HapticMASTER Programming Manual

MOOG B.V.	Visit/shipping address	Phone	+31 (252) 462 065
P.O. Box 187	Pesetaweg 53	fax	+31 (252) 462 001
2150 AD Nieuw-Vennep	2153 PJ Nieuw-Vennep	email	robotics@moog.com
The Netherlands.	The Netherlands.		http://www.moog.com

Contents

1 About This Manual

The HapticAPI is a software interface to the HapticMASTER Hardware. It allows you to create interactive programs which create three-dimensional haptic objects and haptic effects that produce realistic virtual worlds. This manual explains how to program with the HapticAPI interface in order to generate the virtual worlds you want.

What This Manual Contains

This manual has fifteen chapters. Most of the chapters present one topic of HapticAPI programming by showing and explaining the source code of an example. The first five chapters present basic information about programming the HapticMASTER using the HapticAPI. Simple application programs are provided in source code in four chapters and a step by step explanation of this source code is provided. A fifth chapter provides an example which helps understanding the HapticMASTER's states

2, Introduction
This chapter provides information about the HapticMASTER device and its hardware. Some details are presented that you need to know for the subsequent chapters.

3, Hello Haptic World
Here a first 'Hello World' style application is presented that you find in most books about a programming language. The program is explained step by step.

4, Hello Graphic World
This example adds a graphic equivalent of the haptic world. OpenGL code is used to draw a haptic cube, the HapticMASTER's workspace and axes, and the End Effector. Many OpenGL utility functions are used in this example. The program is explained step by step.

5, HapticMASTER Workspace
In this chapter the boundaries of the HapticMASTER workspace are described. This may help you to get an understanding of where in this workspace your haptic world resides.

6, HapticMASTER States
From the point the HapticMASTER is switched on it can go through several states. This chapter provides all the necessary information about these states.

The next three chapters continue to extend your knowledge of programming the HapticMASTER using the HapticAPI by providing more in-depth examples of haptic effects and haptic objects.

7, More Haptic Objects
Until this point only primitive blocks have been used in the sample programs. In this chapter other shapes are introduced.

8, Haptic Effects
In this chapter, the so called haptic effects are introduced. A sample program is given that presents the spring effect.

9, More Haptic Effects
Besides the spring effect, introduced in chapter **8**, **Haptic Effects,** there are some more haptic effects. This chapter describes them.

The chapters 10 through 15 introduce some advanced techniques. Two possible uses of the HapticMASTER are presented that are not quite obvious but nevertheless very powerful – the position control.
For the example in chapter 13, you need two HapticMASTER robots.

10, Force Measurement
The HapticMASTER is equipped with a very sensitive force sensor. The actual current values of this sensor can be read through the HapticAPI, which enables us to query the force measurements.

11, Position Control – Mouse-slave
Although the HapticAPI offers no direct way to move the HapticMASTER to a certain position from an application program, this example presents a workaround that lets you control the HapticMASTER position e.g. using a computer mouse as input device for the position the HapticMASTER obtains.

12, Position Control – Orbiting in a circle
Another position control example is letting the End Effector follow a circle orbit, which is demonstrated in a separate example as well.

13, Master Slave operation
This is a very nice example of running two HapticMASTER robots in a Master-Slave setup. It shows how two HapticMASTERs, coupled only by software, follow each other's movements in a very fast and accurate way.

14, Matlab programs
Repeats the program introduced in chapter 3, only this time we write the program using Matlab instead of C++. Issues regarding interfacing the

HapticMASTER via Matlab are thoroughly described.

15, Real-time Data Logger
The final chapter describes a feature of the HapticMASTER software that allows you to log data directly from the HapticMASTER in extremely high frequencies. You can save the logged data in an output file and later process it according to your needs.

After you have seen all the examples and, hopefully, played with them, you should be able to create your own haptic worlds by using the HapticAPI.

What You Should Know Before Reading This Manual

In this manual the assumption is made that you know how to program using the C/C++ programming language. There are many good books about this programming language, so you won't find any explanation whatsoever about that in this manual.

We believe it is much easier to create haptic virtual worlds and have a graphic equivalent on your computer monitor to see what is going on when moving (and feeling) around with the HapticMASTER. For this reason, almost all the example programs presented in this manual have a piece of OpenGL code that renders a graphics scene on your monitor that displays the objects which are created inside the HapticMASTER real-time computer. Although the OpenGL code used in the examples is really straightforward, it is definitely useful to have some knowledge of OpenGL. In the program sources, you will find lots of code comments to help understand what is going on, but again, there will be no explanation of OpenGL. Some background in mathematics can be of good use as well.

Caution: This is the appropriate moment to consider a few aspects of programming the HapticMASTER. The HapticMASTER is capable of rendering very high forces depending on the spring stiffness of the objects created through the HapticAPI. The programmer should be very careful at all times when using high spring stiffnesses for the haptic objects created. The HapticMASTER may make unexpected swift movements when no care is being taken. However, if the HapticMASTER 'wakes up' inside an object, it will make a very gentle movement in order to get out of the object.

THEREFORE, MAKE SURE TO HAVE A WORKSPACE CLEAR OF ANY OBSTACLES BEFORE TESTING OR STARTING UP A PROGRAM ON THE HAPTICMASTER. THIS REDUCES THE CHANCE OF DANGEROUS SITUATIONS DUE TO STARTUP ISSUES OR PROGRAMMING ERRORS!

Running the examples

In this manual there are many programming examples. The source code of which, is presented in the respective chapters. For your convenience the source code of these examples is also provided on the CD-ROM accompanying a HapticMASTER purchase or an API update.

The source code of the examples is created and compiled using two environments:

1)**Microsoft Visual C++ 2010 IDE.**
The examples in this variant use the HapticAPI.dll file, which is to be used from Visual Studio and Matlab. This DLL is only for 32-bit applications.

2)**Ubuntu 12.04, running G++ version 4.6.**
The examples in this variant use the libHapticAPI-x86_32.0.0.0.so (for 32 bits Linux) or libHapticAPI-x86_64.0.0.0.so (for 64 bit Linux)

Examples 12 and 14 are intended for use with Matlab and use the HapticAPI.dll.

There are some basic system requirements for running the examples on your machine:

- Processor: An Intel Pentium (or higher)
- Operating System: Windows XP/Vista, Windows 2000, or Windows 7
  Linux, with a recent version of the GNU C Compiler
- Ethernet card: 10/100 Mbit Ethernet network card
- Graphics Card: With OpenGL support

Running an example from Visual Studio

1)Open the .SLN file in the examples folder
➔ Visual Studio opens with the examples solution. Each example is a project within this solution.
2)Open the Source Files folder in the project and double-click on the CPP file.
3)Change the IP address to the one corresponding with your HapticMASTER's IP address.
4)Recompile and link the project by pushing F7. No compile or link errors should appear.
5)Finally, set the desired project to be the start-up project – right-click the desired project and choose "Set as StartUp Project" from the pull-down menu.
6)Push F5 to run the example.

Running an example from MatLab (Only 32-bit versions of MatLab)

Preperation work

1)Start MatLab
2)Execute the command ***mex –setup***
3)Choose ***y***, to let mex locate the C/C++ compilers for you
4)From the list of compilers (if any installed), choose a C/C++ compiler.
➔ This compiler choice is needed for MatLab to correctly interpret the HapticAPI.h and HapticAPI.dll files supplied by MOOG.

Running an example

1)From MatLab, navigate to either example 12's or example 14's folder.
2)Open the corresponding .M file (example_12.m or example_14.m)
3)Change the IP address string to your HapticMASTER's IP address.
4)Run the example by giving the command example_12 or example_14 (respectively)

Running an example from Linux

Preperation work

1) Make sure the following packages are installed on your Linux OS (to support OpenGL):
   a. sudo apt-get install freeglut3-dev
   b. sudo apt-get install libgl1-mesa-dev
   c. sudo apt-get install libgfw-dev
   d. sudo apt-get install libglu-dev
2) Copy the contents of the Demos and Examples CD to your linux environment or use a shared library between Windows and Linux.
3) GCC / G++ version should be 4.6 (the Shared Libraries are compiled with version 4.6). Newer versions of GCC / G++ will possibly work as well but we can not guarantee any backwards compatability w.r.t the version that was used to compile the libraries.
   Should you encounter any compatibility issues, please conact MOOG.

Running an example

1)Enter the ***Examples*** folder.
2)Change the IP Address string to your HapticMASTER's IP Address.
3)You can **compile all examples** at once running one of the following scripts:
   ./makeall x86_64 If your linux is 64-bit
   ./makeall x86_32 If your linux is 32-bit
4)You can also enter an **individual example** folder and run one of the following commands from there:
   make x86_64 If your linux is 64-bit
   make x86_32 If your linux is 32-bit
5)Enter the individual example folder and run e.g.:
   ./01-Hello-Haptic-World

➔ The example is being run. If applicable for this example, an OpenGL window will open as well.

Style Conventions

The following style conventions are used in this programming manual:

Regular	Normal text descriptions.
Italics	Names of function calls, function parameters and variables
Bold	References to chapters, figure labels, websites etc. or other items that demand more attention.
`Source Code`	Lines of source code.
`Source Code`	Lines of **HapticAPI** source code.
`// Source Comments`	Lines of source code comments.

2 Introduction

Before running off to chapter 3 (Hello Haptic World), for the first hands-on experience with the HapticMASTER, we would like to introduce you to the haptic world terminology and the HapticMASTER's basics.
The first part of this chapter will discuss the HapticMASTER device, its hardware components and the (local) network. The second part discusses some details you need to know for programming HapticMASTER applications.

Haptics, Haptic Rendering, HapticMASTER

The HapticMASTER is a small robot specially designed for interacting with the human hand. It has a very smooth feel, and a sensitive force sensor. Together with the software, this makes it eminently suitable for simulating virtual or remote worlds, conveying the full spectrum of forces right from the sense of free motion to that of touching a very stiff wall.

The word *"Haptics"* is the modern term for "force feedback" or "artificial feel". The word originates from Greek and means "touching". The "master" part of the name is there for historical reasons. Early haptic devices were typically used for remote *master-slave* operations, so the second part of the name originates from this haptic application. Actually, "haptic display" would be more appropriate, since the device conveys the feel of a virtual (or remote) environment to the user, much like a video display conveys the sense of sight of a virtual or remote scene.

Like in the world of visual display software, there are special ways of simulating a virtual environment. The algorithms involved are commonly called *"rendering" algorithms*, because they attempt to "render" virtual worlds as realistically as possible to the human user. The HapticMASTER software has "haptic" rendering algorithms in the same way that a graphics card or library will have "visual" rendering algorithms.

Unlike many other haptic devices, the HapticMASTER integrates a **force sensor** into the control loop to get very high-quality haptic rendering. The advantage of this method is that the actual mass and friction of the device can be eliminated. A small amount of virtual mass is always needed though, in order to avoid commanding infinite accelerations. The resulting control algorithm is known as *"admittance control"*. The basic control paradigm for an admittance controlled device like the HapticMASTER is:

- Force in, displacement out.

The real-time controller or *"haptic server"* runs the following algorithm at an update rate of 2048 Hz. See Figure 2-1 : System Block Diagram.

- **Read the force** sensor to reveal the external force exerted by the user.
- **Add all virtual forces** due to virtual objects and effects.
- **Calculate** the resulting **virtual motion** in an internal virtual model.

- **Command the HapticMASTER** to move according to the virtual model.

The “force in, displacement out” paradigm is the reverse of that used in many other devices on the haptic market. Most of these devices use *“impedance control”*, where the device measures the displacement given by the user, and reacts with a force. These devices are usually very lightly built and mechanically back drivable, because their control algorithm cannot compensate for the mass and friction of the device itself.

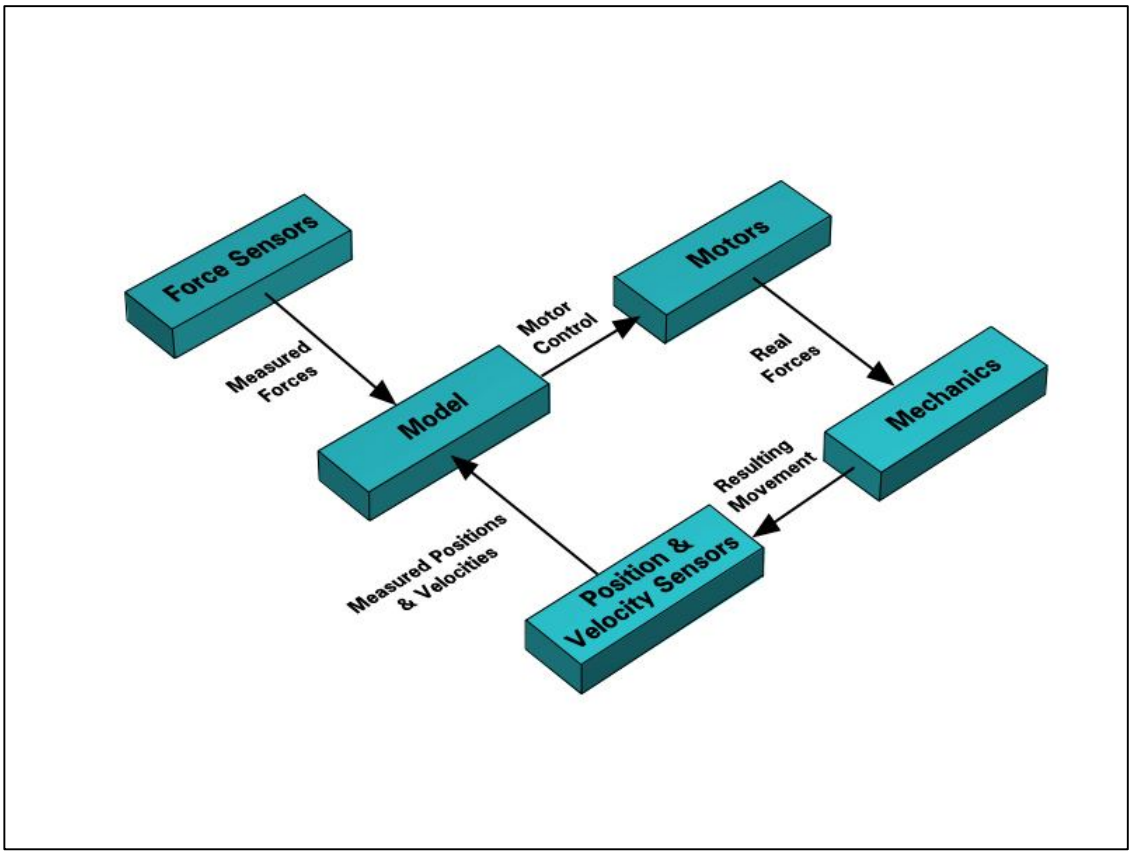


Figure 2-1 : System Block Diagram

Programming the HapticMASTER

Programming the HapticMASTER is really straightforward, which for a great deal, is due to the HapticAPI programming interface. This interface, which is supplied with each HapticMASTER purchase, contains all the functionality to connect to a HapticMASTER, create haptic objects and effects, and access all data on displacements (positions), velocities and acceleration of the machine, but also all data on measured forces. You might also want to take a look at the API reference manual called “The HapticAPI Reference Manual” which describes all the available objects and commands.

At the end of the day, we want to be able to create virtual worlds with 3D objects and effects that can be felt and touched while you wander around with the HapticMASTER’s End Effector. The End Effector is the ball mounted on the HapticMASTER’s force sensor. You can compare the End Effector with a computer mouse. Moving the mouse, results in moving the mouse pointer on the screen. Moving the HapticMASTER’s End Effector (by pushing or pulling in a specific direction), results in moving the End Effector in the virtual world. While moving the End Effector around the workspace, the forces that are applied at this moment are calculated. As described in the previous paragraph this is known as *haptic* rendering. When the End Effector touches an object in the virtual world, you should feel the resistance as if you actually touch the object. The HapticMASTER is used as both input and output device. You supply movement inputs by touching the End Effector, and at the same time you feel the forces that apply on the End Effector at that same moment.

Programming the HapticMASTER is about building the virtual world in which the End Effector can be moved. The HapticAPI provides some basic functions to open a connection to the device, close it and send string commands to it. By sending the string commands you can build the virtual world (create objects, effects, set the haptic properties thereof etc.), change the current state of the HapticMASTER and query information from the device (position, velocity, forces etc.).
In this programming manual we intend to show you enough examples so that you can use the complete functionality of the HapticAPI in order to program the HapticMASTER.

3 Hello Haptic World

In most books about programming languages, the first example you will find is some variation of the well known “Hello World” program. So, to continue the ancient tradition, this manual will also present such an example.

The example in this chapter uses very minimal code to create a simple haptic application program.

As you look at example “01-Hello Haptic World” on the next pages you will find that there are very few code lines (the underlined ones) that are needed to create the virtual haptic world on the HapticMASTER device. The lines that are not underlined are common C++ code that has little to do with the HapticMASTER device.

First, take a look at the next two pages of the example source code. On the pages following the code, a step by step explanation of the code will be given.

Example 1 - Source Code

Example 1 : Hello Haptic World

```
//---------------------------------------------------------------
//        0 1  -  H E L L O   H A P T I C   W O R L D
//
// This example demonstrates how to use the HapticAPI
interface in order
// to communicate with the HapticMASTER device.
// The example creates a sphere at the center of the
HapticMASTER's
// workspace. A spring constantly tries to bring the End Effector
// to the center, but the End Effector should stop on the surface
of the
// created sphere. The user should be able to "feel" the sphere
by moving
// the End Effector around the workspace.
//---------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[300];

//---------------------------------------------------------------
//                    M A I N
//
// 01-Hello-Haptic-World Main Function
```

```
    // No error response from command execution, go further
with settings
    else {
        haSendCommand(dev, "set mySphere pos [0,0,0]",
response);
        printf("set mySphere pos ==> %s\n", response);

        haSendCommand(dev, "set mySphere radius 0.05",
response);
        printf("set mySphere radius ==> %s\n", response);

        haSendCommand(dev, "set mySphere stiffness 20000",
response);
        printf("set mySphere stiffness ==> %s\n", response);

        haSendCommand(dev, "set mySphere enable", response);
        printf("set mySphere enable ==> %s\n", response);
    }

    if ( haSendCommand (dev, "create spring mySpring",
response) ) {
        printf("failed sending command: create spring
mySpring\n");
        getchar();
        exit(-1);
    }

    printf("create spring mySpring ==> %s\n", response);

    // Error response returned from command execution
    if ( strstr(response, "--- ERROR:" ) ) {
        getchar();
        exit(-1);
    }
```

Example 1 - Description

On the previous two pages is the source code of the first example of this manual. Next is a step by step description of this code.

Include and Define Statements

First of all let's take a look at the include statements at the beginning of the example code. There are two include statements.
The "*HapticAPI.h*" includes the common API functions. *"HapticMaster.h"* includes some utility functions (which are not compulsory but make the programming task easier).
Please refer to chapter Appendix A for further explanations about the functions you can use from "*HapticMaster.h"*.

Next, we define the IP-Address of the HapticMASTER we are about to interface, in our case this is the string "10.30.203.10**"**. Change this string to the IP address your HapticMASTER is assigned.

Using the 3 define statements *PosX*, *PosY* and *PosZ* helps us with making the code more readable by replacing the 0, 1 and 2 indices of arrays with these names.

Global variables

The *dev* variable is a 'long integer' that holds a handle to the device. You can think of this handle as a variable that identifies the device you are going to work with for this session. Once the HapticMASTER device is created by the *haDeviceOpen()* function, the handle is placed in this variable, and any communication with the HapticMASTER from this moment on is done by supplying this device identifier. The variable *response* is a C-String variable that holds the response of the HapticMASTER when it is given a command – this can be either a success message or an error message.
Should this be an error message, it will start with the text: "--- ERROR: "

main() function

The code in this function creates the HapticMASTER device by calling the *haDeviceOpen()* function. This function returns an identifier to the HapticMASTER device, which, from this point on, is used to reference the HapticMASTER device. If opening the device fails, the value HARET_ERROR (defined in the file *HapticAPI.h*) is returned. In this case, an error message is printed and the program terminates.
If the device was successfully opened, the *InitializeDevice()* function is called. This function is declared in *HapticMaster.h* (thus a utility function) and initializes the HapticMASTER. You can choose to use this function 'as is', or write the code yourself. Either way, if the HapticMASTER encoders are not yet calibrated, **you must initialize the HapticMASTER before you can use it.** See Chapter Appendix A for more details about the *InitializeDevice()* function.

From the moment that the HapticMASTER device is opened, all commands are sent via the *haSendCommand()* function. This function is defined in HapticMaster.h as well.
The first parameter is the device identifier (variable *dev*).
The second parameter is the command itself. This is a C-String so it is surrounded by a double quote.
The third parameter that you should supply is the C-String variable that will hold the response from the HapticMASTER – this is the variable *response* in our case.

The function *haSendCommand()* returns the integer value 0 if the command was **sent successfully**. If something went wrong with sending the command, an integer value unequal to 0 will be returned. You can surround the *haSendCommand()* function call with an IF statement, to check whether **sending the command was successful or not**.

The *haSendCommand()* function sends the supplied command to the HapticMASTER, which executes this command and returns the response in the *response* C-String.
If the returned string starts with "--- ERROR: ", then there was **a problem with executing** the command in the Real-time computer.

You will notice that if you want to play safe, you will have two IF-statements around the haSendCommand function. The first one will check if sending the command was successful, the second one will check if the execution of the command in the Real-time computer was successful:

```
if (haSendCommand ( dev, "set state force", response) ) {
   printf("--- ERROR: Could not send command set state force");
   getchar();
   exit(-1);
}

printf("set state force ==> %s", response);

if ( strstr ( response, "--- ERROR:" ) ) {
   getchar();
   exit(-1);
}
else {
   … Go further …
}
```

haSendCommand() is called. If it returned with a value other than 0 (there was an error with sending the command), we print an error message, wait for a key press and quit.

Otherwise (sending was successful), print the response returned from the real-time computer. We then check if the string "--- ERROR:" is found inside the

returned string *response*. If this string is found in the response, we wait for a key press and quit.
If the string was not found in the response string, we go further with the rest of the commands.

This is a pattern you can repeat each time you send a command to the HapticMASTER. If you want to make the code shorter and you do not need to detect errors, you can simply write code that looks like this:

```
haSendCommand ( dev, "set state force", response);
```

This will also work of course, but you cannot guarantee the command was sent and that execution of the command succeeded.

Our example creates a haptic sphere, by calling *haSendCommand()* with the "create sphere mySphere" command. *sphere* is the type of the object that we are creating, and *mySphere* is its name. The *haSendCommand()* call is surrounded by an IF statement. If sending the command fails, an error message is generated. If it succeeds, the sphere haptic effect with the name *mySphere* can be addressed from now on and we go further with the else part of the IF statement.

There, we set some parameters for the recently created sphere *mySphere*:

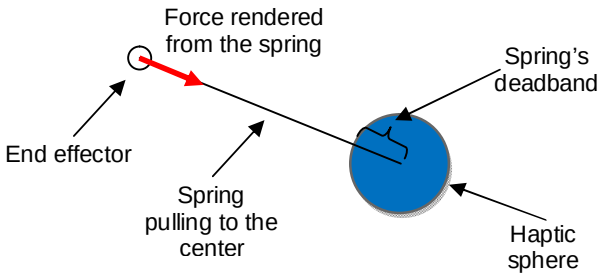
- *"set mySphere pos [0,0,0]"* sets the **position** of the sphere. Notice that every time you supply a vector in the command syntax, it should be surrounded by the '[' and ']' characters.
- *"set mySphere radius 0.05"* sets the **radius** of the sphere (which is a scalar, dimensions in meters).
- *"set mySphere stiffness"* sets the **stiffness** of the sphere – in other words how hard is the haptic sphere.
- *"set mySphere enable"* **enables** the haptic effect. If the effect is not enabled, it is not rendered in the real-time software, which means one cannot feel this object in the virtual haptic world.

What follows is the creation of the spring.
We create the spring by calling *haSendCommand()* with the command "create spring mySpring". A *spring* effect is then created with the name *mySpring*. Consequently, if the creation of the spring succeeds, we set-up the spring's properties:

- *"set mySpring pos [0,0,0]"* sets the **position** of the spring. This is the position that the spring pulls into.
- *"set mySpring stiffness 100.0"* sets the **stiffness** of the spring. This determines how much force the spring applies in relation with the distance of the End Effector from the spring's position in Newton per meter.
- *"set mySpring deadband 0.05"* sets the range (in meters) from the spring's position, where the spring applies no forces. We set this range to 0.05, so the spring is not pulling anymore when the End Effector is touching the sphere's surface.
- *"set mySpring enable"* enables the spring.

