haptic effect is the result of the same source code, a "HapticsClass" found in haptics.cpp and haptics.h. The following sections describe how three different popular graphics technologies are used to produce the same effect.

It is also possible to use HADL in programs using C instead of C++. This may simplify the use of HDAL in applications that use a high level scripting language.

4.6.1 General

The general approach to initialization is the same, no matter what graphics environment is selected:

start application
initialize sound & graphics, including loading appropriate level for game

wait to make sure these systems are up
initialize haptics
check device status
if not calibrated, notify user to do calibration (pull, push Falcon grip)
wait for calibration
continue

This will assure that not only the game but the device are properly initialized and ready for interactive use.

4.6.2 OpenGL

The basic_opengl project uses main_opengl.cpp as the driver for the program, and applies a minimal number of OpenGL functions to achieve the desired objective. In the example, GLUT is used as the window management facility, but other options are certainly available.

4.6.3 DirectX

In developing DirectX code, the programmer has a complexity choice at the outset. Microsoft's IDEs contain wizards for creating full-featured frameworks for writing DirectX applications. The wizard hints that choosing fewer options results in simpler code; choosing the minimal option set produces six .cpp files and seven .h files, even without taking the haptic code into effect. The other obvious choice to the user is to do all the coding "by hand." Following this approach, all the application level code is in one file and the haptics code in the haptics.cpp and haptics.h files. This is the approach taken by this example.

When using DirectX, be careful about coordinate systems. DirectX is left-handed and HDAL is right-handed. This difference is noted in the comments in main_dx9.cpp.

The Visual C++ 6.0 example project fails to compile in Debug mode. There is a library required that is not compatible with VC 6, and Microsoft is no longer supporting it. You will find a few lines at the top of main_dx9.cpp that test for macros _DEBUG and _VC6. These are both defined in the Debug configuration of the VC 6 project, and not both defined elsewhere. This will cause the project to fail at compile time, rather than waiting until link time to produce a rather cryptic error report.

4.6.4 3D Game Studio

Integrating HDAL into 3D Game Studio (3DGS) involves several steps. Note that the included examples have been tested on the A6 engine but not the A7.

1. Build a DLL to implement the haptic effects.
   Because there is no way that the script engine in 3DGS can meet the high sample rates required for servo operation, a separate DLL for haptic effects is required. The DLL must expose an interface defined by the macros provided with 3DGS to implement function calls and parameter passing. See GSHaptics.cpp for illustration of how to do this. In this DLL, it is good programming practice to prefix all the entry points with a prefix that is unique to the application. This is

due to the behavior of 3DGameStudio in connecting the DLLFUNCTION entry points between the game's script and the DLL. If the same entry point is found in more than one DLL in acknex's search path, there is no way to know which one will be used.

It is possible to solve the entry point problem by using dll_open and dll_close commands to provide finer control over which DLLs are loaded. Such examples are beyond the scope of this document.

2. Put this DLL in the 3DGS installation's acknex_plugins folder.
   The project's post-build step should include deleting old copies of the DLL from the 3DGS acknex_plugins directory and leaving the correct DLL in place; this is so that during game development, the user can launch the game from the 3DGS World Editor and the correct DLL will be loaded.

4.7 Multiple devices

Beginning with SDK Version 2.0, it is possible to operate up to four Falcons simultaneously. This section summarizes the issues and techniques involved in using multiple devices.

4.7.1 Pre-configured initialization

In certain applications, it is desirable to firmly associate a specific Falcon with a specific role in the application. For instance, a medical training application may have a left-hand, right-hand association. In military flight training application, one Falcon might be associated with flight control and the other for weapon control. In such fixed cases, it is straightforward to create two sections in HDAL.INI and allow the administrator of the target workstation to load Falcon serial numbers into the MANUFACTURER_NAME field of the associated section. The Multi project in the examples folder uses this concept when it is executed with no command line argument. Here is an excerpt from the associated initialization code, with error handling removed for clarity:

```
...

        // Option 1: Use device names that match your config file
        initDevice("FALCON_1", CURSOR_RED);
        initDevice("FALCON_2", CURSOR_YELLOW);
        hdlStart();
...

// Initialize device by hdal.ini section name
void initDevice(const char* deviceName, GLfloat* color = CURSOR_ORANGE)
{ {
    HapticDevice hd;
    hd.handle = hdlInitNamedDevice(deviceName);
    hd.color = color;
    for (int i = 0; i < 3; i++)
    {
        hd.force[i] = 0;
    }
    g_device.push_back(hd);
}
```

With this approach, the system is immediately ready for use with no user intervention involved in the act of associating specific devices with roles in the application. However, this is not very flexible in situations where Falcons may be connected, disconnected, reconnected and moved around. The next section describes how to initialize multiple devices in such a situation.

4.7.2 Runtime-configured initialization

There are two issues to manage in initializing Falcons with run-time selection and association.

1. The right number of devices
2. Association of device to role

HDAL provides means for handling both of these issues.

4.7.2.1 Device count

The API function hdlCountDevices() returns the number of devices attached to the PC at the moment the function is called. Note that connecting or disconnecting Falcons while an HDAL application is executing results in unpredictable behavior.