While running this example, you can move the End Effector around, by applying force on the ball mounted on the HapticMASTER's force sensor. The spring effect pulls the End Effector in the direction of [0,0,0] (the center of the workspace). However, the spring is set to a deadband of 0.05m. That means the spring stops pulling when it touches the sphere's surface (which has a radius of 0.05m around the center of the workspace). You can actually feel the sphere because it is impossible to get into it with the End Effector. The sphere's stiffness is set quite high (20,000 N/m), so it really feels hard and you can touch the object only from the outside. You can move the End Effector further away from the sphere, and the spring pulls the End Effector again in the direction of the sphere.

Putting this setting in a drawing might make things clearer:



Well, this concludes our explanation of the first example. Congratulations! You've just created your first HapticMASTER application. The next example adds the graphical variant of the haptic world. On a computer screen, you can actually see the haptic world that you feel by moving the End Effector.

4 Hello Graphic World

In the previous example we already introduced you to the HapticMASTER's world and the way programs can communicate with the device. For some applications, it is indeed enough to simply build the haptic world and let a user feel the world without seeing anything. However, the average application will also need to visualize the haptic world it builds within the HapticMASTER's real-time computer. In this example we add the graphic variant of the haptic world. To do this, lots of extra OpenGL code is needed. Don't be scared by the next couple of pages of code as it is really not that difficult to understand. The part of the code that actually has to do with the HapticAPI is underlined, just like in the previous example.

As you look at example "02 - Hello Graphic World" on the next pages you will find that there are very few code lines (the underlined ones) that are needed to create the virtual haptic world on the HapticMASTER device. The lines that are not underlined are necessary for initializing the OpenGL engine, creating the window and displaying the virtual world in that window on the computer monitor.

First, take a look at the next five pages of the example source code. On the pages following the code, a step by step explanation of the code will be given.

Example 2 - Source Code

Example 2 : Hello Graphic World

```
//---------------------------------------------------------------------
//         0 2   -   H E L L O   G R A P H I C   W O R L D
//
// This example demonstrates how to use the OpenGL environment in order
// to draw the HapticMASTER's axes, workspace and a haptic block object.
// This example uses a Timer callback function, in which the position
// of the end effector is being queried periodically.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double CurrentPosition[3];

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Block OpenGL Material Parameters.
GLfloat BlockAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat BlockDiffuse[] = {0.00, 0.80, 0.67, 1.00};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

GLfloat SpecularOff[] = {0.00, 0.00, 0.00, 0.00};
GLfloat EmissiveOff[] = {0.50, 0.50, 0.50, 0.00};
GLfloat ShininessOff = {0.00};

//---------------------------------------------------------------------
//                O B J E C T   P A R A M E T E R S
//---------------------------------------------------------------------
double blockSize[3] = {0.15,0.15,0.15};
double blockPos[3] = {0.0,0.0,0.0};
double blockStiffness = 20000;
```

```
//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                   B L O C K   M A T E R I A L
//
// BlockMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The Block.
//---------------------------------------------------------------------
void BlockMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlockAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlockDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, SpecularOff);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, ShininessOff);
}

//---------------------------------------------------------------------
//                      D R A W   B L O C K
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Haptic Block Object In OpenGl
//---------------------------------------------------------------------
void DrawBlock(void)
{
   glPushMatrix();
   glTranslatef(blockPos[PosX], blockPos[PosY], blockPos[PosZ]);
   glScalef(blockSize[0], blockSize[1], blockSize[2]);
   BlockMaterial();
   glutSolidCube(1.0);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                        I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{

   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_NORMALIZE);

   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//---------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//---------------------------------------------------------------------
void Display (void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   // define eyepoint in such a way that
   // drawing can be done as in lab-frame rather than sgi-frame
   // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);
   DrawEndEffector();
   DrawBlock();

   glPopMatrix ();
   glutSwapBuffers();
}
```

```
//-----------------------------------------------------------------------
//                          R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//-----------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
   glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();

   float fAspect = (float)iWidth/iHeight;
   gluPerspective (30.0, fAspect, 0.05, 20.0);

   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
}

//-----------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//-----------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         if ( haDeviceClose(dev) ) {
            printf("--- ERROR: closing device\n");
         }
         exit(0);
         break;
   }
}

//-----------------------------------------------------------------------
//                        T I M E R   C B
//
// This Is A Timer Function Which Call The HapticMASTER To Get The
// New EndEffector Position
//-----------------------------------------------------------------------
void TimerCB (int iTimer)
{
   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if (strstr(response, "--- ERROR:") ) {
      printf("get modelpos ==> %s", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
CurrentPosition[PosZ] );
   }

   // Set The Timer For This Function Again
   glutTimerFunc (10, TimerCB, 1);
}
```

```
//---------------------------------------------------------------------
//                            M A I N
//
// 02-Hello-Grpahic-World Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Open the HapticMASTER device
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      // Try creating a block object
      if ( haSendCommand(dev, "create block myBlock", response) ) {
         printf("failed sending command: create block myBlock\n");
         getchar();
         exit(-1);
      }

      printf("create block myBlock ==> %s\n", response);

      // Error response returned from command execution
      if ( strstr(response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }

      // No error response from command execution, go further with settings
      else {
         haSendCommand(dev, "set myBlock pos", blockPos[PosX], blockPos[PosY],
                       blockPos[PosZ], response);
         printf("set myBlock pos ==> %s\n", response);

         haSendCommand(dev, "set myBlock size", blockSize[0], blockSize[1], blockSize[2],
                       response);
         printf("set myBlock size ==> %s\n", response);

         haSendCommand(dev, "set myBlock stiffness", blockStiffness, response);
         printf("set myBlock stiffness ==> %s\n", response);

         haSendCommand(dev, "set myBlock enable", response);
         printf("set myBlock enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example02: Hello Graphic World");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutTimerFunc (10, TimerCB, 1);
      glutMainLoop();
   }
   return 0;
}
```

Example 2 - Description

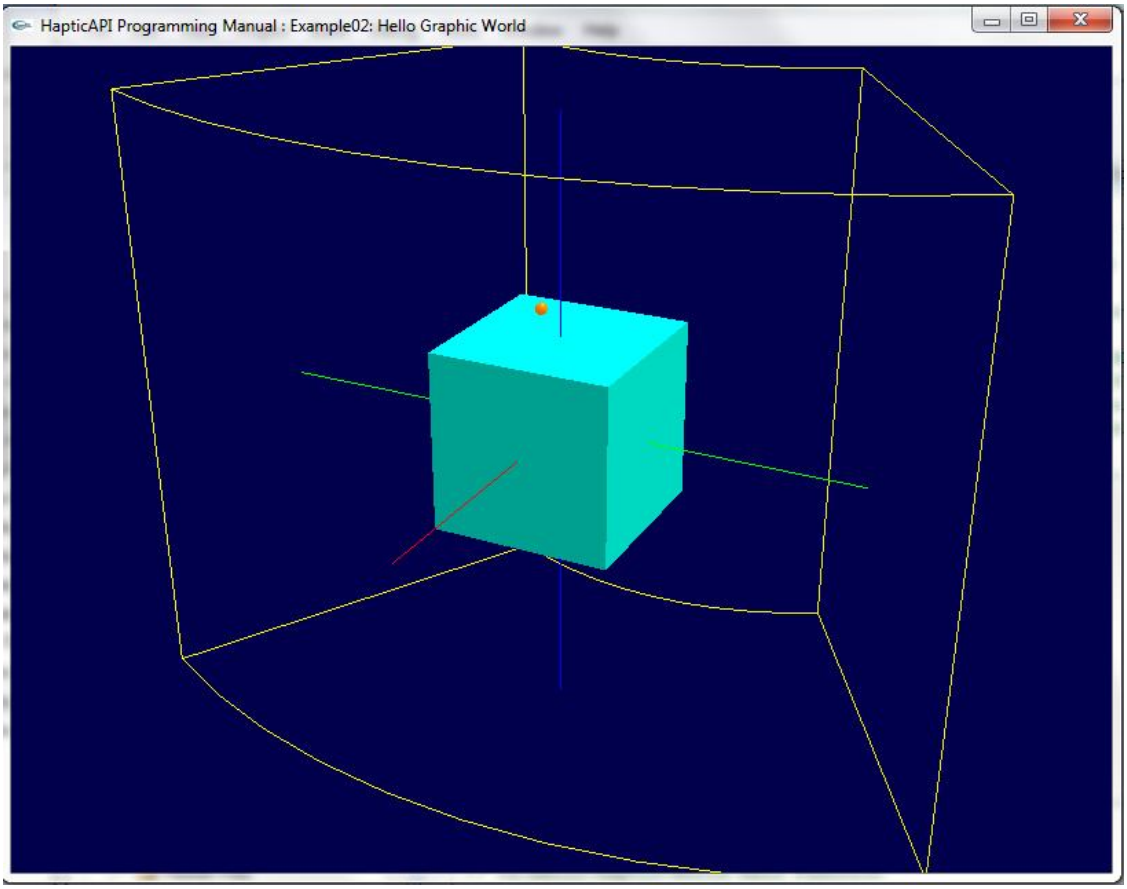


Figure 4-1 : Example 2 Screen Shot

On the previous five pages is the source code of example “02 – Hello Graphic World”. Above is a screenshot of the example program while running. The yellow frame represents the boundaries of the HapticMASTER’s workspace. The green, red and blue lines are the X, Y and Z axes respectively. The haptic block can be seen in the center of the workspace. Finally, the End Effector as a small orange ball indicating its position in the workspace. Next is a step by step description of this code.

Include and Define Statements

First of all let’s take a look at the include statements at the beginning of the example code. There are three include statements.
The “*HapticAPI.h*” includes the common API functions. *"HapticMaster.h"* includes some utility functions (which are not compulsory but make the programming task easier). “*HapticMasterOpenGl.h"* includes some utility functions with respect to the OpenGL drawing.

Next, we define the IP-Address of the HapticMASTER we are about to interface, in our case this is the string "10.30.203.10**"**. Again, change this string to the IP address corresponding to your HapticMASTER.

Using the 3 define statements *PosX*, *PosY* and *PosZ* helps us with making the code more readable by replacing the 0, 1 and 2 indices of arrays with these names.

Global variables

The *dev* variable is a 'long integer' that holds a handle to the device. You can think of this handle as a variable that identifies the device you are going to work with. Once the HapticMASTER device is created by the *haDeviceOpen()* function, the handle is placed in this variable, and any communication with the HapticMASTER from this moment on is done by supplying this device identifier. The variable *response* is a C-String variable that holds the response of the HapticMASTER when it is given a command – this can be either a success message or an error message. *CurrentPosition* is an array of 3 floats that keeps the current position of the End Effector. Since we constantly need to update the End Effector's position on the OpenGL window, we need to keep the position in this array.

What follows is the declaration and initialization of some OpenGL material variables. You don't have to pay much attention to this. The only thing this does is create a solid green material for the block and an orange plastic kind of material for the graphic equivalent of the HapticMASTER End Effector when they are drawn on the computer monitor. The functions *EndEffectorMaterial()* and *BlockMaterial()* are responsible for setting the correct OpenGL drawing color parameters for these two visual objects.

Draw Functions

The functions *DrawEndEffector()* and *DrawBlock()* are responsible for the actual drawing of these objects on the screen. It consists of:

- "Translating" the coordinate system to the position where the object will be drawn
- "Scaling" X, Y and Z coordinates to achieve the correct object size
- Setting the material color
- Drawing the object using the GL utility (GLUT)

InitOpenGl()

The *InitOpenGl()* function initializes the OpenGL graphics engine to create a simple yet efficient enough graphical environment for displaying the virtual world scene on the computer monitor. It sets up the shading model, some lighting in the scene and the blending function.

Display() and Reshape()

The function *Display()* positions the OpenGL viewing point through the *gluLookAt()* call and calls *DrawEndEffector()* and *DrawBlock()*. The *Display()* function is called periodically by OpenGL to redraw the frame. The *Reshape()* function takes care of maintaining the correct aspect ratio when the window is resized. The *Reshape()* function is called by the OpenGL engine when needed.

Keyboard()

Keyboard() is called by the OpenGL engine whenever a key was hit on the keyboard. This function is used here to delete all the haptic objects created on the HapticMASTER real-time computer and after that set back the HapticMASTER to the '**stop state'**. The 'stop state' gradually brings the HapticMASTER End Effector to a stop and holds this position. In the STOP state, it is impossible to move the HapticMASTER from its position.

haSendCommand() and haDeviceSendString() calls

The HapticAPI provides one function that takes care of sending commands to the HapticMASTER: The haDeviceSendString() function (see chapter **Error! Reference source not found.**). Inside the include file HapticMaster.h we define 5 overloaded functions with the name haSendCommand() which all make use of the haDeviceSendString() to send the command to the HapticMASTER. The functions haSendCommand() provide you with 5 different ways you can construct the command you want to send to the HapticMASTER. Each variant of this function expects a different setting of parameters.
In all examples presented in this manual we use the haSendCommand() functions. Please refer to section 0 for more details about the relation between the haDeviceSendString() function and the haSendCommand() functions.

In our example, the first haSendCommand() function call you see is inside the function Keyboard(). There, we handle the ESC key press. If the user presses the ESC key on the keyboard, we want to remove all haptic effects and make a transition to the state STOP in the HaptcMASTER.

The first parameter is the device identifier (variable *dev*), then the string command you want to send to the HapticMASTER. The third parameter that you must supply is the C-String variable that will hold the response from the HapticMASTER – this is the variable *response* in our case. Make sure you declare this C-string with enough space to hold the response messages.

When the command is a '*get*' command, data is queried from the HapticMASTER. If the command execution succeeds, the answer for this *get* request will be found in the *response* string. It is the **programmer's task** to copy this string to another string and / or process the data residing there (see the function *TimerCB()* for an example of this case). If the command execution fails, the *response* string will start with the text "--- ERROR:", trailed with the reason for the failure. E.G: "--- ERROR: state failed to set, incorrect value force""--- ERROR: State NOT set to 'force'. Initialize first."

TimerCB()

The *TimerCB()* function (CB stands for Call Back) is called each time an OpenGL timer gives a tick. We set a timer in OpenGL to tick every 10 milliseconds and connect the timer to this function. This function is then used for retrieving the latest End Effector position from the HapticMASTER. This is done by sending the HapticMASTER the command "get measpos". If the execution of the command fails (response contains "--- ERROR:"), an error message is printed with the corresponding reason. If the command execution succeeds, a string in the format *[float,float,float]* is returned in the *response* variable. This is a string representing a vector with three float values, namely, the x, y and z coordinates of the End Effector. The *ParseFloatVec()* function is a utility function (declared in *HapticMaster.h*) that helps the programmer parse (process) a string in the format *[float,float,float]* and copy the values into 3 float variables. The first parameter is the string to be parsed, and the 3 parameters coming afterwards are the 3 float variables where the x, y and z values should be written.

The new x, y and z values of this position are stored in the *CurrentPosition* variable (an array of 3 floats). These values are then used by the *DrawEndEffector()* function to draw the graphic equivalent of the End Effector (a small orange ball) on the computer monitor.

The *glutTimerFunc()* function sets the next time the *TimerCB()* function will be called.

main() function

Last, but not least, the *main()* function. The code in this function creates the HapticMASTER device handle by calling the *haDeviceOpen()* function with the IP address as a C-String. This function returns an identifier to the HapticMASTER device, which, from this point on, is used to reference the HapticMASTER device. If opening the device fails, the value HARET_ERROR (defined in the file *HapticAPI.h*) is returned. In this case, an error message is printed and the program terminates.
If the device was successfully opened, the *initializeDevice()* function is called. This function is declared in *HapticMaster.h* (thus a utility function) and initializes the HapticMASTER. You can choose to use this function 'as is', or write the code yourself. Either way, if the HapticMASTER encoders are not yet calibrated (E.G: directly after turning on the HapticMASTER), **you must initialize the HapticMASTER before you can use it.** See section Appendix A for more details about the *InitializeDevice()* function.

Example "02 – Hello Graphic World" creates a haptic block, by calling *haSendCommand()* with the "create block myBlock" command. *block* is the type of the object that we are creating, and *myBlock* is its name. The *haSendCommand()* call is surrounded by an IF statement. If sending the command fails, an error message is generated and the program terminates. If sending it succeeds, we print the response from the HapticMASTER and check if execution of the command succeeded. Should the response contain the string "--

- ERROR:", we terminate the program. Execution of the command was successful? The block haptic effect with the name *myBlock* can be addressed from now on and we go further with the else part of the IF statement.

There, we set some parameters for the just created block *myBlock*:

- *"set myBlock pos"* sets the **position** of the block. Notice that every time you supply a vector in the command syntax, it should be surrounded by the '[' and ']' characters. In this specific function call, we supply the position vector as 3 separate floats. The function haSendCommand() automatically assembles the command into its complete string form. The resulted command is then "set myBlock pos [0.0,0.0,0.0]".
- *"set myBlock size"* sets the **size** of the block using the 3 floats supplied as parametrs. The size parameter is a vector as well – x, y, z (dimensions in meters).
- *"set myBlock stiffness"* sets the **stiffness** of the block using the blockStiffness value – in other words how hard is the haptic block.
- *"set myBlock enable"* **enables** the haptic effect. If the effect is not enabled, it is not rendered in the real-time software, which means one cannot feel this object in the virtual haptic world.

Ok, what follows are some calls to setup the OpenGL graphics engine, create a window and pass the function names of the callback functions to OpenGL. Then the OpenGL main loop is started.

While running this example, you can control the End Effector, which is represented on the screen by a small orange ball, with the ball mounted on the HapticMASTER's force sensor. The Block object is represented by a light-green block shaped figure on the screen. Now, use the HapticMASTER to move the End Effector around and actually **touch** the block. What you will **feel**, is that the block is rather hard (its stiffness is set to 20,000). You can touch such an object only from the outside. This example also displays the workspace of the HapticMASTER. You will notice you can easily "walk" along the edges of the workspace since these are the physical limitations of the specific HapticMASTER you use. The workspace can be different for each specific HapticMASTER. More information about the HapticMASTER's workspace is found in the next chapter.

Well, this concludes our explanation of the second example. This time we added a graphical variant of the haptic world we built within the HapticMASTER. By reading the object dimensions from variables, we can make sure the haptic world is correlated with the graphic world.

5 HapticMASTER Workspace

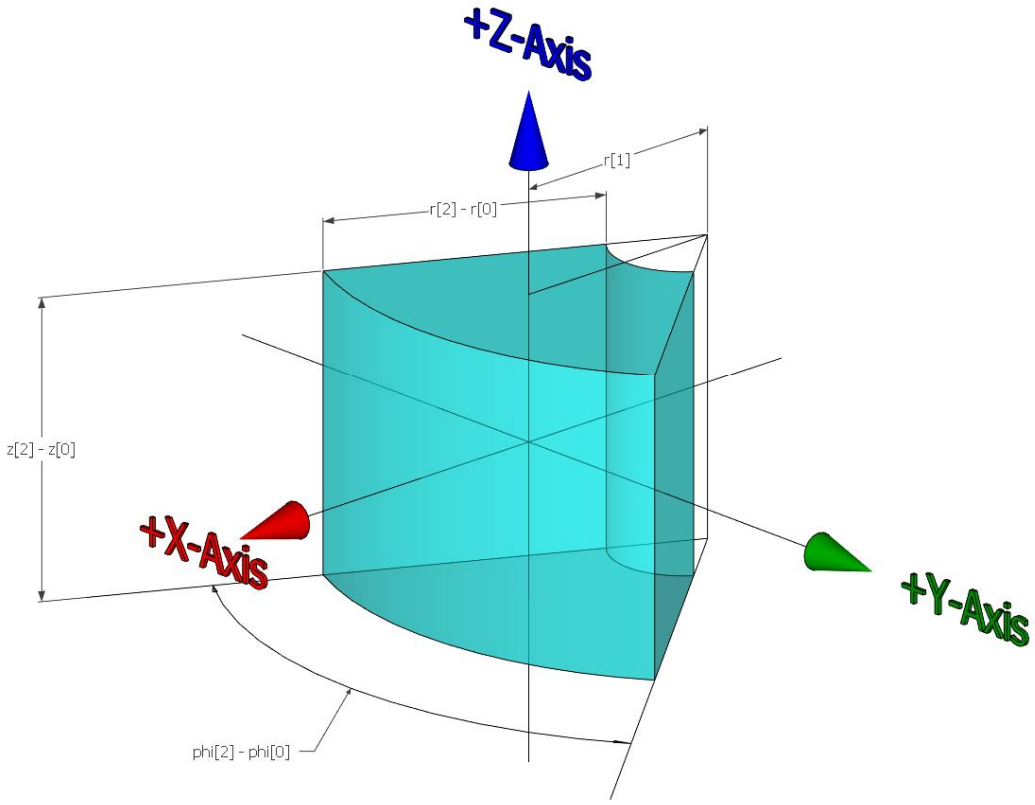


Figure 5-1 : HapticMASTER Workspace

In this chapter, we will look at the maximum movements a HapticMASTER can make, also called the workspace. The HapticMASTER has 3 degrees of freedom (DOFs): The R (radius), Phi (rotation angle) and Z (height) axes – also known as the ***Joint axes system***. These coordinates are continuously translated into the ***Cartesian axes system*** – consisting of the X, Y and Z axes. In this frame the positive X-Axis points towards the user, the positive Y-Axis points right (counter-clockwise rotation when looking from above) and the positive Z-Axis points upwards. Take a look at Figure 5-1 : HapticMASTER Workspace. The workspace is of course limited in all three directions. Example “02 Hello Graphic World” already displays the HapticMASTER’s workspace in the OpenGL window. The yellow lines and arcs sketch this workspace. When you move the HapticMASTER to its extreme positions, you will notice it stops moving there in this particular axis. At this point, you will find the End Effector exactly on the yellow line representing the edge of the workspace.
It is important that you have some understanding of the workspace because it sets the boundaries of the virtual haptic world you want the HapticMASTER to render. Feel free to explore the workspace by moving the HapticMASTER to its

extreme positions. Notice that the forces working on the End Effector are there even if the End Effector reached the workspace boundary. If you push the HapticMASTER to the right for example, and you let it proceed with its own inertia in this direction (without touching it anymore), the End Effector keeps moving to the right. When the End Effector touches the right edge, the inertia moving the End Effector to the right is still there. You will notice the End Effector moves now also in the X axis, in attempt to move in the Y axis as well. This will go on until the End Effector reaches its X boundary as well – where it will stop in this corner, waiting for new force inputs.

You might find the workspace wire frame and the colored X, Y and Z axes helpful during the design phase of your haptic world. So use them and when done, simply out-comment the *DrawWorkspace()* and *DrawAxes()* calls.

In the next chapter a rather important issue of the HapticMASTER is described. The different states the HapticMASTER goes through from power-up to running the haptic model.