The Multi project gives an example of how this might be used. Multi looks at the command line to see how many devices to use. If there are not enough connected, it reports that to the user and continues, with only as many devices as are connected. If more than enough are connected, eventually one will be uninitialized. Here are snippets from Multi.cpp illustrating one way this can be done:

```
int main( int argc, char* argv)
{ {
    if (argc > 1)
    {
        gNumCursors = atoi(argv[1]);
    }

    // Check to see if we have enough devices connected to support the
    // user's request.

    gNumDevices = hdlCountDevices();
    if (gNumDevices < gNumCursors)
    {
        printf("You requested %d devices, but only %d are connected.\n",
                gNumCursors, gNumDevices);
        printf("Continuing with %d devices/cursors.\n", gNumDevices);
        gNumCursors = gNumDevices;
    }

    // If we don't have any devices at all, just quit.
    if (gNumCursors == 0) exit(1);
...
```

4.7.2.2 Device initialization

When the device serial numbers are unknown, a different initialization scheme is required. Instead of using hdlInitNamedDevice(), the API function hdlInitIndexedDevice() is used. The index supplied is greater than or equal to zero, and less than the value returned by hdlCountDevices(). HDAL maintains a private table of information about the devices,

and the index value is used internally to refer to an entry in that table. The return is still a handle, just as hdlInitNamedDevice(), and is used exactly the same way. The following snippet from Multi initializes all the connected Falcons, so that the buttons can be read as part of the selection and association step in the application. In this example, g_device is a vector of HapticDevice structures, each containing information needed by the application. Again, error handling is removed for simplification:

```
...
        for (int i = 0; i < gNumDevices; i++)
        {
            initIndexedDevice(i);
        }
        hdlStart();
...

// Initialize device by index into (hidden) device list.
// Note that the application does not have access to list of device
// serial numbers.  It only knows how many devices are in the list.
// Note the similarity to initDevice(…) above
void initIndexedDevice(const int ndx,  GLfloat* color = CURSOR_ORANGE)
{ {
    HapticDevice hd;
    hd.handle = hdlInitIndexedDevice(ndx);
    hd.color = color;
    for (int i = 0; i < 3; i++)
    {
        hd.force[i] = 0;
    }
    g_device.push_back(hd);
}
```

4.7.2.3 Selective uninitialization

In some circumstances, it may be the case that more devices are connected than are needed. For example, the user may be having a game night with three of his gaming buddies bringing their own Falcons. Someone leaves the room and the other three want to play on without him. The game starts by deciding which of the four Falcons are in play. The following snippet from Multi illustrates one way to do that. Basically, all Falcons are initialized, and the unused ones are uninitialized.

```
...
// It is possible to have more devices connected than needed
for (int ic = 0; ic < gNumCursors; ic++)
{ {
    printf("Press a button on the Falcon you wish to use to control”
           "the %s cursor: ", gColorName[ic]);
    int id = waitForButton();
    printf("%d\n", id);
    hdlMakeCurrent(g_device[id].handle);
    g_device[id].color = gColorChoice[ic];
}

// Now we know which devices are used, let's delete the ones not used.
// Note that one of the selected devices is current, and all selected devices
//   have colors other than orange.
for (HDContainer::iterator it = g_device.begin(); it != g_device.end(); it++)
{ {
    if ((*it).color == CURSOR_ORANGE)
```

```
    {
        hdlUninitDevice((*it).handle);
        (*it).handle = HDL_INVALID_HANDLE;
    }
}
...
```

The main restriction on hdlUninitDevice() is that it must not be called on the current device. Doing so results in unpredictable behavior. In the Multi example, note that when hdlUninitDevice is called (via uninitDevice), it is known that the current device is one that will not be uninitialized. Another restriction is that it should not be called when it might be possible for another application thread to be waiting on a blocking servo callback (see Section 4.4.1).

In the Multi example, waitForButton() returns the index of the device whose button was pressed. A different function, such as waitForMotion(), could be used, with instruction for the user to move the grip on the desired device, while all others remain motionless.

4.7.2.4 Full uninitialization

Robust applications clean up after themselves. In the case of multiple devices, since all devices are to be uninitialized, the servo must be stopped before looping through all devices to uninitialized them. Here is the relevant snippet:

```
void exitHandler()
{ {
    if (g_touchOp != HDL_INVALID_HANDLE)
    {
        hdlDestroyServoOp(g_touchOp);
        g_touchOp = HDL_INVALID_HANDLE;
    }
    hdlStop();

    for (HDContainer::iterator it = g_device.begin();
         it != g_device.end(); ++it)
    {
        if ((*it).handle != HDL_INVALID_HANDLE)
            hdlUninitDevice((*it).handle);
    }

}
```

It is good programming practice to tell Windows about this function, so it can be called in the event that the user does something rude like clicking on the window's close button or otherwise terminating the application in an unexpected way:

```
atexit(exitHandler);
```

4.7.3 Device enumeration

Version 3.0.0 of the SDK introduced a significant advance in the options available to the programmer to know what devices are connected at any moment. Before this, the only option was to call hdlCountDevices(), which would report the number of unopened Falcon devices connected at the start of the application. This section describes how to use a new function, hdlCatalogDevices(...), to get a customized list of connected devices.

4.7.3.1 Serial Number classification

All Novint USB-based devices come from the factory with an 8-character serial number. The first character of the serial number classifies the device, and the second is used as a "version" number. The classification scheme is:

- Leading digit indicates a normal base unit. Most device serial numbers will have a digit as the first character.
- Leading upper-case character indicates a "special" base unit. These are customized in some way for individual customers.
- Leading lower-case character indicates a "smart grip." Some custom-made grips use electronics similar to base units, and are initialized in a way very similar to base units.

4.7.3.2 Masking

hdlCatalogDevices(...) includes a pointer to a list of serial number masks. The purpose of the masks is to hide devices from the device catalog. Thus, if a device carries a serial number that "matches" a mask, it is excluded from the list of serial numbers returned by the function. The list is represented as an array of character pointers, and each character pointer references a string of eight characters. A string of other than eight characters can never match, but no error is returned, so be sure to provide eight-character strings. For a serial number to match the mask, all eight characters in the serial number must match the character in the mask. If the serial number matches any of the supplied masks, the matching process terminates with success, and the serial number is excluded. A character in the serial number matches a character in the mask if:

- The two characters are identical
- The mask character is a dot ('.').
- The mask character is a dollar ('$') and the serial number character is a lower-case letter.
- The mask character is a commercial at ('@') and the serial number character is an upper-case character.
- The mask character is an asterisk ('*') and the serial number character is a digit.