6 HapticMASTER States

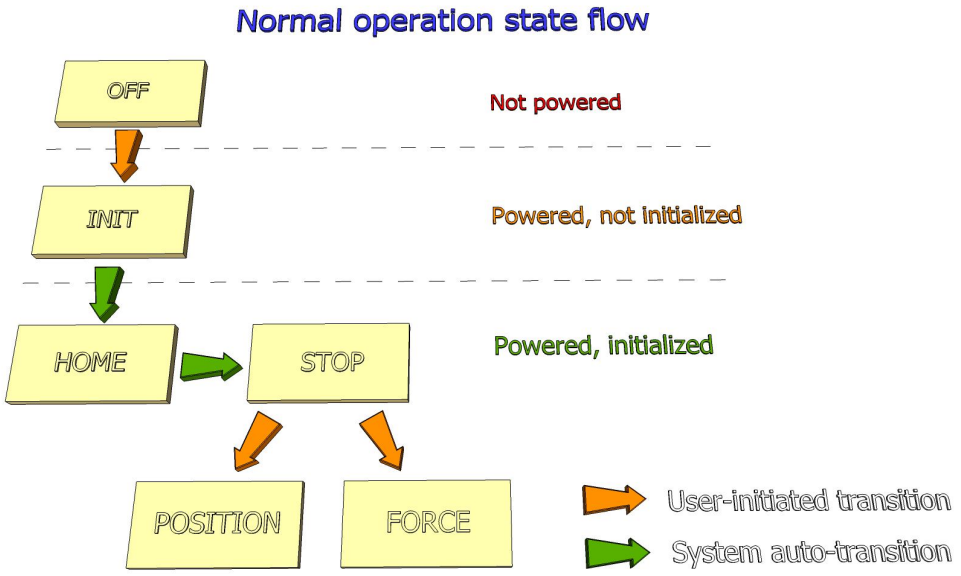


Figure 6-1 : HapticMASTER normal operation state flow

During power-up, the HapticMASTER will go through several states even before being able to run any haptic model whatsoever. As you can see in Figure 6-1 : HapticMASTER normal operation state flow, there are six different states marked yellow in the figure. When turned on, the HapticMASTER assumes the state **OFF**. This state is unpowered (motors are off). On older systems do not forget to press the green button on the HapticMASTER's electronic box after switching it on. On the new system, this switch is in the webinterface under safety, start. If you don't do this, the relays are not powered, and the HapticMASTER will not be able to make a transition to any other state. The user can then initiate a state transition to the state **INIT** (marked orange in the figure). This is done by sending the command "set state init". The HapticMASTER's motors are then powered and a search of the end stops in all three DOFs starts. Once the search is completed, the system auto-transits to the state **HOME**. The state **HOME** moves the end-effector gently to the defined home position. This position is defined in the Persistence file, that is saved in the Real-time computer. One can change this position through the HapticMASTER's Web Interface. When the end effector has reached the home position, another auto-transition takes place – namely the transition to the state **STOP**. In the state **STOP**, the HapticMASTER holds its position firmly. Effects that are present in the virtual world and force sensor inputs **DO NOT** change the End Effector's position.
From this state, one can initiate a transit to either the **POSITION** state (sending a "set state position" command) or the **FORCE** state (sending a "set state force" command). In the **POSITION** state, the forces rendered by the Effects and Objects in the virtual world influence the movement of the End Effector but the force sensor is not functional. The user cannot influence the End Effector position

by supplying force inputs through the force sensor. In the **FORCE** state, however, both Effects, Objects and the force sensor influence the End Effector's position. The user can then move the End Effector by applying force on the force sensor and rendered effects and objects can be felt as well. If the force sensor is not calibrated, the transition to state **FORCE** will fail.

State transition limitations

At any point, the user can **initiate** a state transition to any of the 6 states described above. However, the state transition is not always accepted by the HapticMASTER. If this is the case, you receive an error message from the HapticMASTER's API, starting with the text "--- ERROR:".

Herewith two examples of "illegal" state transitions that will end up by staying in the same state and receiving an error response:

1) When the HapticMASTER was just turned on (and the safety switch is on), the only state transition that will be accepted is to the state **INIT**.

   If you try to transit to any of the powered states (**HOME** / **STOP** / **POSITION** / **FORCE**), before INITializing, you will receive an error response from the HapticMASTER, demanding you to INITialize first.

2) Transition to the state FORCE when the force sensor is not calibrated or not responsive will end up with the HapticMASTER staying in the same state and an error response will be returned.

You can check the status of these two limitations before trying to transit to a specific state. Use the command "get position_calibrated" to check whether the HapticMASTER has successfully completed the initialization. You will receive the message "true" or "false".
To check whether the force sensor is calibrated, send the command "get force_calibrated", which will respond with "true" or "false" strings.

To sum up the HapticMASTER's possible states see the following sections.

The OFF state

Motors: Off
Force Sensor: Readable but not responsive
Effects: Non Active
Command: “set state off”
Result: HapticMASTER enters state OFF.
Description: Robot arm can be moved without any control loop involved (with full mechanical friction).

PAY ATTENTION: Do not move the robot arm by fetching it by the Force Sensor

The INIT state

Motors: On
Force Sensor: Readable but not responsive
Effects: Non Active
Command: “set state init”
Result: HapticMASTER enters state INIT and then automatically transits to state HOME.
Description: The HapticMASTER searches for its end stops (mechanical boundries) in all three degrees of freedom.
Consequently, a force sensor calibration is performed.

PAY ATTENTION: When commanding this state, don’t touch the Robot arm or the force sensor in any way that disrupts the normal initialization process.
If you touch the force sensor while calibrating it, the 0 readouts will be incorrect.

The HOME state

Motors: On
Force Sensor: Readable but not responsive
Effects: Non Active
Command: “set state home” or automatically when INIT finishes.
Result: HapticMASTER enters state HOME. When home position is reached, automatically transits to state STOP.
Description: Robot arm gently moves to the defined home position.

PAY ATTENTION: Do not interfere with the Robot arm’s movement. In this state, the force sensor is not responsive.

The STOP state

Motors: On
Force Sensor: Readable but not responsive
Effects: Non Active
Command: “set state stop” or automatically when home position is reached.
Result: HapticMASTER enters state STOP.

Description: The robot arm firmly keeps it current position in all three DOFs.

PAY ATTENTION: Do not apply too much force on the force sensor in this state. If the maximum allowed force is reached, the HapticMASTER will automatically transit to state OFF.

The POSITION state

Motors:	On
Force Sensor:	Readable but not responsive
Effects:	Active
Command:	“set state position”
Result:	HapticMASTER enters state POSITION.
Description:	The robot arm moves around if there are any effects or objects that influence the End Effector’s position. Force sensor is not responsive. This state should not be used for interactive applications. It simply deterministically follows the forces coming from the effects and objects that are created in the scene.

PAY ATTENTION: Do not apply too much force on the force sensor in this state. If the maximum allowed force is reached, the HapticMASTER will automatically transit to state OFF.
This state should not be used for interactive applications. It simply deterministically follows the forces coming from the effects and objects that are created in the scene.

The FORCE state

Motors:	On
Force Sensor:	Readable and responsive
Effects:	Active
Command:	“set state force”
Result:	HapticMASTER enters state FORCE.
Description:	The robot arm moves around if there are any effects or objects that influence the End Effector’s position and if the force sensor is touched.

State transitions and Error handling

If anything goes wrong during the HOME, STOP, FORCE, or POSITION states (e.g. an error occurs in the encoder or tacho electronics, too much force applied on the force sensor), the HapticMASTER enters the OFF state.
The user can initiate a transition to any of the six states by sending the corresponding “set state” command. The HOME, STOP, FORCE and POSITION states check first if a transition to this state is allowed, depending on the calibration status (position and force).

Further on in this chapter you will also find an example that lets you step through all the different states so you can experience the difference between them.

States summed up

State	Motors intact?	Effects rendered?	Force Sensor Responsive?	Next state
OFF	-	-	-	-
INIT	+	-	-	HOME
HOME	+	-	-	STOP
STOP	+	-	-	-
POSITION	+	+	-	-
FORCE	+	+	+	-

Example 3 - Source Code

Example 3 : HapticMASTER states

```
//---------------------------------------------------------------------
//        0 3   -   H A P T I C   M A S T E R   S T A T E S
//
// This example demonstrates how to make a transition between the
// different states of the HapticMASTER.
// This example uses the OpenGL's Display() function to periodically
// query the position of the end effector and the current state.
// In the previous example a timer callback function was used to
// achieve this.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double CurrentPosition[3];

// Variable that holds the number of states that are available
const int  NrStates = 6;

// Displayed text for the states
char State[NrStates][25] = {"Off (o)", "Init (i)", "Stop (s)",
                            "Home (h)", "Force (f)", "Position (p)"};
// State names as being recognized by HapticMASTER
char stateNames[NrStates][25] = {"off;","init;","stop;",
                            "home;", "force;", "position;"};

unsigned int currentState = 0;
char tempCurrentStateStr[30] = "";

// Some Viewport dimension varaibles and initialization
int VpWidth = 160;
int VpHeight = 60;
int ViewportXPositions[NrStates] = {500, 500, 500, 500, 600, 420};
int ViewportYPositions[NrStates] = {525 ,450, 375, 300, 225, 225};

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Block OpenGL Material Parameters.
GLfloat BlockAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat BlockDiffuse[] = {0.00, 0.80, 0.67, 1.00};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

GLfloat SpecularOff[] = {0.00, 0.00, 0.00, 0.00};
GLfloat EmissiveOff[] = {0.50, 0.50, 0.50, 0.00};
```

```
GLfloat ShininessOff = {0.00};

//---------------------------------------------------------------------
//                O B J E C T   P A R A M E T E R S
//---------------------------------------------------------------------
double blockSize[3] = {0.15,0.15,0.15};
double blockPos[3] = {0.0,0.0,0.0};
double springPos[3] = {0.2,0.0,-0.1};
double blockStiffness = 20000;

//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                   B L O C K   M A T E R I A L
//
// BlockMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The Block.
//---------------------------------------------------------------------
void BlockMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlockAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlockDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, SpecularOff);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, ShininessOff);
}

//---------------------------------------------------------------------
//                       D R A W   B L O C K
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Haptic Block Object In OpenGl
//---------------------------------------------------------------------
void DrawBlock(void)
{
   BlockMaterial();
   glTranslatef(blockPos[PosX], blockPos[PosY], blockPos[PosZ]);
   glScalef(blockSize[0], blockSize[1], blockSize[2]);
   glutSolidCube(1.0);
}

//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                        I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{
   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_NORMALIZE);

   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//---------------------------------------------------------------------
//                       D R A W   H M   S T A T E S
//
// Draw the HapticMASTER's state buttons.
//---------------------------------------------------------------------
void DrawHmStates(void)
{
   int i=0;
   int j=0;

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   gluOrtho2D(0.0, double(VpWidth), 0.0, double(VpHeight));

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glDisable(GL_BLEND);
   for(i=0; i<NrStates; i++)
   {
      j=0;
      glPushMatrix();
      glViewport(ViewportXPositions[i], ViewportYPositions[i], VpWidth, VpHeight);

      if(i == currentState)
         EndEffectorMaterial();
      else
         BlockMaterial();

      glBegin(GL_LINE_LOOP);
         glVertex2d(0.0,  0.0);
         glVertex2d(0.0,  VpHeight);
         glVertex2d(VpWidth,  VpHeight);
         glVertex2d(VpWidth,  0.0);
      glEnd();

      glRasterPos2f(5, VpHeight/2.0);
      while (State[i][j] != '\0')
      {
```

```
         glutBitmapCharacter(GLUT_BITMAP_8_BY_13, State[i][j]);
         j++;
      }
      glPopMatrix();

      glutPostRedisplay();
   }
   glEnable(GL_BLEND);
}

//-----------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
// The position of the EndEffector and the current state are queried
// from the HapticMASTER just before the data is displayed.
//-----------------------------------------------------------------------
void Display (void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   glViewport(0, 300, 400, 300);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   gluPerspective (30.0, 1.3333, 0.05, 20.0);
   gluLookAt (1.0, 0.5, 0.35, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector position from the HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if (strstr(response, "--- ERROR:") ) {
      printf("get modelpos ==> %s", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
                     CurrentPosition[PosZ] );
   }
   DrawEndEffector();
   DrawBlock();

   // Get the current state of the HapticMASTER
   haSendCommand(dev, "get state", response);
   if (strstr(response, "--- ERROR:") ) {
      printf("get state ==> %s\n", response);
      getchar();
      exit(-1);
   }
   bool stateFound = false;
   unsigned int index = 0;
   while (!stateFound) {
      if ( strcmp(response, stateNames[index]) ) {
         index++;
      }
      else {
         stateFound = true;
      }
   }
   currentState = index;
   DrawHmStates();

   glPopMatrix ();
   glutSwapBuffers();
}
```

```
//---------------------------------------------------------------------
//                          R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//---------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
    glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    float fAspect = (float)iWidth/iHeight;
    gluPerspective (30.0, fAspect, 0.05, 20.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
    switch (ucKey)
   {
      case 'o' :
         haSendCommand(dev, "set state off", response);
         printf("set state off ==> %s\n", response);
         break;
      case 'i' :
         haSendCommand(dev, "set state init", response);
         printf("set state init ==> %s\n", response);
         break;
      case 's' :
         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);
         break;
      case 'h' :
         haSendCommand(dev, "set state home", response);
         printf("set state home ==> %s\n", response);
         break;
      case 'f' :
         haSendCommand(dev, "set state force", response);
       printf("set state force == > %s\n", response);
       haSendCommand(dev, "set mySpring enable;set myBlock enable", response);
       printf("set mySpring enable;set myBlock enable == > %s\n", response);
         break;
      case 'p' :
         haSendCommand(dev, "set state position", response);
         printf("set state position ==> %s\n", response);
       haSendCommand(dev, "set mySpring enable;set myBlock enable", response);
       printf("set mySpring enable;set myBlock enable == > %s\n", response);
         break;
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);
         exit(0);
         break;
    }
}
```

```
//-----------------------------------------------------------------------
//                            M A I N
//
// 03-Haptic-MASTER-States Main Function
//-----------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Open the HapticMASTER device
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      // Create the Haptic Block Effect and supply it with parameters
      if ( haSendCommand(dev, "create block myBlock", response) ) {
         printf("could not send command create block myBlock\n");
         getchar();
         exit(-1);
      }

      printf("create block myBlock ==> %s\n", response);

      if (strstr(response, "--- ERROR:") ) {
         printf("create block myBlock ==> %s\n", response);
         getchar();
         exit(-1);
      }

      else {
         haSendCommand(dev, "set myBlock pos", blockPos[PosX], blockPos[PosY],
                       blockPos[PosZ], response);
         printf("set myBlock pos [%g,%g,%g] ==> %s\n", blockPos[PosX], blockPos[PosY],
                blockPos[PosZ], response);

         haSendCommand(dev, "set myBlock size", blockSize[0], blockSize[1], blockSize[2],
                       response);
         printf("set myBlock size [%g,%g,%g] ==> %s\n", blockSize[0], blockSize[1],
                blockSize[2], response);

         haSendCommand(dev, "set myBlock stiffness", blockStiffness, response);
         printf("set myBlock stiffness %g ==> %s\n", blockStiffness, response);

         haSendCommand(dev, "set myBlock enable", response);
         printf("set myBlock enable ==> %s\n", response);
      }

      // Create the Haptic spring Effect and supply it with parameters
      if ( haSendCommand(dev, "create spring mySpring", response) ) {
         printf("could not send command create spring mySpring\n");
         getchar();
         exit(-1);
      }

      printf("create spring mySpring ==> %s\n", response);

      if (strstr(response, "--- ERROR:") ) {
         printf("create spring mySpring ==> %s\n", response);
         getchar();
         exit(-1);
      }

      else {
         haSendCommand(dev, "set mySpring pos", springPos[PosX], springPos[PosY],
                       springPos[PosZ], response);
         printf("set mySpring pos [%g,%g,%g] ==> %s\n", springPos[PosX], springPos[PosY],
                springPos[PosZ], response);
```

```
         haSendCommand(dev, "set mySpring enable", response);
         printf("set mySpring enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example03: HapticMASTER States");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 3 - Description

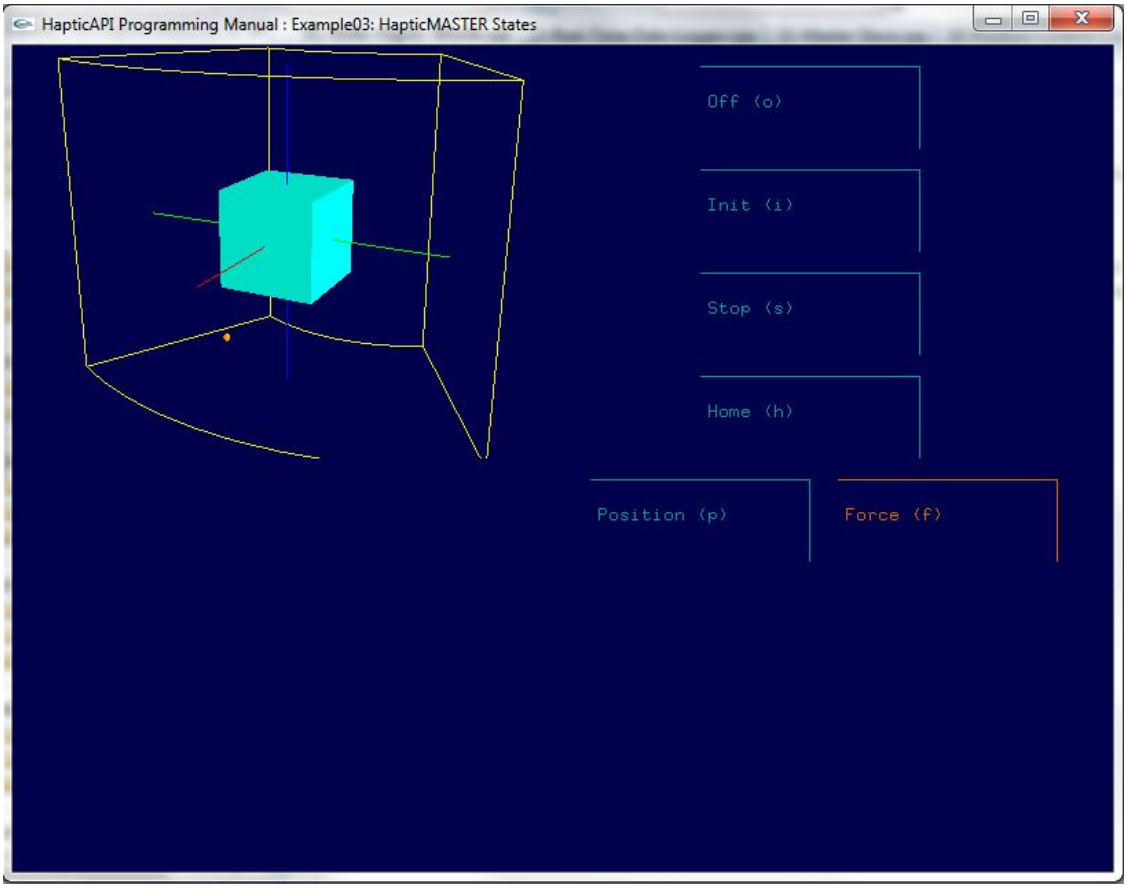


Figure 6-2 : Example 3 Screen Shot

The virtual haptic world created in Example 3 : HapticMASTER states", consists of one block object only. The same block as we've seen in the previous example. Here we add one spring effect that will demonstrate the difference between the FORCE state and the POSITION state. In this example the OpenGL window will be logically split in two halves. The left side of the window will be used to display the HapticMASTER workspace with the block inside. On the right side a simple diagram is shown that displays all the possible HapticMASTER states in blue. An exception to this is the current state, which will be shown in orange. This way it is always clear in what state the HapticMASTER resides. The characters between the brackets after each state name are the keyboard keys you can press to make a transition to that particular state.

We suggest you play around with the different states to thoroughly understand the correct functionality. You can switch to the **OFF** state and see that the HapticMASTER becomes unpowered – the robot arm falls downwards. Then switch to the **INIT** state and notice how the position of the End Effector is updated in three stages: In the beginning all three axes have an unknown

position, then each time an axis position becomes known, the position in this axis is shown correctly on the screen since it is also known for the HapticMASTER. Finally, when the initialization process is finished, the position of the End Effector is correct and the HapticMASTER automatically enters the **HOME** state. The End Effector gently moves to the defined home position and when it reaches this position, the state transits automatically to **STOP**. You can then switch to the **FORCE** state. Move the End Effector around with force sensor inputs and the spring will always pull the End Effector to the spring's position. Finally, while moving the End Effector using the force sensor, you can switch to **POSITION** state. From the moment you are in this state, you can not control the HapticMASTER via the force sensor but you can see that the End Effector still moves towards the spring's position, until it reaches this position and rests there. It is simply not "listening" to the force sensor. When you return to the **FORCE** state, you can again control the HapticMASTER via the force sensor.

Ok, let us take a look now at the source code.

Includes, defines, global variables

At the beginning of the source code listing you notice the (maybe already familiar) HapticAPI and HapticMASTER include statements. The define statements for the IPADDRESS, PosX, PosY and PosZ should also be familiar already. Variables *dev*, *response* and *CurrentPosition* are the same as in the previous example.

Next are some variables that hold the number of states (*int NrStates = 6*), state name for the buttons (*char State[NrStates][25])* and state names as known by the HapticMASTER (*char stateNames[NrStates][25]).*
The variable *currentState* is an integer that holds the index of the current state in these two arrays. As you can see in the array *stateNames*, the mapping of the indices per state is as follows:

State 0 is "OFF",
State 1 is "INIT",
State 2 is "STOP",
State 3 is "HOME",
State 4 is "FORCE",
And state 5 is "POSITION".

This index will be used to identify the current state in loops that will be used in the program (for drawing buttons for example)… The variables *VpWidth* and *VpHeight* define the graphical size of the buttons displaying the possible states. The two arrays *ViewportXPosition[]* and *ViewportYPosition[]* define the X and Y coordinates of each state button respectively.

The functions *EndEffectorMaterial(), BlockMaterial(), DrawBlock(), DrawEndEffector()* and *InitOpenGL()* are all OpenGL functions and are the same as in the previous example.

DrawHmStates() function

DrawHmStates() draws the six state buttons (the right side of the screen). The loop iterates through all states. If the state in the loop is the current state, EndEffectorMaterial() is called – resulting in changing the color to orange. If the state in the loop is not the current state, BlockMaterial() is called, resulting in changing the color to blue. Further, the button and the text are drawn according to the coordinates in the arrays mentioned above.

Display() function

The *void Display(void)* function, as always, is called by OpenGL whenever the screen must be redrawn. In the previous example, we used the *TimerCB()* function to update positions within a specific frequency. The *Display()* function is repeatedly called by the OpenGL engine to update the OpenGL window. This time, instead of using a *TimerCB()* function, we simply use the display() function to update the HapticMASTER data in our program. In this function we'll constantly read the **current position** of the End Effector and the **current state**.

First, the viewport, perspective and position of the OpenGL camera are determined (*glViewport()*, *gluPerspective()* and *gluLookAt()* function calls).

Then, the axes and workspace are drawn (*DrawAxes()* and *DrawWorksapce()*).

Then, the function *haSendCommand()* is called with the command "get modelpos" and consequently calling *ParseFloatVec()* updates the position of the End Effector. At that moment we can draw the End Effector and the block by calling the functions *DrawEndEffector()* and *DrawBlock()*.

Finally, the function *haSendCommand()* with the command "get state" updates the current state. In the *response* variable the name of the state will be returned (with leading and trailing quote characters). Consequently, a *while* loop searches for the index of this name in the array *stateNames[]*. When the index is found, this number is copied to the *currentState* variable, holding the current state number.
Now the state buttons can be drawn by calling the function *DrawHmStates()*.

Keyboard() function

This function takes actions when a key is pressed by the user. In our example this function is used to make a transition to another state. Pressing the '**o**' key transits to the **OFF** state. This is done by sending the HapticMASTER a "set state off" command using the haSendCommand() function.
The same technique is used for the other states:

'o' - State OFF - "set state off"
'i' - State INIT - "set state init"
's' - State STOP - "set state stop"
'h' - State HOME - "set state home"
'f' - State FORCE - "set state force"
'p' - State POSITION - "set state position"

When the escape key is pushed, all haptic objects are removed by sending a "remove all" command and the HapticMASTER's state is changed to **STOP** by sending the command "set state stop".

main() function

The *main()* function opens the HapticMASTER device (with the given IP address) and returns an identifier to the device (variable *dev*).
The device is then initialized (*InitializeDevice()* function). The haptic block is created exactly the same as in the previous example:

"create block myBlock"
"set myBlock pos [0,0,0]"
"set myBlock size [0.15,0.15,0.15]"
"set myBlock stiffness 20000.0"
"set myBlock enable"

And the spring is created as well:

"create spring mySpring"
"set mySpring pos [0.2,0.0,-0.1]"
"set mySpring enable"

The rest of the code is OpenGL functions registration. Notice that we don't use the *TimerCB()* function anymore and therefore we don't need to register it as well. What we used to do in *TimerCB()* is now done in the *Display()* function.

7 More Haptic Objects

In the examples shown so far we have created only two haptic objects: the block and the sphere objects. We have shown you not only how to create it but also how to change the haptic properties – position, size, spring stiffness, damping factor etc.

However, the HapticAPI provides more objects than just these two objects. In this chapter we will provide an example which shows the creation of three haptic objects of different shapes, which are positioned on top of a plane. We will also demonstrate how to rotate objects using quaternions.
Four block objects are created, one of which is used for the plane. This way you can see a different use of a **block** object through scaling. The other 3 blocks will demonstrate how to change the object's orientation. Next to this we introduce the creation of a **sphere**, and the creation of a **torus** (or donut). The **sphere** and the **torus** are interesting shapes when it comes to touching them, they feel really smooth. The last objects that exists but are not demonstrated in this example are the **cylinder** and **capsule**.

As usual we will provide the source code first. You will find it on the coming pages. Again there is a lot of OpenGL code to get the virtual world not only in the HapticMASTER but also on the computer monitor. Now that we have a total of six haptic objects of different shapes, we also need more code to draw them with the corresponding material on the correct positions on the monitor.

Following the source code you will find a thorough explanation of the code.

Example 4 - Source Code

Example 4 : More Haptic Objects

```
//---------------------------------------------------------------------
//         0 4   -   M O R E   H A P T I C   O B J E C T S
//
// This exmaple demonstrates how to create multiple haptic objects
// in the haptic world.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double CurrentPosition[3];

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Floor Object OpenGL Material Parameters
GLfloat FloorAmbient[] = {1.00, 1.00, 0.00, 1.00};
GLfloat FloorDiffuse[] = {1.00, 1.00, 0.00, 1.00};

// General OpenGL Material Parameters
GLfloat ObjectAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat ObjectDiffuse[] = {0.00, 0.80, 0.67, 1.00};

GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

GLfloat SpecularOff[] = {0.00, 0.00, 0.00, 0.00};
GLfloat EmissiveOff[] = {0.50, 0.50, 0.50, 0.00};
GLfloat ShininessOff = {0.00};

//---------------------------------------------------------------------
// O B J E C T S   P A R A M E T E R S
//---------------------------------------------------------------------
// Location And Size Parameters For The Haptic Floor Plane Object
double FloorCenter[3] = {0.0, 0.0, -0.01};
double FloorSize[3] = {0.35, 0.35, 0.01};
double FloorStiffness = 20000.0;

// Location And Size Parameters For The Haptic Cube Object
Vector3d *Cube1Center = new Vector3d(0.1, -0.1, 0.02);
Vector3d *Cube1Size = new Vector3d(0.06, 0.06, 0.06);
Vector3d *Cube1Orientation = new Vector3d(0.0, 0.0, 0.0);
double Cube1Attitude[4] = {0.0, 0.0, 0.0, 1.0};
double Cube1Stiffness = 20000.0;

Vector3d *Cube2Center = new Vector3d(0.1, 0.0, 0.02);
Vector3d *Cube2Size = new Vector3d(0.06, 0.06, 0.06);
Vector3d *Cube2Orientation = new Vector3d(0.0, 0.0, 0.0);
```

```
double Cube2Attitude[4] = {0.0, 0.382, 0.0, 0.923};
double Cube2Stiffness = 20000.0;

Vector3d *Cube3Center = new Vector3d(0.1, 0.1, 0.02);
Vector3d *Cube3Size = new Vector3d(0.06, 0.06, 0.06);
Vector3d *Cube3Orientation = new Vector3d(0.0, 0.0, 0.0);
double Cube3Attitude[4] = {0.382, 0.0, 0.0, 0.923};
double Cube3Stiffness = 20000.0;

// Location And Size Parameters For The Haptic Sphere Object
double SphereCenter[3] = {-0.1, -0.1, 0.02};
double SphereRadius = 0.03;
double SphereStiffness = 20000.0;

// Location, Orientation And Size Parameters For The Haptic Torus Object
double TorusCenter[3] = {-0.1, 0.1, 0.08};
double TorusOrient[3] = {0.0, 1.57, 0.0};
double TorusRingRadius = 0.05;
double TorusTubeRadius = 0.02;
double TorusStiffness = 20000.0;


//--------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//--------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//--------------------------------------------------------------------
//                   F L O O R   M A T E R I A L
//
// FloorMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing The Floor Object.
//--------------------------------------------------------------------
void FloorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, FloorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, FloorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, SpecularOff);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, ShininessOff);
}

//--------------------------------------------------------------------
//                   O B J E C T   M A T E R I A L
//
// ObjectMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing Any Of The Objects.
//--------------------------------------------------------------------
void ObjectMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, ObjectAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, ObjectDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}
```

```
//---------------------------------------------------------------------
//                       D R A W   F L O O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Floor In OpenGl.
//---------------------------------------------------------------------
void DrawFloor(void)
{
   FloorMaterial();
   glTranslatef(FloorCenter[0], FloorCenter[1], FloorCenter[2]);
   glScalef(FloorSize[0], FloorSize[1], FloorSize[2]);
   glutSolidCube(1.0);
}

//---------------------------------------------------------------------
//                      D R A W   C U B E   1
//
// This Function Is Called To Draw The Graphic Equivalent Of
// Cube1 In OpenGl.
//---------------------------------------------------------------------
void DrawCube1(void)
{
   ObjectMaterial();
   glPushMatrix();
   glTranslatef(Cube1Center->x, Cube1Center->y, Cube1Center->z);
   glRotatef(0, 0.0, 0.0, 0.0);
   glutSolidCube(Cube1Size->x);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                      D R A W   C U B E   2
//
// This Function Is Called To Draw The Graphic Equivalent Of
// Cube2 In OpenGl.
//---------------------------------------------------------------------
void DrawCube2(void)
{
   ObjectMaterial();
   glPushMatrix();
   glTranslatef(Cube2Center->x, Cube2Center->y, Cube2Center->z);
   glRotatef(45, 0.0, 1.0, 0.0);
   glutSolidCube(Cube2Size->x);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                      D R A W   C U B E   3
//
// This Function Is Called To Draw The Graphic Equivalent Of
// Cube1 In OpenGl.
//---------------------------------------------------------------------
void DrawCube3(void)
{
   ObjectMaterial();
   glPushMatrix();
   glTranslatef(Cube3Center->x, Cube3Center->y, Cube3Center->z);
   glRotatef(45, 1.0, 0.0, 0.0);
   glutSolidCube(Cube3Size->x);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                      D R A W   S P H E R E
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Sphere In OpenGl.
//---------------------------------------------------------------------
void DrawSphere(void)
{
   ObjectMaterial();
   glPushMatrix();
   glTranslatef(SphereCenter[0], SphereCenter[1], SphereCenter[2]);
   glutSolidSphere(SphereRadius, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                      D R A W   T O R U S
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Torus In OpenGl.
//---------------------------------------------------------------------
void DrawTorus(void)
{
   ObjectMaterial();
   glPushMatrix();
   glTranslatef(TorusCenter[0], TorusCenter[1], TorusCenter[2]);
//   glRotatef(90, 0.0, 0.0, 0.0);
   glutSolidTorus(TorusTubeRadius, TorusRingRadius, 60, 60);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                       I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{

   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
```

```
   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//----------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//----------------------------------------------------------------------
void Display (void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   // define eyepoint in such a way that
   // drawing can be done as in lab-frame rather than sgi-frame
   // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   if (haSendCommand( dev, "get modelpos", response ) ) {
      printf("--- ERROR: Could not send command get modelpos\n");
      getchar();
      exit(-1);
   }

   if (strstr(response, "--- ERROR:") ) {
      printf("get modelpos ==> %s\n", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
CurrentPosition[PosZ] );
   }

   DrawEndEffector();

   DrawCube1();
   DrawCube2();
   DrawCube3();
   DrawSphere();
   DrawTorus();

   DrawFloor();

   glPopMatrix ();
   glutSwapBuffers();
}
```

```
//---------------------------------------------------------------------
//                          R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//---------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
   glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();

   float fAspect = (float)iWidth/iHeight;
   gluPerspective (30.0, fAspect, 0.05, 20.0);

   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
}

//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;
   }
}

//---------------------------------------------------------------------
//                            M A I N
//
// 04-More-Haptic-Objects Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      // Create the Haptic Block Effect as myFloor and supply it with parameters
      if ( haSendCommand(dev, "create block myFloor", response) ) {
         printf("--- ERROR: Could not send command create block myFloor\n");
         getchar();
         exit(-1);
      }

      printf("create block myFloor ==> %s\n", response);

      if ( strstr(response, "--- ERROR:") ) {
         getchar();
         exit(-1);
      }
      else {
```

```
   haSendCommand(dev, "set myFloor pos", FloorCenter[0], FloorCenter[1],
                 FloorCenter[2], response);
   printf("set myFloor pos [%g,%g,%g] ==> %s\n", FloorCenter[0], FloorCenter[1],
          FloorCenter[2], response);

   haSendCommand(dev, "set myFloor size", FloorSize[0], FloorSize[1], FloorSize[2],
                 response);
   printf("set myFloor size [%g,%g,%g] ==> %s\n", FloorSize[0], FloorSize[1],
          FloorSize[2], response);

   haSendCommand(dev, "set myFloor stiffness", FloorStiffness, response);
   printf("set myFloor stiffness %g ==> %s\n", FloorStiffness, response);

   haSendCommand(dev, "set myFloor enable", response);
   printf("set myFloor enable ==> %s\n", response);
}

// Create the Haptic Block Effect as myCube1 and supply it with parameters
if ( haSendCommand(dev, "create block myCube1", response) ) {
   printf("--- ERROR: Could not send command create block myCube1\n");
   getchar();
   exit(-1);
}

printf("create block myCube1 ==> %s\n", response);

if ( strstr(response, "--- ERROR:") ) {
   getchar();
   exit(-1);
}
else {
   haSendCommand(dev, "set myCube1 pos", Cube1Center->x, Cube1Center->y,
                 Cube1Center->z, response);
   printf("set myCube1 pos [%g,%g,%g] ==> %s\n", Cube1Center->x, Cube1Center->y,
          Cube1Center->z, response);

   haSendCommand(dev, "set myCube1 size", Cube1Size->x, Cube1Size->y, Cube1Size->z,
                 response);
   printf("set myCube1 size [%g,%g,%g] ==> %s\n", Cube1Size->x, Cube1Size->y,
          Cube1Size->z, response);

   haSendCommand(dev, "set myCube1 stiffness", Cube1Stiffness, response);
   printf("set myCube1 stiffness %g ==> %s\n", Cube1Stiffness, response);

   haSendCommand(dev, "set myCube1 enable", response);
   printf("set myCube1 enable ==> %s\n", response);
}

// Create the Haptic Block Effect as myCube2 and supply it with parameters
if ( haSendCommand(dev, "create block myCube2", response) ) {
   printf("--- ERROR: Could not send command create block myCube2\n");
   getchar();
   exit(-1);
}

printf("create block myCube2 ==> %s\n", response);

if ( strstr(response, "--- ERROR:") ) {
   getchar();
   exit(-1);
}
else {
   haSendCommand(dev, "set myCube2 pos", Cube2Center->x, Cube2Center->y,
                 Cube2Center->z, response);
   printf("set myCube2 pos [%g,%g,%g] ==> %s\n", Cube2Center->x, Cube2Center->y,
          Cube2Center->z, response);

   haSendCommand(dev, "set myCube2 size", Cube2Size->x, Cube2Size->y, Cube2Size->z,
                 response);
   printf("set myCube2 size [%g,%g,%g] ==> %s\n", Cube2Size->x, Cube2Size->y,
          Cube2Size->z, response);
```

```
   haSendCommand(dev, "set myCube2 stiffness", Cube2Stiffness, response);
   printf("set myCube2 stiffness %g ==> %s\n", Cube2Stiffness, response);

   haSendCommand(dev, "set myCube2 att", Cube2Attitude[0], Cube2Attitude[1],
                 Cube2Attitude[2], Cube2Attitude[3], response);
   printf("set myCube2 att [%g,%g,%g,%g] ==> %s\n", Cube2Attitude[0],
          Cube2Attitude[1], Cube2Attitude[2], Cube2Attitude[3], response);

   haSendCommand(dev, "set myCube2 enable", response);
   printf("set myCube2 enable ==> %s\n", response);
}

// Create the Haptic Block Effect as myCube3 and supply it with parameters
if ( haSendCommand(dev, "create block myCube3", response) ) {
   printf("--- ERROR: Could not send command create block myCube3\n");
   getchar();
   exit(-1);
}

printf("create block myCube3 ==> %s\n", response);

if ( strstr(response, "--- ERROR:") ) {
   getchar();
   exit(-1);
}
else {
   haSendCommand(dev, "set myCube3 pos", Cube3Center->x, Cube3Center->y,
                 Cube3Center->z, response);
   printf("set myCube3 pos [%g,%g,%g] ==> %s\n", Cube3Center->x, Cube3Center->y,
          Cube3Center->z, response);

   haSendCommand(dev, "set myCube3 size", Cube3Size->x, Cube3Size->y, Cube3Size->z,
                 response);
   printf("set myCube3 size [%g,%g,%g] ==> %s\n", Cube3Size->x, Cube3Size->y,
          Cube3Size->z, response);

   haSendCommand(dev, "set myCube3 stiffness", Cube3Stiffness, response);
   printf("set myCube3 stiffness %g ==> %s\n", Cube3Stiffness, response);

   haSendCommand(dev, "set myCube3 att", Cube3Attitude[0], Cube3Attitude[1],
                 Cube3Attitude[2], Cube3Attitude[3], response);
   printf("set myCube3 att [%g,%g,%g,%g] ==> %s\n", Cube3Attitude[0],
          Cube3Attitude[1], Cube3Attitude[2], Cube3Attitude[3], response);

   haSendCommand(dev, "set myCube3 enable", response);
   printf("set myCube3 enable ==> %s\n", response);
}

// Create the Haptic sphere Effect as mySphere and supply it with parameters
if ( haSendCommand(dev, "create sphere mySphere", response) ) {
   printf("--- ERROR: Could not send command create sphere mySphere");
   getchar();
   exit(-1);
}

printf("create sphere mySphere ==> %s\n", response);

if ( strstr(response, "--- ERROR:") ) {
   getchar();
   exit(-1);
}
else {
   haSendCommand(dev, "set mySphere pos", SphereCenter[0], SphereCenter[1],
                 SphereCenter[2], response);
   printf("set mySphere pos [%g,%g,%g] ==> %s\n", SphereCenter[0], SphereCenter[1],
          SphereCenter[2], response);

   haSendCommand(dev, "set mySphere radius", SphereRadius, response);
   printf("set mySphere radius %g ==> %s\n", SphereRadius, response);
```

```
         haSendCommand(dev, "set mySphere stiffness", SphereStiffness, response);
         printf("set mySphere stiffness %g ==> %s\n", SphereStiffness, response);

         haSendCommand(dev, "set mySphere enable", response);
         printf("set mySphere enable ==> %s\n", response);
      }

      // Create the Haptic torus Effect as myTorus and supply it with parameters
      if ( haSendCommand(dev, "create torus myTorus", response) ) {
         printf("--- ERROR: Could not send command create torus myTorus\n");
         getchar();
         exit(-1);
      }

      printf("create torus myTorus ==> %s\n", response);

      if ( strstr (response, "--- ERROR:") ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand(dev, "set myTorus pos", TorusCenter[0], TorusCenter[1],
                       TorusCenter[2], response);
         printf("set myTorus pos [%g,%g,%g] ==> %s\n", TorusCenter[0], TorusCenter[1],
                TorusCenter[2], response);

         haSendCommand(dev, "set myTorus ring_radius", TorusRingRadius, response);
         printf("set myTorus radius %g ==> %s\n", TorusRingRadius, response);

         haSendCommand(dev, "set myTorus tube_radius", TorusTubeRadius, response);
         printf("set myTorus tuberadius %g ==> %s\n", TorusTubeRadius, response);

         haSendCommand(dev, "set myTorus stiffness", TorusStiffness, response);
         printf("set myTorus stiffness %g ==> %s\n", TorusStiffness, response);

         haSendCommand(dev, "set myTorus enable", response);
         printf("set myTorus enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example04: More Haptic Objects");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 4 - Description

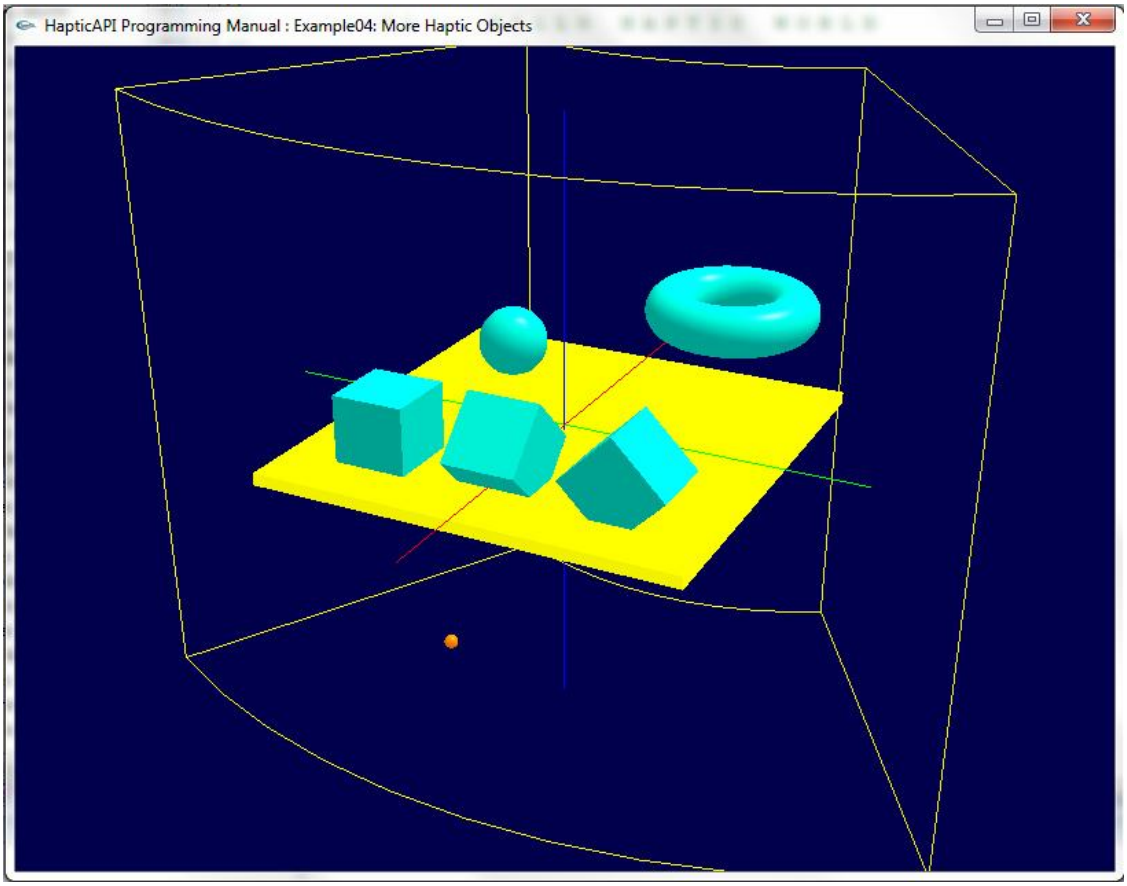


Figure 7-1 : Example 4 Screen Shot

When you run Example “04-More Haptic Effects” you will get something like the screenshot in the above figure. Now let’s take a closer look at the code and figure out what’s happening.

Includes, defines, global variables

All includes, defines and global variables are the same as in previous examples.

OpenGL materials and haptic objects

The following *GLfloat* and *double* arrays are used to define the OpenGL materials (End Effector, Floor, Object) and haptic objects that will be used in this example (Floor, Cube, Sphere, Torus).
You may have noticed that the Cube object’s center and size are defined in another way. We use the Vector3d class, which is defined *HapticAPI.h*.

To define the *CubePosition* variable we declare a pointer to an object of the class Vector3d, and create an instance of it with the values 0.1, -0.1, 0.03 for it's x, y, and z coordinates. The same technique is used for the *CubeSize* variable. We do this because we want to show you different ways of working with the *haSendCommand()* function. Please refer to Appendix A for further explanation.

Material functions and Draw functions

The following functions are used for changing the material settings:
EndEffectorMaterial() - orange,
FloorMaterial() - yellow,
ObjectMaterial() - blue.

The drawing functions for the Floor, Cube, Sphere, Torus and End Effector are as follows:
DrawFloor(), *DrawCube()*, *DrawSphere()*, and *DrawTorus()*.
Because the Cube position and size are defined using the Vector3d class, the function *DrawCube()* has to extract the information about the cube differently. Since *CubeCenter* is a pointer to a Vector3d object, we use the pointer notation to extract the x, y and z coordinates:
CubeCenter->x, *CubeCenter->y* and *CubeCenter->z* respectively.

Display() function

The same as in the previous example. It draws the Axes, the Workspace, then queries the HapticMASTER for the End Effector position ("get measpos" in combination with *ParseFloatVec()*). Then the End Effector is drawn and the Cube, Sphere, Torus and Floor.

The main() function

The main function takes care of opening the HapticMASTER device, initializing it, and creating the haptic Floor, Cube, Sphere and Torus.

Notice the technique we use to create the haptic objects:
The function *haSendCommand* can be used in 4 different configurations. Each configuration expects a different setting of the parameters:

1) **Constant command (no parameters):**

   haSendCommand (long int dev, char command, char* response)*

   This version expects 3 parameters. In the *command* parameter the **complete command** is supplied. Use this version of *haSendCommand()* when the specific command does not depend on any variable.

   E.G: *haSendCommand (dev, "set state force", response)* or
   haSendCommand (dev, "set mySpring enable", response)

2) **Command with one double value:**

   haSendCommand (long int dev,
   char command,*
   double input,
   char response)*

   This version expects 4 parameters. The *command* parameter is then **incomplete**. The value of the *input* parameter is first concatenated (appended) to the command, which makes the command complete. Use this version of *haSendCommand()* when the specific command depends on one scalar double variable.

   E.G: *haSendCommand (dev, "set myBlock stiffness",*
   blockStiffness, response)
   blockStifness is a double with the value 10000.
   The complete command being sent through the *haSendCommand* will be "set myBlock stiffness 10000"

3) Command with three separate double values

haSendCommand (long int dev, char command,*
double input1,
double input2,
double input3,
char response)*

This version expects 6 parameters. The *command* parameter is **incomplete**. The values of the three input parameters input1, input2 and input3 are first concatenated in a vector representation (*[x,y,z]*), and the command parameter becomes complete. Use this version of *haSendCommand()* when 3 separate elements of a 3D vector are to be supplied.

E.G: *haSendCommand (dev, "set myBlock pos", blockPosX, blockPosY, blockPosZ, response)*
If, say, blockPosX = 0.1, blockPosY = -0.1 and blockPosZ = 0.0, then the complete command being sent through *haSendCommand()* looks like this:
"set myBlock pos [0.1,-0.1,0.0]".

4) Command with one Vector3d object

haSendCommand (long int dev, char command,*
Vector3d input,
char response)*

This version expects 4 parameters. The *command* parameter is **incomplete**. The 3 elements of the Vector3d object supplied by the parameter *input* are first concatenated in a vector representation (*[x,y,z]*), and the command parameter becomes complete. Use this version of *haSendCommand()* when the 3D vector is kept in a Vector3d object (available if you use the *HapticMaster.h* library).

E.G: *haSendCommand (dev, "set myBlock pos", blockPos, response)*
If, say, blockPos is a Vector3d object with the following values: blockPos.x = 0.1, blockPos.y = -0.1 and blockPos.z = 0.0, then the complete command being sent through *haSendCommand()* looks like this:
"set myBlock pos [0.1,-0.1,0.0]".

The *main()* function creates the haptic objects using different versions of the *haSendCommand()* function.

First, **the block** “myFloor” is created:
position [0,0,0] (supplying 3 separate double values),
size [0.35,0.35.0.01] (supplying 3 separate double values),
stiffness 20000 (supplying 1 scalar double value),
enable (complete command without parameters).

The **block** “myCube1” is created afterwards:
position [0.1,-0.1,0.02] (supplying 1 Vector3d object),
size [0.06,0.06.0.06](supplying 1 Vector3d object),
attitude [0.0,0.0,0.0,1.0] (supplying 4 separate double values),
stiffness 20000 (supplying 1 scalar double value),
enable (complete command without parameters).

The **block** “myCube2” is created afterwards:
position [0.1,0.0,0.02] (supplying 1 Vector3d object),
size [0.06,0.06.0.06](supplying 1 Vector3d object),
attitude [0.0,0.382,0.0,0.923] (supplying 4 separate double values),
stiffness 20000 (supplying 1 scalar double value),
enable (complete command without parameters).