There is a default mask list to be used in case hdlCatalogDevices(...) is never called. The list is represented in C as:

```
char* DEFAULT_MASKS[] = {"$.......", NULL};
```

This produces the pre-3.0.0 behavior of hdlCountDevices().

4.7.3.3 Device availability

The availability parameter to hdlCatalogDevices(...) indicates how the device should be treated with respect to its being opened or not by the current or some other HDAL-based application. If a value other than HDL_DEFAULT_AVAILABILITY is passed, it is stored for future use by this function and by hdlCountDevices().

- HDL_DEFAULT_AVAILABILITY: Use the currently stored availability option.
- HDL_ALL_DEVICES: Include all devices, whether or not they are currently opened by any application.

- HDL_NOT_OPEN_BY_ANY_APP: Include only those devices that are not currently open by any application. This is the default behavior of hdlCountDevices().
- HDL_NOT_OPEN_BY_OTHER_APP: Include those devices that are either not opened, or are opened by the current application. In such a case, the unopened devices are at the beginning of the returned list, and opened devices at the end.

4.7.3.4 Typical usage

Legacy applications, those written prior to SDK 3.0.0, do not know about hdlCatalogDevices(...), and use only hdlCountDevices(). The defaults for hdlCatalogDevices(...) have been chosen specifically to reproduce the behavior of hdlCountDevices(). That is, availability is HDL_NOT_OPEN_BY_ANY_APP and the mask list is DEFAULT_MASKS as defined above, which filters out "smart grips". New apps that do not call hdlCatalogDevices(...) at all will receive this behavior, which may well be adequate for most applications.

To get finer control, call hdlCatalogDevices(...) once with the desired availability and/or mask list. Typically one call during application setup time will be adequate, as the application's selection requirements are usually unlikely to change. Thereafter, just call hdlCountDevices() to see how many devices are connected that match the desired criteria.

To get maximum control, include a pointer to a buffer to receive a copy of the serial numbers for the application's use. The buffer must be constructed as

```
char devSerNums[HDL_MAX_DEVICES*HDL_SERNUM_BUFFSIZE];
```

The constants are defined in hdlConstants.h, and HDL_SERNUM_BUFFSIZE includes a null terminator for the serial number string. Thus, serial number N is a null-terminated C string starting at `&devSerNums[N*HDL_SERNUM_BUFFSIZE]`.

Under no circumstances should either hdlCatalogDevices(...) or hdlCountDevices() be called inside either a synch callback function or a normal servo callback function. The enumeration of devices is fairly time-consuming (80 msec per connected device is a useful approximation) and performing it from the servo thread would wreck the timing. Since connectivity changes are fairly slow events anyway, there is nothing to be gained from calling them from inside the servo thread. Call them when the application is looking for state changes.

5 Configuration Files

5.1 HDAL.INI

HDAL.INI is the master configuration file for HDAL. HDAL.INI is normally found in the config folder of the the HDAL folder pointed to by the NOVINT_DEVICE_SUPPORT environment variable. However, if the user has not modified the PATH from the normal Novint installation, a "local" copy of HDAL.INI can be placed in the application executable's folder, and it will supersede the normal

copy. This way, applications can be easily customized at the HDAL level without interfering with other HDAL-based applications.

NOTE: To accommodate development from within an IDE, such as Visual Studio, if the executable is in a Release or Debug folder, the search actually begins one level up, where the project files are stored.

HDAL.INI is read during execution of hdlInitNamedDevice() or hdlInitIndexedDevice(), and sets up a variety of parameters controlling the behavior of the selected device. In the case of calling hdlInitNamedDevice() without naming a device, or calling hdlInitIndexedDevice(), certain parameters are read from the [DEFAULT] section of HDAL.INI. This section of the Programmer's Guide explains the control options available. The [DEFAULT] section of the HDAL.INI shipped with the SDK will be used as illustration. It is given here for reference:

```
[DEFAULT]
MANUFACTURER="Novint"
MANUFACTURER_NAME="Default Falcon"
DLL=dhdlcDriver.dll
POSX_SCALE=1.0;
POSY_SCALE=1.0;
POSZ_SCALE=1.0;
POSX_OFFSET=0.0;
POSY_OFFSET=0.0;
POSZ_OFFSET=0.0;
; Set Logging level for xxxDriver.dll
; Logging Level sets level for diagnostic logging of driver events
; Values to use for levels:
; 0        Panic, system is about to stop
; 1        Severe, serious problem; stopping is likely
; 2        Warning, programmer needs to fix this problem
; 3        Info, generally useful informaton
; 4        Debug, only use this level at Novint request; very verbose
LOG_LEVEL=0
LOG_SERVO=0;
LOG_SERVO_THRESHOLD=0.001;
LOG_SERVO_FILENAME=logServo
```

5.1.1 Section Name

Each section has a name, and it is the value of this name, not including the bracket delimiters, that is supplied to hdlInitNamedDevice(). "DEFAULT" is assumed if no name is supplied.

5.1.2 MANUFACTURER

Currently not used.

5.1.3 MANUFACTURER NAME

Each manufacturer has a means of identifying that manufacturer's devices. This field contains the manufacturer's specified identification string. In the case of the Falcon, this may be used to contain the device serial number, which can be obtained from FalconTest.

5.1.4 DLL

This field names the top level device driver to be used with the device. Refer to Figure 2, HDAL Layers. This driver fits in the HDAL Driver slot. Its role is to translate from the HDAL API functions to the manufacturer's API supplied by the manufacturer's driver API in the Haptic Device SDK layer.

A useful driver to use for troubleshooting your program is nullDriver.dll, supplied in the SDK's HDAL/bin folder. This driver does not require a Falcon to be connected, produces no forces, always supplies zero as the position, and always returns normal values to its function calls. If you've reached the point in your code where you're not sure whether it's your code or the Falcon, sometimes replacing dhdlcDriver.dll with nullDriver.dll will help you sort it all out. Just remember to return to dhdlcDriver.dll when you're ready to use the Falcon again.

5.1.5 POS*_SCALE and POS*_OFFSET

The position values normally supplied by hdlToolPosition() are measured in meters, with the origin of the coordinate system approximately in the center of the physical workspace. Thus, since the physical workspace is about 4 inches in each dimension, each dimension is normally in the range +/- 0.05. This can be changed to any reasonable workspace by adjusting these six parameters. The normal development process involves some trial and error to get them adjusted correctly. Many developers never touch these values, preferring instead to make any transformations inside the application itself.

5.1.6 LOG_LEVEL

HDAL has a great deal of diagnostic information available. Normally this information is not output because the storage requirements may be excessive for heavy usage, such as might be found with very active game players or with an application that runs for many hours a day. However, for diagnostic purposes, and usually under the direction of Novint's Technical Support staff, logging can be enabled by setting this value. To repeat: in order not to consume too much file space, it should be left at 0, to minimize storage requirements.

LOG_LEVEL is not included in the end-user version of HDAL.INI, to reduce the likelihood of customer support calls due to users experimenting with it without a solid grasp of what to expect.

5.1.7 LOG_SERVO*

As emphasized in Section 1.4, The Importance of Servo Rate, it is important to maintain a servo tick rate of 1 KHz, corresponding to a 1 millisecond interval, or as close to that as possible. The force calculation function (ContactCB in the examples in Section 4) must be executed within that millisecond. Also, it is possible for other processes to interfere

with the servo tick rate. The three LOG_SERVO* parameters are provided to allow the developer to collect statistics on servo intervals for analysis to assure that the design criteria are being met.

LOG_SERVO specifies the duration of the sample interval, given as an integer number of seconds. Positive numbers specify the number of seconds from the beginning of execution; negative numbers specify the number of seconds just before the application is terminated. This allows the user to capture data beginning at startup time or to capture data pertinent to events that may occur at random times during execution. For instance, by setting the LOG_SERVO time to -30, the developer can run the application until some (un)desired event occurs, and terminate the application at that point. The log file will then contain servo tick rate data for the 30 seconds just before termination.

LOG_SERVO_THRESHOLD defines a threshold value of servo interval to record. Intervals briefer than this are not logged. To capture all intervals, simply set this value to 0.0.

LOG_SERVO_FILENAME defines the root name of the file to use for logging data. HDAL appends a date and timestamp to the root name to make it easy to differentiate among several runs of sampling. The default name is "servoLog". If there is no LOG_SERVO_FILENAME specified, a typical file name might be "servoLog_2007.09.12_09h.27m.35s.txt. Also note that the filename must be "bare"—no quote marks, no trailing semicolon. A valid entry could be

```
LOG_SERVO_FILENAME=myFileName
```

But the following would be invalid

```
LOG_SERVO_FILENAME="myFileName"
LOG_SERVO_FILENAME=myFileName;
LOG_SERVO_FILENAME="myFileName";
```

The format of the file content is suitable for processing by data analysis tools such as Microsoft Excel. The fields are fixed width, making it suitable for "eyeballing" and for importing using fixed-width import processing.

LOG_SERVO* are not included in the end-user version of HDAL.INI, to reduce the likelihood of customer support calls due to users experimenting with it without a solid grasp of what to expect.

5.1.8 GRIP

In Version 3.0.0 of the SDK, Novint introduced the concept of the "smart grip." All Falcon grips have a microprocessor on board. The original grips' microprocessor firmware was pre-programmed at the factory. Novint now has the ability to produce grips that are based on technologies that allow the firmware to be downloaded at application execution time, similar to the procedure used to download the firmware to the

base unit. If such a grip is to be used, the grip is associated with the base unit by adding a "GRIP" item to the descriptor in HDAL.INI. The value assigned to GRIP is the name of another section in the INI file. In the HDAL.INI file that ships with this SDK, see the example where the [FALCON_1] section includes a "GRIP=FALCON_2" entry. This means that the device specified in the [FALCON_2] section is the smart grip associated with the FALCON_1 device.

No special handling is required by the application. In the FALCON_1/FALCON_2 example, when FALCON_1 is initialized, HDAL detects that there is an associated grip and initializes it as part of the initialization of FALCON_1. The application has no direct communication with FALCON_2 except through the hdlGrip* function calls.

5.2 FalconCommon.txt

Control of gravity compensation is in a separate file, FalconCommon.txt. This file is searched for according to the same rules as those used to find HDAL.INI. Thus, if it exists in the NOVINT_DEVICE_SUPPORT\config folder, it applies to all applications. This "master" version is overridden if there is a FalconCommon.txt file in the application's launch folder. If neither exists, the defaults are used.

The content of this file is a single line: "0 gravity" to turn gravity compensation off, or "1 gravity" to turn it on (the default). Note that spelling and capitalization matters, and only one space is allowed between the digit and "gravity".

This file is read on two types of occasions: when a Falcon is initialized and when a grip is changed. To experiment with gravity compensation, just create a FalconCommon.txt file in the application's launch folder, and with the application running, edit the file to change the "1 gravity" line to "0 gravity" and back, removing and reattaching the grip after each change to make it take effect.

5.3 binaries_index.ini

The addition of the Grip API and the smart grip capability resulted in the addition of a new index file. This file is used from within HDAL to look up file names for binary files to be loaded as needed, depending on the actual devices in operation. The two sections of the file are [GRIPS] and [BINARIES]. This file is not to be edited by the programmer or end user.

5.3.1 [GRIPS]

The Grip API architecture has been designed to minimize the need to produce a new release of HDAL every time a new grip is released. Each new grip will be delivered with a DLL and an updated copy of binaries_index.ini. This section is used by HDAL to determine which Grip DLL to load when a new grip is attached to the base unit. The programmer may read it to see what Grip API functions have meaningful results for each grip.