The **block** “myCube3” is created afterwards:
position [0.1,0.1,0.02] (supplying 1 Vector3d object),
size [0.06,0.06.0.06](supplying 1 Vector3d object),
attitude [0.382,0.0, 0.0,0.923] (supplying 4 separate double values),
stiffness 20000 (supplying 1 scalar double value),
enable (complete command without parameters).

Then the **sphere** “mySphere” is created:
position [0,0,0.02] (supplying 3 separate double values),
radius 0.03 (supplying 3 separate double values),
stiffness 20000 (supplying 1 scalar double value),
enable (complete command without parameters).

Finally, the **torus** “myTorus” is created:
position [-0.1,0.1,0.08] (supplying 3 separate double values),
ringradius 0.05 (supplying 1 scalar double value),
tuberadius 0.02 (supplying 1 scalar double value),
stiffness 20000 (supplying 1 scalar double value),
enable (complete command without parameters).

Finally, OpenGL functions are registered and the OpenGL’s main loop is started.

8 Haptic Effects

In previous chapters we've shown several examples of how to build simple 3D virtual worlds using object creation that the HapticAPI provides. These worlds were then also rendered graphically on the screen by the OpenGL graphics engine.

Now it is time to add so called Haptic Effects to your virtual haptic worlds.
But first, let's define the concept of *a Haptic Effect*.

A haptic effect is the effect that a spring, damper or a constant force would have on the End Effector of the HapticMASTER so you can feel the result of the effect. The characteristics of effects are modeled in the HapticMASTER real-time computer.

So, you can create spring effects, damper effects, constant force effects or any combination of them and add them to your virtual haptic world. The *haSendCommand()* function is being used to create a haptic effect as well as change its parameters.

In complete analogy with the commands "create block myBlock" and "create sphere mySphere", the commands "create spring mySpring", "create damper myDamper" and "create biasforce myBiasForce" can be sent to create these haptic effects. Consequent "set" commands can determine the specific properties of the created haptic effect.

In the following paragraphs some examples will be presented that will demonstrate the use of Haptic Effects. Each of the three haptic effects will be drawn on the computer monitor in a different way:
A **spring** will be drawn as a purple line connecting the End Effector with a small purple ball at the end position of the spring.
A **bias force** will be drawn as a purple arrow.
It is difficult to try and draw the **damper** effect, since it is a global effect, but we try to symbolize its presence by an opaque cube that demonstrates a cube filled with a gelatin that damps the movement of the End Effector once the End Effector is found within this space.

Another effect is the **shaker** which generates oscillations, moving the end effector around a specific point in space.

Example 5 - Source Code

Example 5 : Haptic Effects

```
//---------------------------------------------------------------------
//                 0 5   -   H A P T I C   E F F E C T S
//
// This example demonstrates how to create haptic effects in the haptic
// world. The damper is being used in this example and is being activated
// only when the end effector is inside the cube.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#ifdef WIN32
   #include <windows.h>
   #include <process.h>
#endif

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];
char gCurrentState[30] = "unknown";

double CurrentPosition[3];

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Block OpenGL Material Parameters.
GLfloat BlockAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat BlockDiffuse[] = {0.00, 0.80, 0.67, 0.58};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

//---------------------------------------------------------------------
// O B J E C T   P A R A M E T E R S
//---------------------------------------------------------------------
double cubeSize = 0.15;
double damperCoef[] = {50.0,50.0,50.0};
```

```
//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                    B L O C K   M A T E R I A L
//
// BlockMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The Block.
//---------------------------------------------------------------------
void BlockMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlockAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlockDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                       D R A W   B L O C K
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Haptic Block Object In OpenGl
//---------------------------------------------------------------------
void DrawBlock(void)
{
   BlockMaterial();
   glutSolidCube(cubeSize);
}

//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                        I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{
   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//---------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//---------------------------------------------------------------------
void Display (void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   // define eyepoint in such a way that
   // drawing can be done as in lab-frame rather than sgi-frame
   // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelpos ==> %s", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
                     CurrentPosition[PosZ] );
   }

   if ( CurrentPosition[PosX]>=-(cubeSize/2.0) && CurrentPosition[PosX]<=cubeSize/2.0 &&
        CurrentPosition[PosY]>=-(cubeSize/2.0) && CurrentPosition[PosY]<=cubeSize/2.0 &&
        CurrentPosition[PosZ]>=-(cubeSize/2.0) && CurrentPosition[PosZ]<=cubeSize/2.0 ) {
      haSendCommand(dev, "set myDamper enable", response);
      printf("set myDamper enable ==> %s\n", response);
   }
   else {
      haSendCommand(dev, "set myDamper disable", response);
      printf("set myDamper disable ==> %s\n", response);
```

```
   }

   DrawEndEffector();
   DrawBlock();

   glPopMatrix ();
   glutSwapBuffers();
}

//---------------------------------------------------------------------
//                          R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//---------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
    glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    float fAspect = (float)iWidth/iHeight;
    gluPerspective (30.0, fAspect, 0.05, 20.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;
   }
}
```

```
//---------------------------------------------------------------------
//                            M A I N
//
// 05-Haptic-Effects Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      // Create the Haptic damper Effect and supply it with parameters
      if ( haSendCommand(dev, "create damper myDamper", response) ) {
         printf("---ERROR: Could not send command create damper myDamper\n");
         getchar();
         exit(-1);
      }

      printf("create damper myDamper ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:") ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand(dev, "set myDamper dampcoef", damperCoef[0], damperCoef[1],
                       damperCoef[2], response);
         printf("set myDamper dampcoef [%g,%g,%g] ==> %s\n", damperCoef[0],
                damperCoef[1], damperCoef[2], response);

         haSendCommand(dev, "set myDamper enable", response);
         printf("set myDamper enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example05: Haptic effects -
                         Regional Damper");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 5 - Description

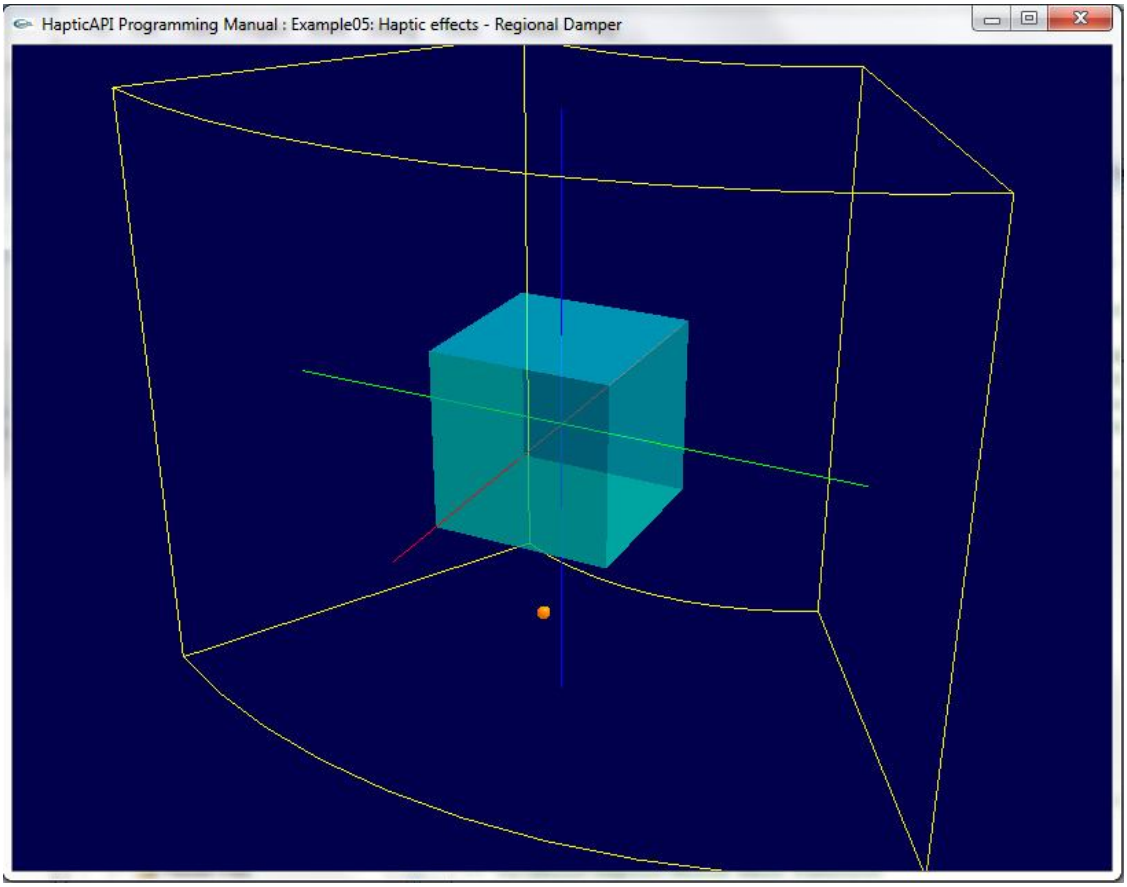


Figure 8-1 : Example 5 Screen Shot

Example 5 - Source Code", makes use of a damper effect. Next we will explain the code step by step. You should be familiar with the code that repeats itself within the previous examples. Exceptions and changes for this particular example are described in the following subsections.

OpenGL Materials

By supplying the *BlockDiffuse[]* vector with a 0.58 in the fourth element (alpha channel), the cube is drawn by OpenGL with a certain opacity.

Object parameters

The variable *cubeSize* determines the size of the cube.
You can play with this variable's value to determine the size of the gelatin cube in which the damper effect is active.

The display() function

This function sets the perspective view, and draws the HapticMASTER's axes and workspace. Then, the End Effector's position is queried from the HapticMASTER. If the End Effector happens to be found within the gelatin cube (within -cubeSize/2.0 and cubeSize/2.0 in all 3 axes), the damper effect with the name *myDamper* is enabled. If the End Effector is not in the cube, the effect *myDamper* is disabled.
Further, the End Effector and the block are drawn.

The main() function

In the main() function, the damper effect is created by calling the *haSendCommand()* function with the command "create damper myDamper".

Consequently, the damping coefficient of the damper is set by sending the "set myDamper dampcoef [50.0,50.0,50.0]" command. The damping factor can be set for each axis separately by providing the coefficient for this axis.

You are welcome to play with these values to feel the damping difference.

9 More Haptic Effects

In chapter 8, Haptic Effects, an introduction to haptic effects was presented and one example was shown to get the idea behind haptic effects. That example used the damper effect. The damper effect was enabled or disabled, depending on the position of the HapticMASTER's End Effector. This way a local damper effect could be simulated. If the damper stays enabled all the time (no IF clause), a global damper is simulated, meaning the damping is active in the complete workspace.

In this chapter we will show you another haptic effect that you can create within the HapticMASTER's haptic world. This is the **spring** effect, created with (surprise, surprise…) the "create spring" command. In this example we use the spring effect in combination with the damper effect from the previous example. The damper effect will realize a box where you can switch the spring effect on or off, therefore the name of this example – "Switchable spring".

Example 6 - Source Code

Example 6: Haptic Effects 01 – Switchable Spring

```
//---------------------------------------------------------------------
//      0 6   -   M O R E   H A P T I C   E F F E C T S   0 1
//
// This example demonstrates how to create haptic effects in the haptic
// world. The spring effect is being used in this example.
// A spring is created with a specific position (not at the origin of
// the cartesian axes). When the end effector is moved inside the small
// cube and stays there for one second, the spring effect is turned
// on or off (flipped).
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

bool gSpringActive = true;
unsigned int gInCubeCounter = 0;

double CurrentPosition[3];

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Spring OpenGL Material Parameters.
GLfloat SpringAmbient[] = {1.00, 0.00, 1.00, 1.00};
GLfloat SpringDiffuse[] = {0.97, 0.0, 0.97, 1.00};

// Block OpenGL Material Parameters.
GLfloat BlockAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat BlockDiffuse[] = {0.00, 0.80, 0.67, 0.58};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

//---------------------------------------------------------------------
// O B J E C T   P A R A M E T E R S
//---------------------------------------------------------------------
double cubeSize = 0.05;
double damperCoef[] = {50.0,50.0,50.0};
double springStiffness = 100;
double springPos[] = {0.15, 0.05, -0.05};
double springDampFactor = 0.7;
double springMaxForce = 7.0;
```

```
//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                   S P R I N G   M A T E R I A L
//
// SpringMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The spring.
//---------------------------------------------------------------------
void SpringMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, SpringAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, SpringDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                    B L O C K   M A T E R I A L
//
// BlockMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The Block.
//---------------------------------------------------------------------
void BlockMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlockAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlockDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                       D R A W   B L O C K
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The Haptic Block Object In OpenGl
//---------------------------------------------------------------------
void DrawBlock(void)
{
   BlockMaterial();
   glutSolidCube(cubeSize);
}
```

```
//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                 D R A W   S P R I N G   P O S
//
// This Function Is Called To Draw The origin position of the spring
//---------------------------------------------------------------------
void DrawSpringPos(void)
{
   SpringMaterial();
   glPushMatrix();
   glTranslatef(springPos[PosX], springPos[PosY], springPos[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                       D R A W   S P R I N G
//
// This Function Is Called To Draw The spring itself
//---------------------------------------------------------------------
void DrawSpring(void)
{
   SpringMaterial();
   glBegin(GL_LINES);
      glVertex3f(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
      glVertex3f(springPos[PosX], springPos[PosY], springPos[PosZ]);
   glEnd();

}

//---------------------------------------------------------------------
//                       I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{

   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   glEnable (GL_BLEND);
```

```
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//---------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//---------------------------------------------------------------------
void Display (void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();

   // define eyepoint in such a way that
    // drawing can be done as in lab-frame rather than sgi-frame
    // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if ( strstr ( response, "--- ERROR:") ) {
      printf("get modelpos ==> %s", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
CurrentPosition[PosZ] );
   }

   if ( CurrentPosition[PosX]>=-(cubeSize/2.0) && CurrentPosition[PosX]<=cubeSize/2.0 &&
        CurrentPosition[PosY]>=-(cubeSize/2.0) && CurrentPosition[PosY]<=cubeSize/2.0 &&
        CurrentPosition[PosZ]>=-(cubeSize/2.0) && CurrentPosition[PosZ]<=cubeSize/2.0 ) {
      haSendCommand(dev, "set myDamper enable", response);
      printf("set myDamper enable ==> %s\n", response);

      if (gInCubeCounter > 100) {
         if (gSpringActive) {
            haSendCommand(dev, "set mySpring disable", response);
            printf("set mySpring disable ==> %s\n", response);

            gSpringActive = false;
         }
         else {
            haSendCommand(dev, "set mySpring enable", response);
            printf("set mySpring enable ==> %s\n", response);

            gSpringActive = true;
         }
         gInCubeCounter = 0;
      }
      else {
         gInCubeCounter++;
      }
   }
   else {
      haSendCommand(dev, "set myDamper disable", response);
      printf("set myDamper disable ==> %s\n", response);
   }

   if (gSpringActive) {
      DrawSpring();
   }
```

```
   DrawEndEffector();
   DrawSpringPos();
   DrawBlock();

   glPopMatrix ();
   glutSwapBuffers();
}

//---------------------------------------------------------------------
//                          R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//---------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
    glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    float fAspect = (float)iWidth/iHeight;
    gluPerspective (30.0, fAspect, 0.05, 20.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;
   }
}

//---------------------------------------------------------------------
//                            M A I N
//
// 06-More-Haptic-Effects-01 Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      // Create the Haptic damper Effect and supply it with parameters
      if ( haSendCommand(dev, "create damper myDamper", response) ) {
         printf("--- ERROR: Could not send command create damper myDamper\n");
         getchar();
         exit(-1);
      }
```

```
      printf("create damper myDamper ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand(dev, "set myDamper dampcoef", damperCoef[0], damperCoef[1],
                       damperCoef[2], response);
         printf("set myDamper dampcoef [%g,%g,%g] ==> %s\n", damperCoef[0],
                damperCoef[1], damperCoef[2], response);

         haSendCommand(dev, "set myDamper enable", response);
         printf("set myDamper enable ==> %s\n", response);
      }

      // Create the Haptic spring Effect and supply it with parameters
      if ( haSendCommand(dev, "create spring mySpring", response) ) {
         printf("--- ERROR: Could not send command create spring mySpring\n");
      }

      printf("create spring mySpring ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand(dev, "set mySpring stiffness", springStiffness, response);
         printf("set mySpring stiffness %g ==> %s\n", springStiffness, response);

         haSendCommand(dev, "set mySpring pos", springPos[PosX], springPos[PosY], |
                       springPos[PosZ], response);
         printf("set mySpring pos [%g,%g,%g] ==> %s\n", springPos[PosX], springPos[PosY],
                springPos[PosZ], response);

         haSendCommand(dev, "set mySpring dampfactor", springDampFactor, response);
         printf("set mySpring dampfactor %g ==> %s\n", springDampFactor, response);

         haSendCommand(dev, "set mySpring maxforce", springMaxForce, response);
         printf("set mySpring maxforce %g ==> %s\n", springMaxForce, response);

         haSendCommand(dev, "set mySpring enable", response);
         printf("set mySpring enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example06: More Haptic effects -
                         Switchable Spring");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 6 - Description

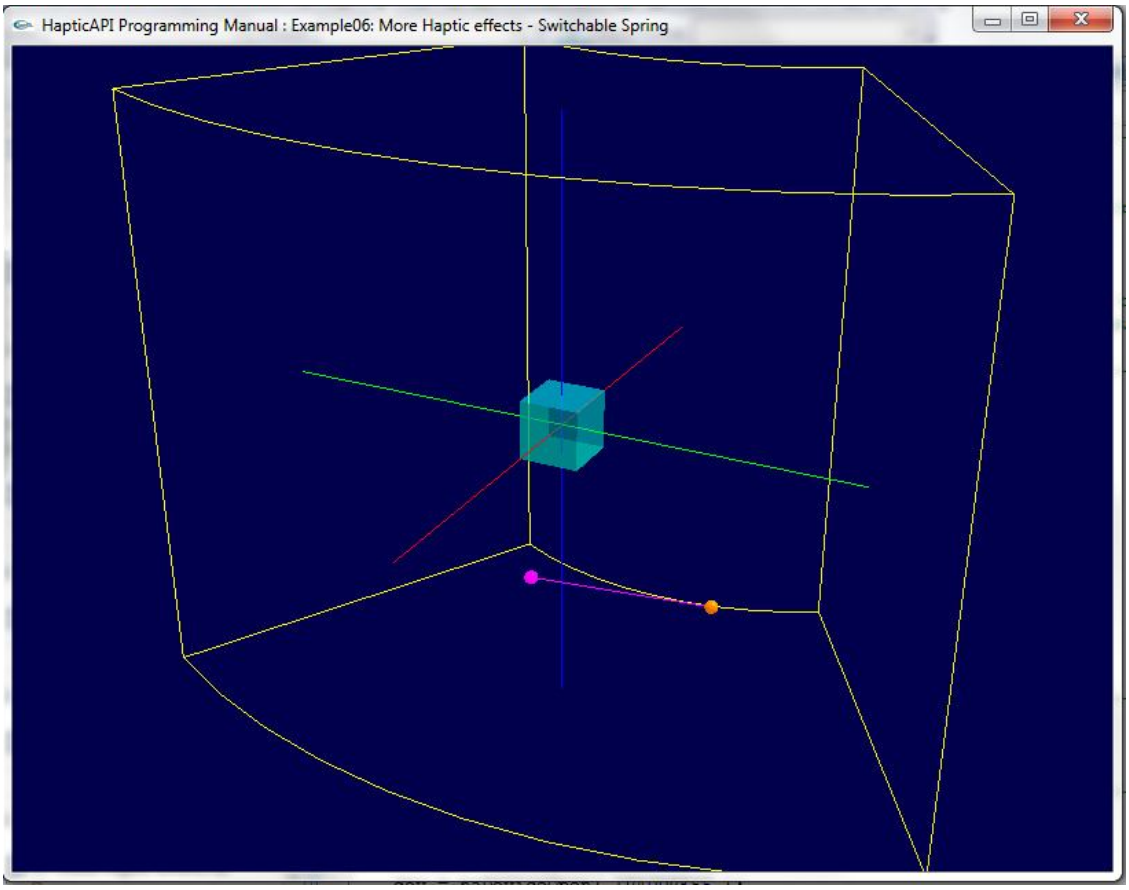


Figure 9-1 : Example 6 Screen Shot

The source code creates a **spring** effect. The spring effect is enabled when the program starts. That means the End Effector is brought to the given spring position, which is [0.15, 0.05, -0.05]. You can move the End Effector as before, but as long as the spring effect is enabled, the End Effector will always return to the spring position when no force is applied on the End Effector. You can indeed see this as a spring connecting the End Effector with the spring's position (on the screen, the spring itself is the purple line and the spring position is the purple ball. The example also creates a small gelatin cube in the center of the workspace ([0.0, 0.0, 0.0]), similar to the one we created in the previous example only smaller. If you bring the End Effector inside this cube and stay there for about a second, the spring effect will toggle between enabled and disabled. When the spring effect is disabled, the purple line disappears, and you will notice the spring does not pull anymore in the direction of its position. If you enter the cube again and stay there for a while, the spring will be enabled again. Following is some explanation about the parts of the code that are new in this example.

Object parameters

cubeSize determines the gelatin cube's size.
damperCoef[] are de damping coefficients inside the gelatin cube.
springStiffness determines the spring's stiffness
springPos[] determines the spring's position (in 3D coordinates)
springDampFactor determines how efficient the End Effector will stop around the spring's position. When this factor is 1.0, the End Effector doesn't overshoot (passes) the position of the spring. If the factor is 0.7, the End Effector slightly overshoots the position of the spring and then returns to the correct position.
springMaxForce determines the maximum force that the spring can apply on the End Effector. This value is given in Newtons. **Take care with changing this value, since the HapticMASTER can make very swift and powerful movements if this value is set too high!**

DrawSpringPos()

This function draws the purple ball, representing the spring's position.

DrawSpring()

This function draws the purple line, representing the spring itself.
The function is called only if the spring is active.

Display()

As usual:
- Draws the axes and the workspace.
- Queries the End Effector position.

If the End Effector is found within the gelatin box, the damper effect is enabled.

A global counter *gInCubeCounter* keeps track of the number of frames since we entered the gelatin cube. When the variable *gInCubeCounter* is not yet greater than 100, the counter is increased. When it is greater than 100, it means the End Effector is already about one second inside the cube. We then check whether the global variable *gSpringActive* is true or false. If it's true, we disable the spring effect and set the variable to false. If it's false, we enable the spring effect and set the variable to true. Notice the spring effect parameters are all set in the *main()* function. With respect to the spring effect, the *Display()* function takes care only of enabling / disabling this effect. This is the way we realize the flip between an enabled and disabled spring.

If the spring effect is enabled, the spring line is drawn, followed by the End Effector, the spring position and the gelatin cube.

The main() function

The main function opens the device and initializes it if necessary.
Consequently it creates a damper effect using the “create damper myDamper” command and sets its damping coefficients (“set myDamper dampcoef”, damperCoef[0], damperCoef[1], dampetCoef[2]). The damper effect is then enabled.
Then, the spring effect is created (“create spring mySpring”) and its parameters are set using the “set mySpring” commands. Stiffness, pos, dampfactor and maxforce properties are set, copied from the corresponding variables. The spring effect is then enabled.

The rest of the code is the usual OpenGL function registration and the command to run OpenGL’s main loop.