5.3.2 [BINARIES]

The files downloaded to base units and smart grips are called “binaries”. The two leading characters of the device serial number are used in a lookup table to learn the name of the file to download to the device.

5.4 Utilities

5.4.1 HDLU

HDLU defines several useful entry points for assisting programmers. See the API Reference manual for their descriptions.

5.4.2 NDSSetter

There are two environment variables used by HDAL to locate the various DLLs, BINs, and configuration files needed at run time. Developers frequently like to keep different versions and configurations around, and these two variables make it easy to use different configurations. All that is needed is to edit NOVINT_DEVICE_SUPPORT to point to the HDAL folder than contains the bin and config folders to be used, and to add that folder’s bin to the PATH environment variable. The traditional way to do this is to open Computer Properties, navigate to Environment Variables, select the two variables and type in the paths you want. And when you’re done, you need to remember to undo what you just did. This is manually intensive and error prone. Enter NDSSetter to relieve you of that effort.

Just launch NDSSetter from the Utilities folder, click the “…” button and browse to the HDAL folder you want, and click either Apply or OK. If you prefer, you can type in the name of the desired folder. On every character change, NDSSetter checks to see if the current value represents a proper HDAL folder, and enables or disables the OK and Apply buttons accordingly.

If you click Apply and leave NDSSetter running, you can revert to the values active when it was launched by clicking Restore. If you realize you’ve navigated to the wrong folder, you can click Cancel and start over again.

Note that in Windows Vista, you must launch NDSSetter with Administrator privileges (“Run as Administrator”).

6 Troubleshooting

This section is specific to the Novint Falcon. For troubleshooting other devices, see the appropriate manufacturer’s documentation.

6.1 Badge

The “badge” is the front panel of the Falcon. It features three hidden LEDs which light up under specific circumstances mentioned in the following sections.

6.1.1 Homed status

If any one of the Falcon's motors are not 'homed' the badge will show red. In this condition, forces are disabled, and the user should home the motors by pulling the grip all the way away from the base unit, and then in one continuous motion push the grip toward the base unit. As the grip passes through the approximate center of the physical workspace, the badge will switch to blue, indicating successful homing.

6.1.2 Power

When the Falcon is plugged into a live USB connector, all three LEDs (green, red, blue) will light, producing a teal-like color. They will remain in this condition until homed, at which time they will turn blue until the controlling application terminates, at which time they will turn off, leaving the badge unilluminated. If the Falcon remains connected to a live USB port, the homed status is remembered and the next time an application starts, the badge should immediately show blue.

6.2 FalconCheck

FalconCheck is a simple connectivity and basic operation check. When FalconCheck is launched, the LEDs will begin cycling green-blue-red. This will continue until you press a button, at which time the cycle rate will speed up and the Falcon will vibrate the grip at a low "rumble" frequency. When you release the button FalconCheck will terminate. This shows quite a lot for such a simple test: the Falcon is connected, the drivers are properly installed, the LEDs are functioning, the grip is communicating with the base unit, and forces are operational.

FalconCheck (and FalconTest, described in the next section) both produce a log file, FALCONTH_LOG.txt, in their respective executable folders. This file contains diagnostic information that may be required by Novint Technical Support in the event of a support call.

6.3 FalconTest

FalconTest is a factory test tool that end users may be called upon to use if needed by Novint Customer Support. It gives the user full control over all the inputs and outputs of the Falcon. Homed status for each motor is displayed. Raw encoder values are displayed (not translated to meters, as HDAL presents in the API). Raw torques can be commanded to the motors, with a safety control, defaulted to not allow negative (pull away from the body) forces. Error rates on the USB data transfer are shown (about 10% is normal). The LEDs can be commanded. Full details on usage are in the Falcon Test Procedure document included.

6.4 NtInGUI

NtInGUI is FalconTest's precursor, and is included for legacy users who may prefer it to FalconTest.

7 References

Shreiner, Dave, Mason Woo, Jackie Neider, and Tom Davis, *OpenGL Programming Guide*, Boston, Addison-Wesley, 2004 “The Red Book”

Watt, Alan, *3D Computer Graphics, Third Edition,* Harlow, England, Addison-Wesley, 2000.

McLaughlin, Margaret L., João P. Hespanha, and Gaurav S. Sukhatme, *Touch in Virtual Environments*, Upper Saddle River, New Jersey, Prentice Hall PTR, 2002.

Burdea, Grigore C., *Force and Touch Feedback for Virtual Reality*, New York, John Wiley & Sons, 1996.

Shneider, Philip J. and David H. Eberly, *Geometric Tools for Computer Graphics*, San Francisco, Marvin Kaufman Publishers, 2003.

Appendix A: Example Project

This appendix contains a complete example of an HDAL application. It is the OpenGL version of the BasicTest example in the SDK's Examples folder. The full Visual Studio project is not listed here (see the Examples folder), but the first section below describes the key setup steps.

7.1 Project Structure

The project files include:

main_openGL.cpp	The entry point for the application
haptics.h	The HapticsClass declaration file
haptics.cpp	The HapticsClass definition file

Prerequisites:
OpenGL and GLUT

The specific development environment setup is not described here, as there are so many to choose from. The setups that are required are:

- Include path = ..\..\..\..\include
  This is the setup specific to the Examples folder in the SDK. The directory traversal may vary depending on where and how you set up your project. The key is that the relative path must point to the HDAL\include folder in the SDK. An option is to define an environment variable, say, HDAL_SDK to point to the HDAL folder of the SDK, then make the include path = $(HDAL_SDK)\include, or whatever syntax is appropriate to your compiler.
- Library path = ..\..\..\..\lib.
  The same discussion applies here.
- Libraries = hdl.lib

7.2 Haptics.h

```
// Copyright 2007 Novint Technologies, Inc. All rights reserved.
// Available only under license from Novint Technologies, Inc.

// Make sure this header is included only once
#ifndef HAPTICS_H
#define HAPTICS_H

#include <hdl/hdl.h>
#include <hdlu/hdlu.h>

// Know which face is in contact
enum RS_Face {
    FACE_NONE = -1,
    FACE_NEAR, FACE_RIGHT,
    FACE_FAR, FACE_LEFT,
    FACE_TOP, FACE_BOTTOM,
    FACE_DEFAULT,
    FACE_LAST               // reserved to allow iteration over faces
    };

// Blocking values
const bool bNonBlocking = false;
```

```
const bool bBlocking = true;

class HapticsClass
{ {

// Define callback functions as friends
friend HDLServoOpExitCode ContactCB(void *data);
friend HDLServoOpExitCode GetStateCB(void *data);

public:
    // Constructor
    HapticsClass();

    // Destructor
    ~HapticsClass();

    // Initialize
    void init(double a_cubeSize, double a_stiffness);

    // Clean up
    void uninit();

    // Get position
    void getPosition(double pos[3]);

    // Get state of device button
    bool isButtonDown();

    // synchFromServo() is called from the application thread when it wants to exchange
    // data with the HapticClass object.  HDAL manages the thread synchronization
    // on behalf of the application.
    void synchFromServo();

    // Get ready state of device.
    bool isDeviceCalibrated();

private:
    // Move data between servo and app variables
    void synch();

    // Calculate contact force with cube
    void cubeContact();

    // Check error result; display message and abort, if any
    void testHDLError(const char* str);

    // Nothing happens until initialization is done
    bool m_inited;

    // Variables used only by servo thread
    double m_positionServo[3];
    bool   m_buttonServo;
    double m_forceServo[3];

    // Variables used only by application thread
    double m_positionApp[3];
    bool   m_buttonApp;

    // Keep track of last face to have contact
    int    m_lastFace;

    // Handle to device
    HDLDeviceHandle m_deviceHandle;

    // Handle to Contact Callback
    HDLServoOpExitCode m_servoOp;

    // Size of cube
    double m_cubeEdgeLength;

    // Stiffness of cube
```

```
    double m_cubeStiffness;
};

#endif // HAPTICS_H
```