Example 7 - Source Code

Example 7: More Haptic Effects 02 – Magnetic Field

```
//---------------------------------------------------------------------
//      0 7   -   M O R E   H A P T I C   E F F E C T S   0 2
//
// This example demonstrates how to create haptic effects in the haptic
// world. The constant force effect is being used in this example.
// When the end effector enters the "magnetic field" region, a constant
// force is applied on the end effector in the Y direction.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double CurrentPosition[3];

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Arrow OpenGL Material Parameters.
GLfloat ArrowAmbient[] = {1.00, 0.00, 1.00, 1.00};
GLfloat ArrowDiffuse[] = {0.97, 0.0, 0.97, 1.00};

// Block OpenGL Material Parameters.
GLfloat BlockAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat BlockDiffuse[] = {0.00, 0.80, 0.67, 0.58};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}
```

```
//---------------------------------------------------------------------
//                   A R R O W   M A T E R I A L
//
// ArrowMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing an arrow in the magnetic field.
//---------------------------------------------------------------------
void ArrowMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, ArrowAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, ArrowDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                    B L O C K   M A T E R I A L
//
// BlockMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The Block.
//---------------------------------------------------------------------
void BlockMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlockAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlockDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                       D R A W   B L O C K
//
// This Function Is Called To Draw The dividing wall
//---------------------------------------------------------------------
void DrawBlock(void)
{
   BlockMaterial();
   glScalef(0.4, 0.005, 0.5);
   glTranslatef(0.0, 0.0, 0.0);
   glutSolidCube(1.0);
}

//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                      D R A W   A R R O W
//
// This Function Is Called To Draw one arrow in the magnetic field
//---------------------------------------------------------------------
void DrawArrow(double arrowXPos, double arrowZPos)
{
   ArrowMaterial();

   glBegin(GL_LINES);
      glVertex3f(arrowXPos, 0.07, arrowZPos);
      glVertex3f(arrowXPos, 0.07 + 0.1, arrowZPos);
   glEnd();

   glBegin(GL_LINES);
      glVertex3f(arrowXPos, 0.07 + 0.02, arrowZPos + 0.01);
      glVertex3f(arrowXPos, 0.07 + 0.02, arrowZPos - 0.01);
   glEnd();

   glBegin(GL_LINES);
      glVertex3f(arrowXPos, 0.07 + 0.02, arrowZPos + 0.01);
      glVertex3f(arrowXPos, 0.07, arrowZPos);
   glEnd();

   glBegin(GL_LINES);
      glVertex3f(arrowXPos, 0.07 + 0.02, arrowZPos - 0.01);
      glVertex3f(arrowXPos, 0.07, arrowZPos);
   glEnd();
}

//---------------------------------------------------------------------
//             D R A W   M A G N E T I C   F I E L D
//
// This Function Is Called To Draw The magnetic field
//---------------------------------------------------------------------
void DrawMagneticField(void)
{
   for (int i = -1; i <= 1; i++) {
      for (int j = -1; j <= 1; j++) {
         double tempX = i * 0.15;
         double tempZ = j * 0.15;
         DrawArrow(tempX, tempZ);
      }
   }
}

//---------------------------------------------------------------------
//                        I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{

   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
```

```
   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//-----------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//-----------------------------------------------------------------------
void Display (void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();

   // define eyepoint in such a way that
    // drawing can be done as in lab-frame rather than sgi-frame
    // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );

   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelpos ==> %s", response);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
                     CurrentPosition[PosZ] );
   }

   if ( CurrentPosition[PosY] > 0.0 ) {
      haSendCommand(dev, "set myBiasForce enable", response);
      printf("set myBiasForce enable ==> %s\n", response);
   }
   else {
      haSendCommand(dev, "set myBiasForce disable", response);
      printf("set myBiasForce disable ==> %s\n", response);
   }

   DrawEndEffector();
   DrawMagneticField();
   DrawBlock();

   glPopMatrix ();
   glutSwapBuffers();
}

//-----------------------------------------------------------------------
//                            R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//-----------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
    glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    float fAspect = (float)iWidth/iHeight;
    gluPerspective (30.0, fAspect, 0.05, 20.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}
```

```
//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;
   }
}
```

```
//---------------------------------------------------------------------
//                            M A I N
//
// 07-More-Haptic-Effects-02 Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      // Create the Haptic BiasForce Effect and supply it with parameters
      if ( haSendCommand(dev, "create biasforce myBiasForce", response) ) {
         printf ("--- ERROR: Could not send command create biasforce myBiasForce\n");
      }

      printf("create biasforce myBiasForce ==> %s\n", response);

      if ( strstr (response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         printf("create biasforce myBiasForce ==> %s\n", response);

         haSendCommand(dev, "set myBiasForce force [0.0,-1.0,0.0]", response);
         printf("set myBiasForce force [0.0,-1.0,0.0] ==> %s\n", response);

         haSendCommand(dev, "set myBiasForce enable", response);
         printf("set myBiasForce enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example07: More Haptic effects -
                         Magnetic Field");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 7 - Description

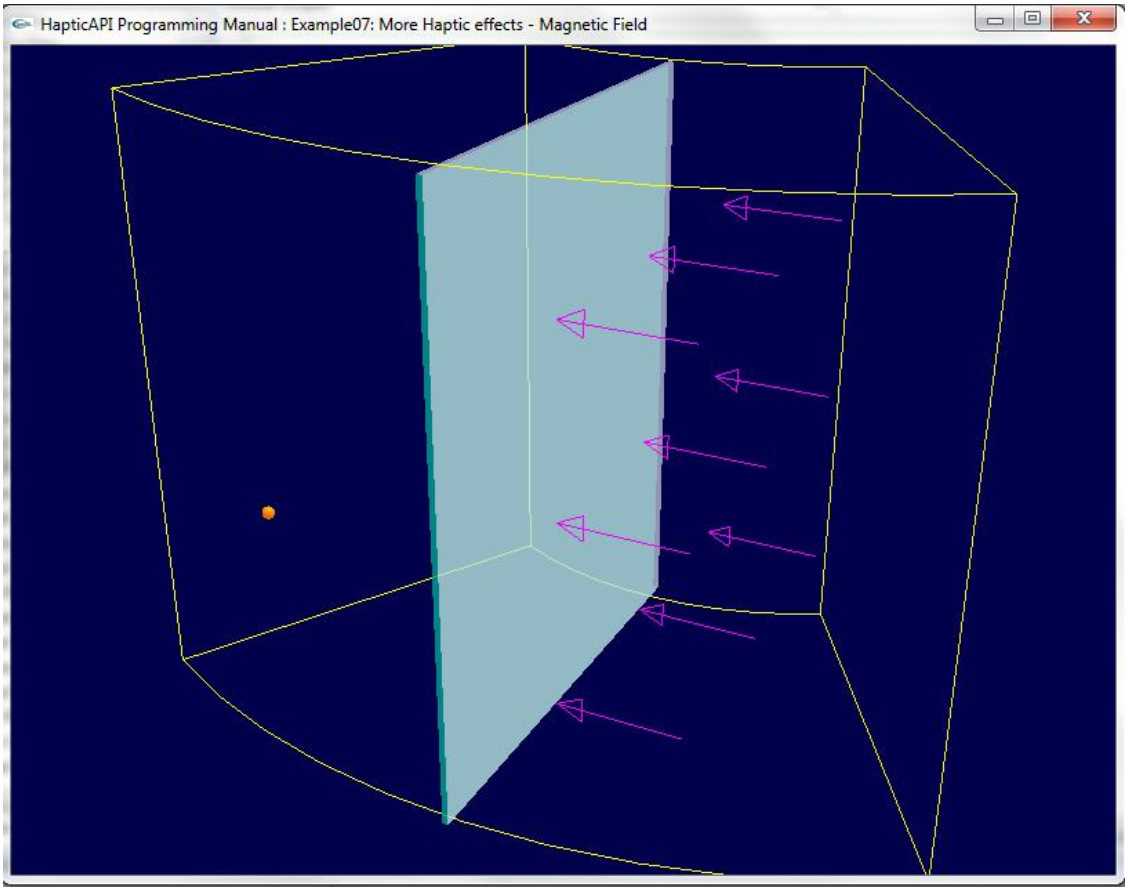


Figure 9-2 : Example 7 Screen Shot

Example 6: Haptic Effects creates a **bias force** effect. In this example we want to simulate a magnetic field that is active in one half of the HapticMASTER's working space. An opaque layer is drawn, virtually dividing the working space in two sides – namely the **positive y side** and the **negative y side**.
When the End Effector resides in the negative Y side, no bias force is active. If you move the End Effector beyond the "membrane" and it enters the positive Y side, the "magnetic field force" is applied and the End Effector is pushed back with a constant force in the direction of the negative Y side.
This constant force results in an accelerated movement of the End Effector in the direction of the negative Y axis. When the End Effector leaves the magnetic field, it maintains its speed and does not accelerate anymore. (Actually it decelerates a little bit due to the simulated friction but it is definitely not accelerating anymore due to the magnetic field).
The code of example 7 should be straight forward by now, but following is some explanation of the difference with respect to the previous examples.

DrawBlock()
This function draws the dividing wall between the negative Y side and the positive Y side. This block is not rendered in the HapticMASTER's haptic world. It is only for visualizing the division of the working space into two parts.

DrawArrow()
This function draws one single purple arrow, representing the biased force. Given the arrowXPos and arrowZPos, the 4 lines constructing the arrow are drawn.

DrawMagneticField()
This function draws nine arrows in the positive Y side of the working space, representing the magnetic field present on this side.
It is simply a nested loop that calls the *DrawArrow()* function 9 times in different X and Z positions.

Display()
Workspace is drawn, End Effector position is queried.
If End Effector is in the positive Y axis, the biasforce effect is enabled ("set myBiasForce enable"), otherwise disable ("set myBiasForce disable").
End Effector, Magnetic Field and the dividing wall are drawn.

The main() function
The HapticMASTER is opened and initialized. The biasforce effect is created ("create biasforce myBiasForce"). The bias force is assigned with a constant direction [0.0,-1.0,0.0] ("set myBiasForce force [0.0,-1.0,0.0]"). The rest of the code is the usual OpenGL function registration and the command to run OpenGL's main loop.

10 Force Measurement

In the following two chapters, some advanced topics are presented. We will show you two not so obvious, but very useful techniques to operate the HapticMASTER. Again, for both topics there will be examples shown and explained to you.

This chapter shows you how to use the HapticMASTER for force measurement. After all, the HapticMASTER is controlled by a very sensitive force sensor.

So, in the example shown in the following pages a simple haptic world is created and shown on the screen. Basically, this is the world we created in the previous chapter. However, this time some code is added to plot graphs of several parameters including X, Y and Z forces, that can be read from the HapticMASTER, on the window's right side, next to the representation of the workspace and haptic world.

PLEASE NOTICE this example is used only for visualization purposes. When you want to have guarantee about the time the samples are taken, you need to use the Real-time data logger (see example 13 in this manual)

Example 8 - Source Code

Example 8: Force Measurement

```
//---------------------------------------------------------------------
//           0 8   -   F O R C E   M E A S U R E M E N T
//
// This example demonstrates how to query the HapticMASTER sensors
// and display the data in an OpenGL window.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double CurrentPosition[3];

double ViewportWidth;
double ViewportHeight;

//---------------------------------------------------------------------
// M E A S U R E D   P A R A M E T E R S
//---------------------------------------------------------------------
const int MaxParams = 10;
const int MaxSamples = 50;
const int MaxSampleNr = MaxSamples-1;
int SampleNr;
double ParamSamples[MaxParams][MaxSamples];

double ParamMax[MaxParams] = {0.19, 0.25, 0.2, 1.0, 1.0, 1.0, 35.0, 35.0, 35.0, 10.0};

double ParamScale[MaxParams] = {1.0, 1.0, 1.0, 0.25, 0.25, 0.25, 0.0075, 0.0075, 0.0075,
1.0};

char ParamNameStrings[MaxParams][10] = {"X-Pos", "Y-Pos", "Z-Pos",
                                        "X-Vel", "Y-Vel", "Z-Vel",
                                        "X-Force", "Y-Force", "Z-Force", "Inertia"};

char ParamUnitStrings[MaxParams][8] = {"[m]", "[m]", "[m]",
                                       "[m/s]", "[m/s]", "[m/s]",
                                       "[N]", "[N]", "[N]", "[kg]"};

char ParamValueStrings[MaxParams][11];

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// Spring OpenGL Material Parameters.
GLfloat SpringAmbient[] = {1.00, 0.00, 1.00, 1.00};
GLfloat SpringDiffuse[] = {0.97, 0.0, 0.97, 1.00};

GLfloat DarkGrayLineAmbient[] = {0.25, 0.25, 0.25, 1.00};
GLfloat DarkGrayLineDiffuse[] = {0.25, 0.25, 0.25, 1.00};
```

```
GLfloat GrayLineAmbient[] = {0.650, 0.650, 0.650, 1.00};
GLfloat GrayLineDiffuse[] = {0.650, 0.650, 0.650, 1.00};

GLfloat BlueLineAmbient[] = {0.00, 0.76, 0.70, 1.00};
GLfloat BlueLineDiffuse[] = {0.00, 0.90, 0.77, 1.00};

// Block OpenGL Material Parameters.
GLfloat BlockAmbient[] = {0.00, 0.66, 0.60, 1.00};
GLfloat BlockDiffuse[] = {0.00, 0.80, 0.67, 0.28};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

//---------------------------------------------------------------------
// O B J E C T   P A R A M E T E R S
//---------------------------------------------------------------------
double springStiffness = 100;
double springMaxForce = 8.0;
double springPos[] = {0.15, 0.05, -0.05};
double springDampFactor = 0.7;

//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                  S P R I N G   M A T E R I A L
//
// SpringMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The spring.
//---------------------------------------------------------------------
void SpringMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, SpringAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, SpringDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                B L O C K   M A T E R I A L
//
// BlockMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing The Block.
//---------------------------------------------------------------------
void BlockMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlockAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlockDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}
```

```
//---------------------------------------------------------------------
//              B L U E   L I N E   M A T E R I A L
//
// BlueLineMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing A Line Segment.
//---------------------------------------------------------------------
void BlueLineMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, BlueLineAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, BlueLineDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//              G R A Y   L I N E   M A T E R I A L
//
// GrayLineMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing A Line Segment.
//---------------------------------------------------------------------
void GrayLineMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, GrayLineAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, GrayLineDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//         D A R K   G R A Y   L I N E   M A T E R I A L
//
// DarkGrayLineMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing A Line Segment.
//---------------------------------------------------------------------
void DarkGrayLineMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, DarkGrayLineAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, DarkGrayLineDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                D R A W   S P R I N G   P O S
//
// This Function Is Called To Draw The origin position of the spring
//---------------------------------------------------------------------
void DrawSpringPos(void)
{
   SpringMaterial();
   glPushMatrix();
   glTranslatef(springPos[PosX], springPos[PosY], springPos[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                        D R A W   S P R I N G
//
// This Function Is Called To Draw The spring itself
//---------------------------------------------------------------------
void DrawSpring(void)
{
   SpringMaterial();
   glBegin(GL_LINES);
      glVertex3f(CurrentPosition[PosX], CurrentPosition[PosY], CurrentPosition[PosZ]);
      glVertex3f(springPos[PosX], springPos[PosY], springPos[PosZ]);
   glEnd();

}

//---------------------------------------------------------------------
//             D R A W   P A R A M   G R A P H
//
// This Function Plots A Graph Of Some HapticMASTER Parameter
// On The Screen
//---------------------------------------------------------------------
void DrawParamGraph(int Param)
{
   int i;
   double ScaleX = 0.7 / MaxSamples;
   double OffsetX = -0.35;

   BlueLineMaterial();

   if(SampleNr == MaxSampleNr)
   {
      glBegin(GL_LINE_STRIP);
         for(i=0; i<=MaxSampleNr; i++)
            glVertex3f(0.0, OffsetX+(i*ScaleX),
                       (ParamSamples[Param][i]/ParamMax[Param])*0.20);
      glEnd();
   }
   else
   {
      glBegin(GL_LINE_STRIP);
         for(i=SampleNr+1; i<MaxSamples; i++)
            glVertex3f(0.0, OffsetX+((i-(SampleNr+1))*ScaleX),
                       (ParamSamples[Param][i]/ParamMax[Param])*0.25);

         for(i=0; i<=SampleNr; i++)
            glVertex3f(0.0, OffsetX+((i+MaxSampleNr-SampleNr)*ScaleX),
                       (ParamSamples[Param][i]/ParamMax[Param])*0.25);

      glEnd();
   }
```

```
   GrayLineMaterial();
   glBegin(GL_LINE_STRIP);
      glVertex3f(0.0, -0.35, -0.25);
      glVertex3f(0.0, -0.35, 0.25);
      glVertex3f(0.0, 0.35, 0.25);
      glVertex3f(0.0, 0.35, -0.25);
      glVertex3f(0.0, -0.35, -0.25);
   glEnd();

   glBegin(GL_LINE_STRIP);
      glVertex3f(0.0, 0.0, -0.25);
      glVertex3f(0.0, 0.0, 0.25);
   glEnd();

   DarkGrayLineMaterial();
   for(i=0; i<20; i++)
   {
      glBegin(GL_LINE_STRIP);
         glVertex3f(0.0, -0.35+(i*0.035), -0.25);
         glVertex3f(0.0, -0.35+(i*0.035), 0.25);
      glEnd();
   }

   for(i=0; i<3; i++)
   {
      glBegin(GL_LINE_STRIP);
         glVertex3f(0.0, -0.35, -0.125+(i*0.125));
         glVertex3f(0.0, 0.35, -0.125+(i*0.125));
      glEnd();
   }

   BlockMaterial();
   glBegin(GL_POLYGON);
      glVertex3f(0.0, -0.35, -0.25);
      glVertex3f(0.0, -0.35, 0.25);
       glVertex3f(0.0, 0.35, 0.25);
      glVertex3f(0.0, 0.35, -0.25);
   glEnd();
}

//---------------------------------------------------------------------
//                 D R A W   P A R A M   I N F O
//
// This Function Plots A Vu-Meter Like Graph Of Some HapticMASTER Parameter
// On The Screen
//---------------------------------------------------------------------
void DrawParamInfo(int Param)
{
   int i=0;

   BlueLineMaterial();
   glRasterPos3f(0.0, -0.32, 0.15);

   while( (i<sizeof(ParamNameStrings[Param])) && (ParamNameStrings[Param][i] != '\0'))
   {
      glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ParamNameStrings[Param][i]);
      i++;
   }

   for(i=0; i<sizeof(ParamUnitStrings[Param]); i++)
      glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ParamUnitStrings[Param][i]);

   glRasterPos3f(0.0, -0.25, 0.05);
   for(i=0; i<sizeof(ParamValueStrings[Param]); i++)
      glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ParamValueStrings[Param][i]);
```

```
   EndEffectorMaterial();
   glBegin(GL_POLYGON);
      glVertex3f(0.0, 0.0, -0.2);
      glVertex3f(0.0, 0.0, -0.125);
       glVertex3f(0.0, (ParamSamples[Param][SampleNr]/ParamMax[Param])*0.3, -0.125);
      glVertex3f(0.0, (ParamSamples[Param][SampleNr]/ParamMax[Param])*0.3, -0.2);
   glEnd();

   GrayLineMaterial();
   glBegin(GL_LINE_STRIP);
      glVertex3f(0.0, -0.35, -0.25);
      glVertex3f(0.0, -0.35, 0.25);
      glVertex3f(0.0, 0.35, 0.25);
      glVertex3f(0.0, 0.35, -0.25);
      glVertex3f(0.0, -0.35, -0.25);
   glEnd();

   glBegin(GL_LINE_STRIP);
      glVertex3f(0.0, -0.3, -0.2);
      glVertex3f(0.0, -0.3, -0.125);
      glVertex3f(0.0, 0.3, -0.125);
      glVertex3f(0.0, 0.3, -0.2);
      glVertex3f(0.0, -0.3, -0.2);
   glEnd();

   glBegin(GL_LINE_STRIP);
      glVertex3f(0.0, 0.0, -0.22);
      glVertex3f(0.0, 0.0, -0.105);
   glEnd();

   BlockMaterial();
   glBegin(GL_POLYGON);
      glVertex3f(0.0, -0.35, -0.25);
      glVertex3f(0.0, -0.35, 0.25);
      glVertex3f(0.0, 0.35, 0.25);
      glVertex3f(0.0, 0.35, -0.25);
   glEnd();
}

//---------------------------------------------------------------------
//                        I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{
   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}
```

```
//---------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//---------------------------------------------------------------------
void Display (void)
{
   int i;

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   // define eyepoint in such a way that
   // drawing can be done as in lab-frame rather than sgi-frame
   // (so X towards user, Z is up)

   glViewport (0, ViewportHeight*((MaxParams)/2), ViewportWidth*5,
               ViewportHeight*((MaxParams)/2));

   gluLookAt (1.0, 0.5, 0.35, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelpos ==> %s", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, CurrentPosition[PosX], CurrentPosition[PosY],
                     CurrentPosition[PosZ] );
   }

   DrawEndEffector();
   DrawSpring();
   DrawSpringPos();

   glPopMatrix();

   glPushMatrix();
   gluLookAt (1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

   SampleNr++;
   if (SampleNr >= MaxSamples)
      SampleNr = 0;

   haSendCommand(dev, "get modelpos", response);
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelpos ==> %s\n", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec(response, ParamSamples[0][SampleNr], ParamSamples[1][SampleNr],
                    ParamSamples[2][SampleNr]);
   }

   haSendCommand(dev, "get modelvel", response);
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelvel ==> %s\n", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec(response, ParamSamples[3][SampleNr], ParamSamples[4][SampleNr],
ParamSamples[5][SampleNr]);
```

```
   }

   haSendCommand(dev, "get measforce", response);
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get measforce ==> %s\n", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec(response, ParamSamples[6][SampleNr], ParamSamples[7][SampleNr],
                    ParamSamples[8][SampleNr]);
   }

   haSendCommand(dev, "get inertia", response);
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get inertia ==> %s\n", response);
      getchar();
      exit(-1);
   }
   else {
      ParamSamples[9][SampleNr] = atof(response);
   }

   for(i=0; i<MaxParams; i++)
      sprintf(ParamValueStrings[i], "%+08.5f", ParamSamples[i][SampleNr]);

   // For Each Parameter Draw The Graph
   for(i=0; i<MaxParams; i++)
   {
      glViewport(6*ViewportWidth, (MaxParams-i-1)*ViewportHeight, 4*ViewportWidth,
                 ViewportHeight);
      DrawParamGraph(i);
   }

   // For Each Parameter Draw The Meter
   for(i=0; i<MaxParams; i++)
   {
      glViewport(5*ViewportWidth, (MaxParams-i-1)*ViewportHeight, ViewportWidth,
                 ViewportHeight);
      DrawParamInfo(i);
   }

   glPopMatrix ();

   glutSwapBuffers();
}

//------------------------------------------------------------------------
//                           R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//------------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
   glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();

   float fAspect = (float)iWidth/iHeight;
   gluPerspective (30.0, fAspect, 0.05, 20.0);

   ViewportWidth = ((GLsizei)glutGet(GLUT_WINDOW_WIDTH)/10);
   ViewportHeight = ((GLsizei)glutGet(GLUT_WINDOW_HEIGHT)/MaxParams);

   glMatrixMode (GL_MODELVIEW);
   glLoadIdentity ();
}
```

```
//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;
   }
}
```

```
//---------------------------------------------------------------------
//                            M A I N
//
// 08-Force-Measurement Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      if ( haSendCommand (dev, "create spring mySpring", response) ) {
         printf ( "--- ERROR: Could not send command create spring mySpring\n" );
      }

      printf( "create spring mySpring ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand (dev, "set mySpring stiffness", springStiffness, response);
         printf( "set mySpring stiffness %g ==> %s\n", springStiffness, response);

         haSendCommand (dev, "set mySpring dampfactor", springDampFactor, response);
         printf( "set mySpring dampfactor %g ==> %s\n", springDampFactor, response);

         haSendCommand (dev, "set mySpring pos", springPos[PosX], springPos[PosY],
                        springPos[PosZ], response);
         printf( "set mySpring pos [%g,%g,%g] ==> %s\n", springPos[PosX],
                 springPos[PosY], springPos[PosZ], response);

         haSendCommand (dev, "set mySpring maxforce", springMaxForce, response);
         printf( "set mySpring maxforce %g ==> %s\n", springMaxForce, response);

         haSendCommand (dev, "set mySpring enable", response);
         printf( "set mySpring enable ==> %s\n", response);
      }
      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (1024, 768);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example08: Force Measurement");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 8 - Description

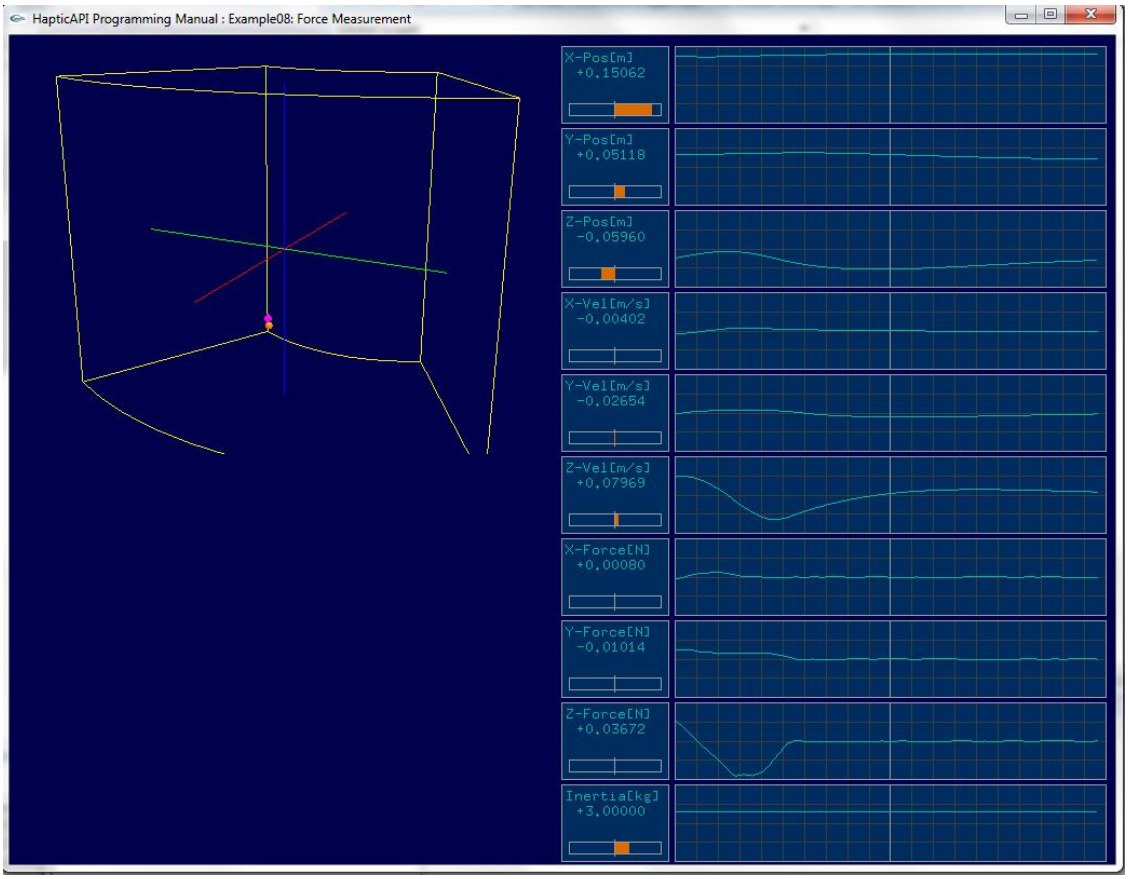


Figure 10-1 : Example 8 Screen Shot

The previous pages showed the source code for Example 8: Force Measurement". As with the other examples, the bigger part of the code is OpenGL code. We need all this code to create some nice moving graphs on the computer monitor showing the values that we read from the HapticMASTER.