7.3 Haptics.cpp

```
#include "haptics.h"
#include <windows.h>
#include <math.h>

// Continuous servo callback function
HDLServoOpExitCode ContactCB(void* pUserData)
{ {
    // Get pointer to haptics object
    HapticsClass* haptics = static_cast< HapticsClass* >( pUserData );

    // Get current state of haptic device
    hdlToolPosition(haptics->m_positionServo);
    hdlToolButton(&(haptics->m_buttonServo));

    // Call the function that does the heavy duty calculations.
    haptics->cubeContact();

    // Send forces to device
    hdlSetToolForce(haptics->m_forceServo);

    // Make sure to continue processing
    return HDL_SERVOOP_CONTINUE;
}

// On-demand synchronization callback function
HDLServoOpExitCode GetStateCB(void* pUserData)
{ {
    // Get pointer to haptics object
    HapticsClass* haptics = static_cast< HapticsClass* >( pUserData );

    // Call the function that copies data between servo side
    // and client side
    haptics->synch();

    // Only do this once.  The application will decide when it
    // wants to do it again, and call CreateServoOp with
    // bBlocking = true
    return HDL_SERVOOP_EXIT;
}

// Constructor--just make sure needed variables are initialized.
HapticsClass::HapticsClass()
    : m_lastFace(FACE_NONE),
      m_deviceHandle(HDL_INVALID_HANDLE),
      m_servoOp(HDL_INVALID_HANDLE),
      m_cubeEdgeLength(1),
      m_cubeStiffness(1),
      m_inited(false)
{ {
    for (int i = 0; i < 3; i++)
        m_positionServo[i] = 0;
}

// Destructor--make sure devices are uninited.
HapticsClass::~HapticsClass()
{ {
    uninit();
}



void HapticsClass::init(double a_cubeSize, double a_stiffness)
{ {
    m_cubeEdgeLength = a_cubeSize;
```

```
    m_cubeStiffness = a_stiffness;


    HDLError err = HDL_NO_ERROR;

    // Passing "DEFAULT" or 0 initializes the default device based on the
    // [DEFAULT] section of HDAL.INI.   The names of other sections of HDAL.INI
    // could be passed instead, allowing run-time control of different devices
    // or the same device with different parameters.  See HDAL.INI for details.
    m_deviceHandle = hdlInitNamedDevice("DEFAULT");
    testHDLError("hdlInitDevice");

    if (m_deviceHandle == HDL_INVALID_HANDLE)
    {
        MessageBox(NULL, "Could not open device", "Device Failure", MB_OK);
        exit(0);
    }

    // Now that the device is fully initialized, start the servo thread.
    // Failing to do this will result in a non-funtional haptics application.
    hdlStart();
    testHDLError("hdlStart");

    // Set up callback function
    m_servoOp = hdlCreateServoOp(ContactCB, this, bNonBlocking);
    if (m_servoOp == HDL_INVALID_HANDLE)
    {
        MessageBox(NULL, "Invalid servo op handle", "Device Failure", MB_OK);
    }
    testHDLError("hdlCreateServoOp");

    // Make the device current.  All subsequent calls will
    // be directed towards the current device.
    hdlMakeCurrent(m_deviceHandle);
    testHDLError("hdlMakeCurrent");

    m_inited = true;
}

// uninit() undoes the setup in reverse order.  Note the setting of
// handles.  This prevents a problem if uninit() is called
// more than once.
void HapticsClass::uninit()
{ {
    if (m_servoOp != HDL_INVALID_HANDLE)
    {
        hdlDestroyServoOp(m_servoOp);
        m_servoOp = HDL_INVALID_HANDLE;
    }
    hdlStop();
    if (m_deviceHandle != HDL_INVALID_HANDLE)
    {
        hdlUninitDevice(m_deviceHandle);
        m_deviceHandle = HDL_INVALID_HANDLE;
    }
    m_inited = false;
}

// This is a simple function for testing error returns.  A production
// application would need to be more sophisticated than this.
void HapticsClass::testHDLError(const char* str)
{ {
    HDLError err = hdlGetError();
    if (err != HDL_NO_ERROR)
    {
        MessageBox(NULL, str, "HDAL ERROR", MB_OK);
        abort();
    }
}

// This is the entry point used by the application to synchronize
```

```
// data access to the device.  Using this function eliminates the
// need for the application to worry about threads.
void HapticsClass::synchFromServo()
{ {
    hdlCreateServoOp(GetStateCB, this, bBlocking);
}

// GetStateCB calls this function to do the actual data movement.
void HapticsClass::synch()
{ {
    // m_positionApp is set in cubeContact().
    m_buttonApp = m_buttonServo;
}


// Here is where the heavy calculations are done.  This function is
// called from ContactCB to calculate the forces based on current
// cursor position and cube dimensions.  A simple spring model is
// used.
void HapticsClass::cubeContact()
{ {
    m_positionApp[0] = m_positionServo[0];
    m_positionApp[1] = m_positionServo[1];
    m_positionApp[2] = m_positionServo[2];

    m_forceServo[0] = 0;
    m_forceServo[1] = 0;
    m_forceServo[2] = 0;

    // Skip the whole thing if not initialized
    if (!m_inited) return;

    const double halfCube = m_cubeEdgeLength/2.0;

    double radiusWC = 0.0;
    double pointLC[3];

    // Get the cursor position in Local Coordinates.
    pointLC[0] = m_positionApp[0];
    pointLC[1] = m_positionApp[1];
    pointLC[2] = m_positionApp[2];

    // expand the cube faces by the radius of the tool
    // gives sphere-cube force model
    double l_size = halfCube + radiusWC;

    // compute signed distance to each face
    // distance > 0 implies tool position penetrates cube
    // according the specific face
    const int faces = 6;
    double l_distance[faces];
    l_distance[FACE_NEAR]   = l_size - pointLC[2];
    l_distance[FACE_RIGHT]  = l_size - pointLC[0];
    l_distance[FACE_FAR]    = pointLC[2] + l_size;
    l_distance[FACE_LEFT]   = pointLC[0] + l_size;
    l_distance[FACE_TOP]    = l_size - pointLC[1];
    l_distance[FACE_BOTTOM] = pointLC[1] + l_size;

    // Keep track of which cube face the cursor is nearest to.
    int l_nearestFace = FACE_NONE;
    if (l_nearestFace == FACE_NONE)
    {
        l_nearestFace = FACE_NEAR;
        double l_minDistance = fabs(l_distance[FACE_NEAR]);
        for (int index = FACE_RIGHT; index <= FACE_BOTTOM; ++index)
        {
            if (fabs(l_distance[index]) < l_minDistance)
            {
                l_nearestFace = index;
                l_minDistance = fabs(l_distance[index]);
            }
```

```
        }
    }

    // have the nearest face
    // tool must be interior to the face for a collision
    if (l_distance[l_nearestFace] < 0)
    {
        m_lastFace = FACE_NONE;
        m_forceServo[0] = 0;
        m_forceServo[1] = 0;
        m_forceServo[2] = 0;
        return;
    }

    // We know what we need to know.  Handle one of six possibilities:
    switch (l_nearestFace)
    {
        case FACE_NEAR:
        if (fabs(pointLC[0]) < l_size && fabs(pointLC[1]) < l_size)
        {
            m_forceServo[2] = l_size - pointLC[2];
            m_lastFace = l_nearestFace;
        }
        break;

        case FACE_FAR:
        if (fabs(pointLC[0]) < l_size && fabs(pointLC[1]) < l_size)
        {
            m_forceServo[2] = -l_size - pointLC[2];
            m_lastFace = l_nearestFace;
        }
        break;

        case FACE_RIGHT:
        if (fabs(pointLC[1]) < l_size && fabs(pointLC[2]) < l_size)
        {
            m_forceServo[0] = l_size - pointLC[0];
            m_lastFace = l_nearestFace;
            break;
        }
        break;

        case FACE_LEFT:
        if (fabs(pointLC[1]) < l_size && fabs(pointLC[2]) < l_size)
        {
            m_forceServo[0] = -l_size - pointLC[0];
            m_lastFace = l_nearestFace;
        }
        break;

        case FACE_TOP:
        if (fabs(pointLC[0]) < l_size && fabs(pointLC[2]) < l_size)
        {
            m_forceServo[1] = l_size - pointLC[1];
            m_lastFace = l_nearestFace;
        }
        break;

        case FACE_BOTTOM:
        if (fabs(pointLC[0]) < l_size && fabs(pointLC[2]) < l_size)
        {
            m_forceServo[1] = -l_size - pointLC[1];
            m_lastFace = l_nearestFace;
        }
    }

    // add spring stiffness to force effect
    m_forceServo[0] *= m_cubeStiffness;
    m_forceServo[1] *= m_cubeStiffness;
    m_forceServo[2] *= m_cubeStiffness;
}
```

```
// Interface function to get current position
void HapticsClass::getPosition(double pos[3])
{ {
    pos[0] = m_positionApp[0];
    pos[1] = m_positionApp[1];
    pos[2] = m_positionApp[2];

}

// Interface function to get button state.  Only one button is used
// in this application.
bool HapticsClass::isButtonDown()
{ {
    return m_buttonApp;
}

// For this application, the only device status of interest is the
// calibration status.  A different application may want to test for
// HDAL_UNINITIALIZED and/or HDAL_SERVO_NOT_STARTED
bool HapticsClass::isDeviceCalibrated()
{ {
    unsigned int state = hdlGetState();

    return ((state & HDAL_NOT_CALIBRATED) == 0);
}
```