Now when you run the application you should feel a spring object located somewhere in the workspace (not at the center of the HapticMASTER workspace) and when you move the End Effector, the graphs on the right side of the window will display the respective parameter values. The window should look like Figure 10-1 : Example 8 Screen Shot above. From the top down, the first three graphs show the current X, Y and Z positions of the HapticMASTER's End Effector. The next six show the velocity in the X, Y and Z direction and the force on the End Effector in the X, Y and Z direction. The last graph shows the HapticMASTER Inertia parameter, which is a constant value in this case.

Measured Parameters

MaxParams determines the number of measured variables.
MaxSamples determines the number of samples in each graph (history)
MaxSampleNr and *SampleNr* are used as indices for the array in the loop.
ParamSamples[][] is a matrix holding the history of all measured variables.
ParamMax[] determines the maximum values of each measured variable.
ParamScale[] determines the scale of each graph.
ParamNameStrings[] holds the names of all graphs.
ParamUnitStrings[] holds the units that are to be displayed by each graph.
ParamValueStrings[][] holds the queries values as a string.

Object Parameters

springStiffness determines the stiffness of the spring.
springPos determines the position of the spring.
springDampFactor determines the damp factor of the spring. Value 1.0 will generate a spring that pulls to its position without overshooting.

DrawParamGraph()

This function plots a graph of some HapticMASTER parameter.
First, the blue line of the measured values is drawn. Then the gray grid is plotted (20 vertical lines and 3 horizontal)

DrawParamInfo()

This function plots a Vu-Meter like graph of some HapticMASTER parameter. This is the small window on the left of each graph. It shows the current value of the parameter: The name of the parameter, the units, the current value as text and the current value as a bar.

Display()

First, the axes and the workspace are drawn.
Then the End Effector's position is queried.
The End Effector, the spring and the spring's position are drawn.
The End Effector's **position** is queried and copied into *ParamSamples[0][]*, *ParamSamples[1][]* and *ParamSamples[2][]*.
The End Effector's **velocity** is queried and copied into *ParamSamples[3][]*, *ParamSamples[4][]* and *ParamSamples[5][]*.
The **measured force** is queried and copied into *ParamSamples[6][]*, *ParamSamples[7][]* and *ParamSamples[8][]*.
The inertia is queried and copied to *ParamSamples[9]*.

Then, the queried values are copied to the string versions of the values (ParamSample[x] copied into ParamValueStrings[x]).

Finally, the graph and the Vu-meter of each measured parameter is drawn by calling the *DrawParamGraph()* and *DrawParamInfo()* functions respectively.

The main() function

Creates the spring effect in order to make things more interesting…

The next chapter will provide one more advanced example of using the HapticMASTER, namely moving the HapticMASTER to a certain location within its workspace by using the mouse.

11 Position Control – Mouse-slave

This chapter shows you how you can control the position of the HapticMASTER End Effector from you application program.
We have created an example which takes the mouse movements as input and makes the HapticMASTER follow that input depending on the mouse buttons – the so called mouse-slave system.

As there is, at this time, no direct way to tell the HapticMASTER to move to a certain location within its workspace (except for the state HOME), we have to find some other way. The solution we suggest here is that we create a spring effect with carefully chosen properties so that when we move the spring's position, the End Effector will follow this position, since it is virtually connected to the other end of the spring.
As you will soon find out, this solution works surprisingly well.

Another thing that you might learn from this chapter, although not described explicitly, is that you can move the haptic objects that you have created within the HapticMASTER's haptic world, by changing its position parameters from your application code. The HapticAPI reference manual describes the commands that are needed to do this.

Example 9 - Source Code

Example 9: Position Control – Mouse-Slave

```
//---------------------------------------------------------------------
// 0 9   -   P O S I T I O N   C O N T R O L   -   M O U S E   S L A V E
//
// This example demonstrates how to control the position of the Haptic
// Master. It uses the mouse inputs to change the HapticMASTER's position
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#include <math.h>

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double currentPosition[3];
double DamperCoeff[3] = {63.20, 63.20, 63.20};
double SpringPosition[3] = {0.0, 0.0, 0.0};
double SpringMoveSpeed = 0.0005;
double inertia = 4.0;

bool LeftButtonDown = false;
bool RightButtonDown = false;
int PreviousX = 0.0;
int PreviousY = 0.0;

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}
```

```
//---------------------------------------------------------------------
//                 D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(currentPosition[PosX], currentPosition[PosY], currentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                       I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{

   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//---------------------------------------------------------------------
//                         D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//---------------------------------------------------------------------
void Display (void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   // define eyepoint in such a way that
   // drawing can be done as in lab-frame rather than sgi-frame
   // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelpos ==> %s", response);
```

```
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, currentPosition[PosX], currentPosition[PosY],
                     currentPosition[PosZ] );
   }

   DrawEndEffector();
   glPopMatrix ();
   glutSwapBuffers();
}

//---------------------------------------------------------------------
//                           R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//---------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
    glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    float fAspect = (float)iWidth/iHeight;
    gluPerspective (30.0, fAspect, 0.05, 20.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

//---------------------------------------------------------------------
//                           K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;
    }
}
```

```
//---------------------------------------------------------------------
//                            M O U S E
//
// This Function Is Called By OpenGl WhenEver A mouse button is pushed.
// It records the position of the click and which button it was.
//---------------------------------------------------------------------
void Mouse(int Button, int State, int x, int y)
{
   switch(Button)
   {
      case GLUT_LEFT_BUTTON:
         switch(State)
         {
            case GLUT_DOWN:
               LeftButtonDown = true;
               PreviousX = x;
               PreviousY = y;
               break;
            case GLUT_UP :
               LeftButtonDown = false;
               break;
            default : break;
         }
         break;
      case GLUT_RIGHT_BUTTON:
         switch(State)
         {
            case GLUT_DOWN:
               RightButtonDown = true;
               PreviousX = x;
               PreviousY = y;
               break;
            case GLUT_UP :
               RightButtonDown = false;
               break;
            default : break;
         }
         break;
      case GLUT_MIDDLE_BUTTON:
         switch(State)
         {
            case GLUT_DOWN:
               break;
            case GLUT_UP :
               break;
            default : break;
         }
         break;
      default: break;
   } // switch
}
```

```
//---------------------------------------------------------------------
//                            M O T I O N
//
// This Function Is Called By OpenGl WhenEver the mouse moves with one
// of its buttons pushed.
// This function takes care of changing the position of the EndEffector
// dependent on which mouse button is pressed.
//---------------------------------------------------------------------
void Motion(int X, int Y)
{
   if (LeftButtonDown)
   {
      SpringPosition[1] += SpringMoveSpeed*(X-PreviousX);
      SpringPosition[0] += SpringMoveSpeed*(Y-PreviousY);
      haSendCommand(dev, "set mySpring pos", SpringPosition[0], SpringPosition[1],
                    SpringPosition[2], response);
      printf("set mySpring pos [%g,%g,%g] ==> %s\n", SpringPosition[0],
             SpringPosition[1], SpringPosition[2], response);
   }
   else if (RightButtonDown)
   {
      SpringPosition[1] += SpringMoveSpeed*(X-PreviousX);
      SpringPosition[2] += -SpringMoveSpeed*(Y-PreviousY);
      haSendCommand(dev, "set mySpring pos", SpringPosition[0], SpringPosition[1],
                    SpringPosition[2], response);
      printf("set mySpring pos [%g,%g,%g] ==> %s\n", SpringPosition[0],
             SpringPosition[1], SpringPosition[2], response);
   }

   PreviousX = X;
   PreviousY = Y;

}


//---------------------------------------------------------------------
//                            M A I N
//
// 09-Position-Control-Mouse-Slave Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      haSendCommand(dev, "set inertia", inertia, response);
      printf("set inertia ==> %s\n", response);

      // Create a spring effect to bring HM to start position
      if ( haSendCommand(dev, "create spring mySpring", response) ) {
         printf ( "--- ERROR: Could not send command create spring mySpring\n" );
         getchar();
         exit(-1);
      }

      printf("create spring mySpring ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand(dev, "set mySpring pos", SpringPosition[PosX],
                       SpringPosition[PosY], SpringPosition[PosZ], response);
```

```
         printf("set mySpring pos [%g,%g,%g] ==> %s\n", SpringPosition[PosX],
                SpringPosition[PosY], SpringPosition[PosZ], response);

         haSendCommand(dev, "set mySpring stiffness 500.0", response);
         printf("set mySpring stiffness 500.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring dampfactor 1.0", response);
         printf("set mySpring dampfactor 1.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring maxforce 15.0", response);
         printf("set mySpring maxforce 15.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring direction [0.0,0.0,0.0]", response);
         printf("set mySpring direction [0.0,0.0,0.0] ==> %s\n", response);

         haSendCommand(dev, "set mySpring deadband 0.0", response);
         printf("set mySpring deadband 0.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring enable", response);
         printf("set mySpring enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example09, Position Control -
                         Mouse-Slave");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMouseFunc (Mouse);
      glutMotionFunc (Motion);
      glutMainLoop();
   }
   return 0;
}
```

Example 9 - Description

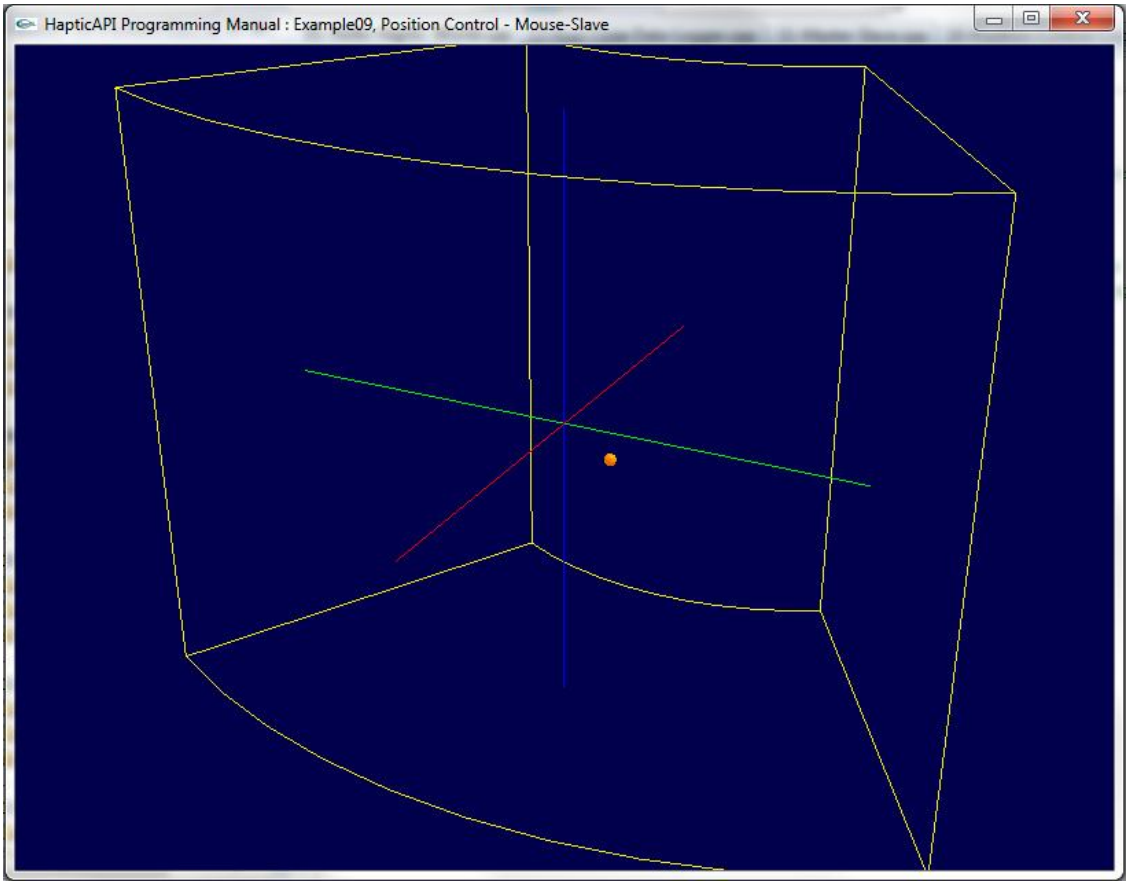


Figure 11-1 : Example 9 Screen Shot

In this example we will use two different effects: the **spring** effect and the **damper** effect. You will find all the necessary includes and variables at the beginning of the code listing as usual.

The OpenGL material variables and functions should be straight forward by now. Following is some explanation about new functions or new functionality.

Mouse() function

This function is a glut callback function being called by the OpenGL framework whenever a mouse button is pushed or released. You can read all the details of this function in an OpenGL book. As this function is only called upon pushing or releasing a mouse button, we use it to store the state of the left and right mouse button in the *RightButtonDown* and *LeftButtonDown* variables respectively. We also store the mouse position at which the pushing or releasing of a mouse button occurred.

Motion() function

This function is a glut callback function being called by the OpenGL framework when the mouse moves while one of the mouse buttons is currently pushed. The parameters supplied with this function do not provide any information about which of the buttons are currently pushed we use our *RightButtonDown* and *LeftButtonDown* variables for this purpose.

The main() function

This function creates and enables a damper effect, next to the spring effect. The reason for this damper effect will be explained later.
The spring effect is created at the origin of the HapticMASTER's workspace. We will move this spring's position depending on the mouse movements and the mouse button currently pushed.
In the *void Motion(int X, int Y)* function the actual changing of the spring's position takes place. If the left mouse button is currently pushed, the horizontal mouse movements change the position of the End Effector along its Y-Axis and the vertical mouse movements change the End Effector's position along the X-Axis. Whenever the right mouse button is pushed, the horizontal mouse movements change the End Effector's position along the Y-Axis and the vertical mouse movements change the position along the Z-Axis.

When you run the application, you will notice that the End Effector follows the relocation of the spring quickly and accurately. This is accomplished by setting the right properties for the spring and damper effects used in this scene. The spring stiffness must be chosen relatively high so that it has a strong pulling effect on the End Effector when it is moved. The damper coefficients are set so that the movements are just critically damped. See the line *double DamperCoeff[3] = {63.20, 63.20, 63.20};* If the damping coefficients are set to lower values, the End Effector following the spring will react with some overshooting when finding its new position. If the damping coefficients are set to higher values, the End Effector will follow the spring with a slower speed. You can try this by changing the spring or damper effect properties and recompiling and running the example.

Be very careful with changing the described parameters for this spring effect because when a program starts the spring effect will be enabled, and depending on the HapticMASTER's End Effector position right before the start, it can make very swift and powerful movements.

12 Position Control – Orbiting in a circle

This chapter shows you how you can control the position of the HapticMASTER End Effector from your application program.
We have created an example which demonstrates how the End Effector can follow an orbit in a form of a circle.

We use some physics to guide the End Effector into a circle orbit.
We define a spring that will bring the End Effector to the horizontal plane where the Z axis is 0.0. The orbit circle will be found on this plane.

There are two forces driving the End Effector into the orbit: The tangential force, which is tangent to the circle and gives the End Effector the movement direction on the circle, and the centripetal force, which always points to the center of the circle and keeps the End Effector in orbit. The user will be able to change the radius of the circle (with the '-' and '+' keys), and the tangential speed of the End Effector (with the '/' and '*' keys). However, to prevent things getting out of hand (dangerous speeds and forces), we limit the speed and the radius by letting the first influence the second and the other way around. If you reduce the circle's radius with too much tangential speed, the speed will be reduced automatically. If you increase speed with a very small radius, the radius can increase as well to prevent the HapticMASTER from generating too swift movements.

For your own safety, take extra care while experimenting with the HapticMASTER's example programs, and in particular this example. Always be ready to push the red 'emergency stop' button should anything go wrong.