7.4 Main_opengl.cpp

```
// Comment the following line to get a console window on startup
#pragma comment( linker, "/subsystem:\"windows\" /entry:\"mainCRTStartup\"" )

// windows.h must precede glut.h to eliminate the "exit" compiler error in VS2003, VS2005
#include <windows.h>
#include "glut.h"
#include <math.h>
#include "haptics.h"

// Cube parameters
const double gStiffness = 5000.0;
const double gCubeEdgeLength = 0.05;

// Some OpenGL values
static GLuint gCursorDisplayList = 0;
static double gCursorRadius = 0.003;
static GLfloat colorRed[]  = {1.0, 0.0, 0.0};
static GLfloat colorTeal[] = {0.0, 0.5, 0.5};
static GLfloat* gCurrentColor;

// The haptics object, with which we must interact
HapticsClass gHaptics;

// Forward declarations
void glutDisplay(void);
void glutReshape(int width, int height);
void glutIdle(void);
void glutKeyboard(unsigned char key, int x, int y);
void exitHandler(void);
void initGL();
void initScene();
void drawGraphics();
void drawCursor();

int main(int argc, char *argv[])
{ {

    // Normal OpenGL Setup
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

```
    glutInitWindowSize(500, 500);
    glutCreateWindow("Basic--OpenGL");
    glutDisplayFunc(glutDisplay);
    glutReshapeFunc(glutReshape);
    glutIdleFunc(glutIdle);
    glutKeyboardFunc(glutKeyboard);

    // Set up handler to make sure teardown is done.
    atexit(exitHandler);

    // Set up the scene with graphics and haptics
    initScene();

    // Now start the simulation
    glutMainLoop();

    return 0;
}

// drawing callback function
void glutDisplay()
{
    drawGraphics();
    glutSwapBuffers();
}

// reshape function (handle window resize)
void glutReshape(int width, int height)
{ {
    static const double kPI = 3.14159265358979323846264338327795;
    static const double kFovY = 40;

    double nearDist, farDist, aspect;

    glViewport(0, 0, width, height);

    nearDist = 0.075 / tan((kFovY / 2.0) * kPI / 180.0);
    farDist = nearDist + 0.05;
    aspect = (double) width / height;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(kFovY, aspect, nearDist, farDist);

    /* Place the camera down the Z axis looking at the origin */
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0, 0, nearDist + 0.06,
              0, 0, 0,
              0, 1, 0);
}

// What to do when doing nothing else
void glutIdle()
{ {
    glutPostRedisplay();
}


// Handle keyboard.  In this case, just the Escape key
void glutKeyboard(unsigned char key, int x, int y)
{ {
    static bool inited = true;

    if (key == 27) // esc key
    {
        exit(0);
    }
}

// Scene setup
```

```
void initScene()
{ {

    // Call the haptics initialization function
    gHaptics.init(gCubeEdgeLength, gStiffness);

    // Set up the OpenGL graphics
    initGL();

    // Some time is required between init() and checking status,
    // for the device to initialize and stabilize.  In a complex
    // application, this time can be consumed in the initGL()
    // function.  Here, it is simulated with Sleep().
    Sleep(100);

    // Tell the user what to do if the device is not calibrated
    if (!gHaptics.isDeviceCalibrated())
        MessageBox(NULL,
                   // The next two lines are one long string
                   "Please home the device by extending\n"
                   "then pushing the arms all the way in.",
                   "Not Homed",
                   MB_OK);


}

// Set up OpenGL.  Details are left to the reader
void initGL()
{ {
    static const GLfloat light_model_ambient[] = {0.3f, 0.3f, 0.3f, 1.0f};
    static const GLfloat light0_diffuse[] = {0.9f, 0.9f, 0.9f, 0.9f};
    static const GLfloat light0_direction[] = {0.0f, -0.4f, 1.0f, 0.0f};

    /* Enable depth buffering for hidden surface removal. */
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);

    /* Cull back faces. */
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);

    /* Set lighting parameters */
    glEnable(GL_LIGHTING);
    glEnable(GL_NORMALIZE);
    glShadeModel(GL_SMOOTH);

    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE);
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, light_model_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light0_direction);
    glEnable(GL_LIGHT0);
}

// Make sure we exit cleanly
void exitHandler()
{ {
    gHaptics.uninit();
}

// Draw the cursor and the cube.  In a real application,
// this function would be much more complex.
void drawGraphics()
{ {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawCursor();
    glutSolidCube(gCubeEdgeLength);
}
```

```
// Draw the cursor
void drawCursor()
{ {
    static const int kCursorTess = 15;

    // Haptic cursor position in "world coordinates"
    double cursorPosWC[3];

    // Must synch before data is valid
    gHaptics.synchFromServo();
    gHaptics.getPosition(cursorPosWC);

    // The color will depend on the button state.
    gCurrentColor = gHaptics.isButtonDown() ? colorRed : colorTeal;

    GLUquadricObj *qobj = 0;

    glPushAttrib(GL_CURRENT_BIT | GL_ENABLE_BIT | GL_LIGHTING_BIT);
    glPushMatrix();

    // Page 505-506 of The Red Book recommends using a display list
    // for static quadrics such as spheres
    if (!gCursorDisplayList)
    {
        gCursorDisplayList = glGenLists(1);
        glNewList(gCursorDisplayList, GL_COMPILE);
        qobj = gluNewQuadric();

        gluSphere(qobj, gCursorRadius, kCursorTess, kCursorTess);

        gluDeleteQuadric(qobj);
        glEndList();
    }

    glTranslatef(cursorPosWC[0], cursorPosWC[1], cursorPosWC[2]);

    glEnable(GL_COLOR_MATERIAL);
    glColor3fv(gCurrentColor);

    glCallList(gCursorDisplayList);

    glPopMatrix();
    glPopAttrib();
}
```