Example 10 - Source Code

Example 10: Position Control – Circle orbit

```
//---------------------------------------------------------------------
// 1 0   -   P O S I T I O N   C O N T R O L   -   C I R C L E   O R B I T
//
// This example demonstrates how to control the position of the Haptic
// Master. It tries to orbit in a circle
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "HapticMasterOpenGl.h"
#include "glut.h"

#include <math.h>

#define IPADDRESS "10.30.203.12"

#define PosX 0
#define PosY 1
#define PosZ 2

long dev = 0;
char response[100];

double currentPosition[3];
double dampingCoef[3] = {30.0,30.0,30.0};

double radius = 0.15;
double inertia = 25.0;
double desiredSpeed = 0.08;
double maxCentriForce = 4.0;

double startPosition[3] = {radius, 0.0, 0.0};
Vector3d fCentripetal;
Vector3d fTangential;
Vector3d measSpeed;

bool startPositionReached = false;
bool springTurnedOff = false;

//---------------------------------------------------------------------
// O P E N G L   M A T E R I A L S
//---------------------------------------------------------------------
// EndEffector OpenGL Material Parameters.
GLfloat EndEffectorAmbient[] = {0.91, 0.44, 0.00, 1.00};
GLfloat EndEffectorDiffuse[] = {0.90, 0.38, 0.00, 1.00};

// General OpenGL Material Parameters
GLfloat Specular[] = {1.00, 1.00, 1.00, 1.00};
GLfloat Emissive[] = {0.00, 0.00, 0.00, 1.00};
GLfloat Shininess = {128.00};

// General OpenGL Material Parameters
GLfloat ObjectAmbient[] = {0.00, 0.66, 0.60, 0.50};
GLfloat ObjectDiffuse[] = {0.00, 0.80, 0.67, 0.50};
```

```
//---------------------------------------------------------------------
//             E N D   E F F E C T O R   M A T E R I A L
//
// EndEffectorMaterial() Sets The Current OpenGl Material Paremeters.
// Call This Function Prior To Drawing The EndEffector.
//---------------------------------------------------------------------
void EndEffectorMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, EndEffectorAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, EndEffectorDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                  O B J E C T   M A T E R I A L
//
// ObjectMaterial() Sets The Current OpenGl Material Parameters.
// Call This Function Prior To Drawing Any Of The Objects.
//---------------------------------------------------------------------
void ObjectMaterial()
{
   glMaterialfv(GL_FRONT, GL_AMBIENT, ObjectAmbient);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, ObjectDiffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, Specular);
   glMaterialfv(GL_FRONT, GL_EMISSION, Emissive);
   glMaterialf(GL_FRONT, GL_SHININESS, Shininess);
}

//---------------------------------------------------------------------
//                D R A W   E N D   E F F E C T O R
//
// This Function Is Called To Draw The Graphic Equivalent Of
// The EndEffector In OpenGl.
// The EndEffector Is Drawn At The Current Position
//---------------------------------------------------------------------
void DrawEndEffector(void)
{
   EndEffectorMaterial();
   glPushMatrix();
   glTranslatef(currentPosition[PosX], currentPosition[PosY], currentPosition[PosZ]);
   glutSolidSphere(0.005, 20, 20);
   glPopMatrix();
}

//---------------------------------------------------------------------
//                          D R A W   T O R U S
//
// This function draws the torus as the path for the end effector
//---------------------------------------------------------------------
void DrawTorus(void)
{
   ObjectMaterial();
   glPushMatrix();
   glutSolidTorus(0.001, radius, 60, 60);
   glPopMatrix();
}
```

```
//---------------------------------------------------------------------
//                        D R A W   V E C T O R S
//
// This function draws the torus as the path for the end effector
//---------------------------------------------------------------------
void DrawVectors(void)
{
   XAxisMaterial();
   glBegin(GL_LINES);
      glVertex3f(currentPosition[0], currentPosition[1], currentPosition[2]);
      glVertex3f(currentPosition[0] + fTangential[0] / 40.0,
                 currentPosition[1] + fTangential[1] / 40.0,
                 currentPosition[2] + fTangential[2] / 40.0);
   glEnd();

   YAxisMaterial();
   glBegin(GL_LINES);
      glVertex3f(currentPosition[0], currentPosition[1], currentPosition[2]);
      glVertex3f(currentPosition[0] + fCentripetal[0] / 40.0,
                 currentPosition[1] + fCentripetal[1] / 40.0,
                 currentPosition[2] + fCentripetal[2] / 40.0);
   glEnd();
}


//---------------------------------------------------------------------
//                        I N I T   O P E N   G L
//
// This Function Initializes the OpenGl Graphics Engine
//---------------------------------------------------------------------
void InitOpenGl (void)
{

   glShadeModel(GL_SMOOTH);

   glLoadIdentity();

   GLfloat GrayLight[] = {0.75, 0.75, 0.75, 1.0};
   GLfloat LightPosition[] = {1.0, 2.0, 1.0, 0.0};
   GLfloat LightDirection[] = {0.0, 0.0, -1.0, 0.0};

   glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
   glLightfv(GL_LIGHT0, GL_AMBIENT, GrayLight);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, GrayLight);
   glLightfv(GL_LIGHT0, GL_SPECULAR, GrayLight);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   glEnable (GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

   glClearColor(0.0, 0.0, 0.3, 0.0);
}

//---------------------------------------------------------------------
//                          D I S P L A Y
//
// This Function Is Called By OpenGL To Redraw The Scene
// Here's Where You Put The EndEffector And Block Drawing FuntionCalls
//---------------------------------------------------------------------
void Display (void)
{
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix ();

   // define eyepoint in such a way that
   // drawing can be done as in lab-frame rather than sgi-frame
   // (so X towards user, Z is up)
   gluLookAt (1.0, 0.5, 0.5, 0.0, 0.0, -0.03, 0.0, 0.0, 1.0);
```

```
   glutPostRedisplay();

   DrawAxes();
   DrawWorkspace(dev, 3);

   // Get The Current EndEffector Position From THe HapticMASTER
   haSendCommand( dev, "get modelpos", response );
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelpos ==> %s", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, currentPosition[PosX], currentPosition[PosY],
                     currentPosition[PosZ] );
   }

   // find radial vector on circle
   Vector3d circleCenter( 0.0, 0.0, 0.0);
   Vector3d currentRadial( currentPosition[PosX] - circleCenter.x,
                           currentPosition[PosY] - circleCenter.y,
                           currentPosition[PosZ] - circleCenter.z);
   currentRadial.z = 0.0;
   double currentRadialLength = currentRadial.length();
   currentRadial.normalize();

   // find tangential drive force
   Vector3d zAxis( 0.0, 0.0, 1.0);
   fTangential = desiredSpeed * dampingCoef[0] * crossProduct( zAxis, currentRadial );

   haSendCommand (dev, "get modelvel", response);
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("get modelvel ==> %s\n", response);
      getchar();
      exit(-1);
   }
   else {
      ParseFloatVec( response, measSpeed[PosX], measSpeed[PosY], measSpeed[PosZ] );
   }

   // find centripetal force
   if ( radius > 0.0 )  {
      Vector3d tangentialSpeed( measSpeed.x, measSpeed.y, 0.0);
      tangentialSpeed = tangentialSpeed - dotProduct( tangentialSpeed, currentRadial) *
                        currentRadial;
      fCentripetal = -inertia * tangentialSpeed * tangentialSpeed / radius *
                     currentRadial;
   }

   // sum to total driving force
   Vector3d fDrive = fTangential + fCentripetal;

   char tempString[100] = "";
   sprintf(tempString, "set myDrivingForce force [%g,%g,%g]", fDrive.x, fDrive.y,
           fDrive.z);
   haSendCommand(dev, tempString, response);
   if ( strstr ( response, "--- ERROR:" ) ) {
      printf("set myDrivingForce force ==> %s", response);
      getchar();
      exit(-1);
   }

   DrawVectors();
   DrawEndEffector();
   DrawTorus();
   glPopMatrix ();
   glutSwapBuffers();
}
```

```
//---------------------------------------------------------------------
//                          R E S H A P E
//
// The Function Is Called By OpenGL Whenever The Window Is Resized
//---------------------------------------------------------------------
void Reshape(int iWidth, int iHeight)
{
    glViewport (0, 0, (GLsizei)iWidth, (GLsizei)iHeight);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    float fAspect = (float)iWidth/iHeight;
    gluPerspective (30.0, fAspect, 0.05, 20.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
}

//---------------------------------------------------------------------
//                          K E Y B O A R D
//
// This Function Is Called By OpenGl WhenEver A Key Was Hit
//---------------------------------------------------------------------
void Keyboard(unsigned char ucKey, int iX, int iY)
{
   switch (ucKey)
   {
      case 27:
         haSendCommand(dev, "remove all", response);
         printf("remove all ==> %s\n", response);

         haSendCommand(dev, "set state stop", response);
         printf("set state stop ==> %s\n", response);

         exit(0);
         break;

      case '+':
         if (radius < 0.18) {
            radius += 0.02;
         }
         break;

      case '-':
         if (radius > 0.04) {
            radius -= 0.02;
            if ( inertia * desiredSpeed * desiredSpeed / radius > maxCentriForce ) {
               if (desiredSpeed < 0) {
                  desiredSpeed = - sqrt(maxCentriForce * radius / inertia);
               }
               else {
                  desiredSpeed = sqrt(maxCentriForce * radius / inertia);
               }
            }
         }
         break;

      case '/':
         if (desiredSpeed > -0.5) {
            desiredSpeed -= 0.01;
            if ( inertia * desiredSpeed * desiredSpeed / radius > maxCentriForce ) {
               radius = inertia * desiredSpeed * desiredSpeed / maxCentriForce;
               if (radius > 0.18) {
                  desiredSpeed += 0.01;
                  radius = inertia * desiredSpeed * desiredSpeed / maxCentriForce;
               }
            }
         }
         break;

      case '*':
```

```
         if (desiredSpeed < 0.5) {
            desiredSpeed += 0.01;
            if ( inertia * desiredSpeed * desiredSpeed / radius > maxCentriForce ) {
               radius = inertia * desiredSpeed * desiredSpeed / maxCentriForce;
               if (radius > 0.18) {
                  desiredSpeed -= 0.01;
                  radius = inertia * desiredSpeed * desiredSpeed / maxCentriForce;
               }
            }
         }
         break;
    }
}


//---------------------------------------------------------------------
//                            M A I N
//
// 10-Position-Control-Circle-Orbit Main Function
//---------------------------------------------------------------------
int main(int argc, char** argv)
{
   // Call The Initialize HapticMASTER Function
   dev = haDeviceOpen( IPADDRESS );

   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      return HARET_ERROR;
   }
   else {
      InitializeDevice( dev );

      char tempString[50] = "";
      sprintf(tempString, "set inertia %g", inertia);

      haSendCommand(dev, tempString, response);
      printf("set inertia ==> %s\n", response);

      // Create a damper effect
      if ( haSendCommand(dev, "create damper myDamper", response) ) {
         printf ( "--- ERROR: Could not send command create damper myDapmer\n" );
         getchar();
         exit(-1);
      }

      printf("create damper myDamper ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         char tempString [100] = "";
         sprintf(tempString, "set myDamper dampcoef [%g,%g,%g]", dampingCoef[0],
                 dampingCoef[1], dampingCoef[2]);
         haSendCommand(dev, tempString, response);
         printf("set myDamper dampcoef ==> %s\n", response);

         haSendCommand(dev, "set myDamper enable", response);
         printf("set myDamper enable ==> %s\n", response);
      }

      // Create a spring effect to bring HM to start position
      if ( haSendCommand(dev, "create spring mySpring", response) ) {
         printf ( "--- ERROR: Could not send command create spring mySrping\n" );
         getchar();
         exit(-1);
      }

      printf("create spring mySpring ==> %s\n", response);
```

```
      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         char tempString[100] = "";
         sprintf(tempString, "set mySpring pos [%g,%g,%g]", startPosition[PosX],
                 startPosition[PosY], startPosition[PosZ]);
         haSendCommand(dev, tempString, response);
         printf("set mySpring pos [%g,%g,%g] ==> %s\n", startPosition[PosX],
                startPosition[PosY], startPosition[PosZ], response);

         haSendCommand(dev, "set mySpring stiffness 50.0", response);
         printf("set mySpring stiffness 50.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring direction [0.0,0.0,1.0]", response);
         printf("set mySpring direction [0.0,0.0,1.0] ==> %s\n", response);

         haSendCommand(dev, "set mySpring dampfactor 1.0", response);
         printf("set mySpring dampfactor 1.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring maxforce 25.0", response);
         printf("set mySpring maxforce 25.0 ==> %s\n", response);

         haSendCommand(dev, "set mySpring enable", response);
         printf("set mySpring enable: %s\n", response);
      }

      // Create a bias force Effect and supply it with parameters
      if ( haSendCommand(dev, "create biasforce myDrivingForce", response) ) {
         printf ( "--- ERROR: Could not send command create biasforce myDrivingForce\n"
                );
         getchar();
         exit(-1);
      }
      printf("create biasforce myDrivingForce ==> %s\n", response);

      if ( strstr ( response, "--- ERROR:" ) ) {
         getchar();
         exit(-1);
      }
      else {
         haSendCommand(dev, "set myDrivingForce force [0,0,0]", response);
         printf("set myDrivingForce force ==> %s\n", response);

         haSendCommand(dev, "set myDrivingForce enable", response);
         printf("set myDrivingForce enable ==> %s\n", response);
      }

      // OpenGL Initialization Calls
      glutInit(&argc, argv);
      glutInitDisplayMode (GLUT_DOUBLE| GLUT_RGB | GLUT_DEPTH);
      glutInitWindowSize (800, 600);

      // Create The OpenGlWindow
      glutCreateWindow ("HapticAPI Programming Manual : Example10: Position Control -
                         Circle Orbit");

      InitOpenGl();

      // More OpenGL Initialization Calls
      glutReshapeFunc (Reshape);
      glutDisplayFunc(Display);
      glutKeyboardFunc (Keyboard);
      glutMainLoop();
   }
   return 0;
}
```

Example 10 - Description

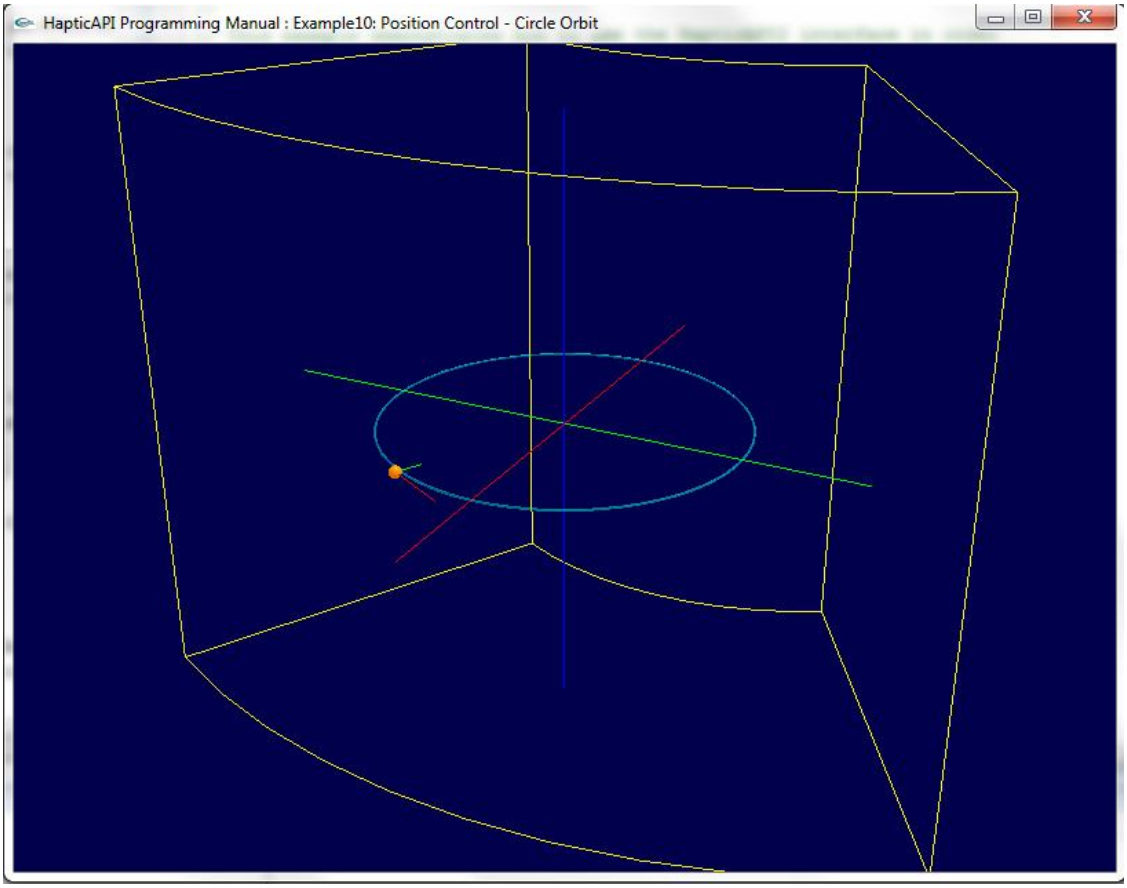


Figure 12-1 : Example 10 Screen Shot

The End Effector starts in the position where the HapticMASTER is found when the program starts. The tangential force is set and the centripetal force is constantly calculated to start intercepting the orbit. You will see that after a couple of seconds the End Effector follows the designated circle perfectly. You can then change the radius of the circle ('-' and '+' keys) and you will see the End Effector automatically intercepts the new circle again. If you change the desired tangential speed ('/' and '*' keys), you see that the tangential force grows or shrinks. When you hit the '/' key many times, you will notice the direction of the orbit will flip, since you set the desired tangential speed to be negative. Notice that the desired tangential speed and desired radius do influence each other in our program. Otherwise you could have generated dangerous speeds with a small radius, which would have ended up in "loosing control" of the HapticMASTER. **For safety reasons, please do not make rapid changes in radius or tangential speed. Give the HapticMASTER sufficient time to react on the change.**

During runtime, the HapticMASTER is in the FORCE state, meaning you can also move the End Effector by applying force on the force sensor. This will result in moving the End Effector out of orbit. When you let the HapticMASTER to move freely, it will intercept the orbit again.
Following are some explanations of the source code and the functions.

Glabal variables

radius	Holds the current radius of the circle
inertia	The mass of the End Effector
deisredSpeed	The desired tangential speed in the orbit
maxXnetriForce	Maximum centripetal force during orbit This determines when the radius changes influence the speed and vice versa
fCentripetal	A Vector3d object to hold the centripetal force
fTangential	A Vector3d object to hold the tangential force
measSpeed	A Vector3d object to hold the measured speed queried from the HapticMASTER

DrawTorus()

Draws the circle of the orbit we want to follow. The torus's thickness is 0.001 and the radius is determined by the *radius* variable.

DrawVectors()

Draws lines representing the driving force (fTangential) and the centripetal force (fCentripetal)

Display()

The function first sets the perspective view and draws the HapticMASTER's axes and workspace. It then queries the End Effector's position from the HapticMASTER into the *currentPosition[]* vector.

What follows is some physics formulas to set the tangential force and the centripetal force in such a way that the End Effector constantly intercepts the orbit and is driven with a desired tangential speed.

First, we find the **radial vector** (vector from the End Effector's current position to the center of the circle). The Vector3d object *currentRadial* holds this vector, which is the result of subtracting the current position coordinates from these of the circle's center. The Z coordinate of *currentRadial* is set to 0.0. The spring effect (created in the main() function) takes care of bringing the End Effector to the plane in which Z is 0.0.
The double *currentRadialLength* keeps the **length of the current radial vector**. The radial vector itself is then normalized, to get a vector with length 1.0.

Now we calculate the **tangential force** *fTangential*. It is the **desired tangential speed** *desiredSpeed* (given by the user in m/sec) multiplied by the **damping coefficient** on the X axis *dampingCoef[0]* and then multiplied by the **cross product** of the two vectors *zAxis* (which is simply a vector pointing in the direction of the Z axis) and *currentRadial*, which we calculated earlier in the code.

The cross product of two vectors is obtained by calling the *crossProduct()* function with two Vector3d objects as parameters.

The **End Effector's velocity** is then queried from the HapticMASTER and copied to the *measSpeed[]* vector. This is done by sending the command "get measvel" to the HapticMASTER and then parsing the response string using the *ParseFloatVec()* function.

Now we have to calculate the **centripetal force** *fCentripetal*. Since we are dividing by the radius, we have to make sure the radius is greater than 0.0. First we find the **current tangential speed** *tangentialSpeed*. This will be the X and Y coordinates of the *measSpeed* vector that we queried from the HapticMASTER (we don't take the Z axis into consideration – the spring takes care of leveling the End Effector). Consequently, we subtract from the tangential speed vector the multiplication of the current radial with the dot product of the tangential speed and the current radial. The dot product of two vectors is obtained by calling the *dotProduct()* function supplying two Vector3d objects as parameters. The formula for obtaining the new tangential speed is as follows:

*tangentialSpeed -= (tangentialSpeed · currentRadial) * currentRadial*

The **centripetal force** *fCentripetal* is then

*fCentripetal = –inertia * tangentialSpeed * tangentialSpeed / radius * currentRadial*

Now we combine the tangential force and the centripetal force into the **driving force** *fDrive* by adding the two vectors (Since we use the Vector3d objects, we can simply insert the '+' operator between the two vectors).

Finally, the **force *fDrive* is sent to the HapticMASTER** as the current bias force in order to actually let the End Effector intercept the orbit. This is done by sending the command "set myDrivingForce force" and supplying the *haSendCommand()* function with the 3 coordinates of the force.

Last actions of the *Display()* function are:

- Drawing the vectors of the tangential force and the centripetal force
- Drawing the End Effector
- Drawing the torus to display the orbit

The Keyboard() function

Here the reaction on pressing one of the four keys '-', '+', '/' and '*' is implemented. For safety reasons, when the user decreases the desired radius (with the '-' key), we need to check if the maximum allowed centripetal force is exceeded. If it is exceeded, we decrease the desired tangential speed to fit this

limit. When the user increases the desired tangential speed (with the '*' key), we perform the same check. If the maximum allowed centripetal force is exceeded, we increase the radius to fit this limit. Pressing the '/' key often enough generates a negative tangential speed, resulting in the End Effector following the orbit in the reverse direction. Since this negative speed can also become too large, we need to perform the maximum centripetal force check here as well.

To **perform the check**, we compare the maximum centripetal force with
*inertia * desiredSpeed * desiredSpeed / radius*

To **fit the desired speed** while changing the radius, we calculate:
*desiredSpeed = sqrt(maxCentriForce * radius / inertia)*
(*sqrt()* calculates the square root)

To **fit the desired radius** while changing the speed, we calculate:
*radius = inertia * desiredSpeed * desiredSpeed / maxCentriForce*

When the '+' key is pressed (increase desired radius):
If (*radius* < 18 cm)
Add 2 cm to the *radius*

When the '-' key is pressed (decrease desired radius):
If (*radius* > 4 cm)
Subtract 2 cm from the *radius*
If (*maxCentriForce* exceeded with this *radius)*
Decrease *desiredSpeed* (in the correct direction)

When the '/' key is pressed (decrease desired speed):
If (*desiredSpeed* > *-0.5 m/sec*)
Subtract 0.01 m/sec from the *desiredSpeed*
If (*maxCentriForce* exceeded with this *desiredSpeed)*
Increase *radius*
If (*radius* > 18 cm)
Add 0.01 m/sec to *desiredSpeed*
Recalculate *radius (*aborted speed change)

When the '*' key is pressed (increase desired speed):
If (*desiredSpeed* < *0.5 m/sec*)
Add 0.01 m/sec to the *desiredSpeed*
If (*maxCentriForce* exceeded with this *desiredSpeed)*
Increase *radius*
If (*radius* > 18 cm)
Subtract 0.01 m/sec from *desiredSpeed*
Recalculate *radius (*aborted speed change)

The main() function

- Sets the **inertia**:
  - "set inertia", *inertia*

- Creates and sets **damper effect**:
  - "create damper myDamper"
  - "set myDamper dampcoef", *dampingCoef[0], dampingCoef[1], dampingCoef[2]*
  - "set myDamper enable"

  The damper takes care of smoothing the movement.

- Creates and sets vertical **spring effect**:
  - "create spring mySpring"
  - "set mySpring pos", *startPosition[0], startPosition[1], startPosition[2],*
  - "set mySpring stiffness 50.0"
  - "set mySpring direction [0.0,0.0,1.0]"
  - "set mySpring dampffactor 0.0"
  - "set mySpring maxforce 25.0"
  - "set mySpring enable"

  The spring works only on the Z axis and takes care of bringing the End Effector to the Z = 0.0 plane. Notice the spring itself has no damping – the damping comes from the damper effect which is effective in the complete workspace.

- Creates and sets **bias force effect**:
  - "create biasforce myDrivingForce"
  - "set myDrivingForce force [0,0,0]"
  - "set myDrivingForce enable"

  The bias force is created with force [0,0,0], meaning it does not influence the End Effector in the beginning. Each time the *Display()* function is executed, the force is recalculated and thus effects the End Effector, moving it into orbit.

13 Master Slave operation

The example in this chapter is one of the most interesting in this manual.
In this example a Master-Slave setup is created between two HapticMASTERs.

The only link between the two HapticMASTER devices is created by software, an application running on a client PC with a network connection to both devices.

The Master-Slave operation is realized by applying the measured forces to each HapticMASTER's own End Effector but also forwarding the force inputs to the other HapticMASTER. HapticMASTER #1 sends the force inputs to itself but also to HapticMASTER #2. HapticMASTER #2 sends the force inputs to itself but also to HapticMASTER #1. A (virtual) coupling spring pulls both HapticMASTERs to the same position within their respective working spaces. This is done by calculating the differences in position and velocity between the two HapticMASTERs.

As a matter of fact, this is a Master-Master operation, since both HapticMASTERs can move its partner. If you want a Master-Slave operation (only one HapticMASTER will give the force inputs, the other one is slaved to the master and cannot generate force inputs), you need to make some minor changes in the code. These changes are out of the scope of this example.

Again, this is one of the most interesting examples, as it shows how movements and forces can be copied between two haptic devices over a network. This example has no visual equivalent so the code is quite short.

Example 11 - Source Code

Example 11 : Master-Slave Operation

```
//---------------------------------------------------------------------
//                11   -   M A S T E R   S L A V E
//
// This example demonstrates how two HapticMASTER devices can control
// each other and follow each other's position in the "master-slave"
// setting. Moving one HapticMASTER results in moving the other one as
// well and the other way around.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMaster.h"
#include "Vector3d.h"

#include <math.h>

#ifdef WIN32
   #include <process.h>
   #include <conio.h>
   #include <io.h>
#else
    #include <pthread.h>
   #include <sys/time.h> /* struct timeval, select() */
   /* ICANON, ECHO, TCSANOW, struct termios */
   #include <termios.h> /* tcgetattr(), tcsetattr() */
   #include <stdlib.h> /* atexit(), exit() */
   #include <unistd.h> /* read() */
   #include <stdio.h> /* printf() */

   static struct termios g_old_kbd_mode;
   /*****************************************************************************
   *****************************************************************************/
   static void cooked(void)
   {
      tcsetattr(0, TCSANOW, &g_old_kbd_mode);
   }
   /*****************************************************************************
   *****************************************************************************/
   static void raw(void)
   {
      static char init;
   /**/
      struct termios new_kbd_mode;

      if(init)
         return;
   /* get current state of console */
      tcgetattr(0, &g_old_kbd_mode);
   /* put keyboard (stdin, actually) in raw, unbuffered mode */
      memcpy(&new_kbd_mode, &g_old_kbd_mode, sizeof(struct termios));
      new_kbd_mode.c_lflag &= ~(ICANON | ECHO);
      new_kbd_mode.c_cc[VTIME] = 0;
      new_kbd_mode.c_cc[VMIN] = 1;
      tcsetattr(0, TCSANOW, &new_kbd_mode);
   /* when we exit, go back to normal, "cooked" mode */
      atexit(cooked);

      init = 1;
   }
```

```
   /*****************************************************************************
   *****************************************************************************/
   static int _kbhit(void)
   {
      struct timeval timeout;
      fd_set read_handles;
      int status;

      raw();
   /* check stdin (fd 0) for activity */
      FD_ZERO(&read_handles);
      FD_SET(0, &read_handles);
      timeout.tv_sec = timeout.tv_usec = 0;
      status = select(0 + 1, &read_handles, NULL, NULL, &timeout);
      if(status < 0)
      {
         printf("select() failed in kbhit()\n");
         exit(1);
      }
      return status;
   }
   /*****************************************************************************
   *****************************************************************************/
   static int _getch(void)
   {
      unsigned char temp;

      raw();
   /* stdin = fd 0 */
      if(read(0, &temp, 1) != 1)
         return 0;
      return temp;
   }
#endif
#include <stdio.h>

#define IPADDRESS_1 "10.30.203.12"
#define IPADDRESS_2 "10.30.203.10"

#define PosX 0
#define PosY 1
#define PosZ 2

// Set the variables to their default values
double dInertia = 3.5;
double dSpringConst = 5.0;
double dDampFact = 10.0;

bool bContinue = true;

// HapticMASTER handles
long dev1 = 0;
long dev2 = 0;

char response[100];

int gUpdateRateCounter;

// 3D vectors for measured values, calculation and commands
Vector3d dSpringForce;
Vector3d dDiffPos;
Vector3d dDiffVel;
Vector3d measuredPos1;
Vector3d measuredPos2;
Vector3d measuredVel1;
Vector3d measuredVel2;
Vector3d measuredForce1;
Vector3d measuredForce2;
Vector3d commandedForce1;
Vector3d commandedForce2;
```

```
//---------------------------------------------------------------------
//                          K B   M O N
//
// This function monitors the keyboard for any keys being hit.
// It runs in a thread for as long as the boolean parameter
// given with the call to this function is true.
// When this boolean variable is set to false, on application termination,
// this tread will end itself
// In this app we use the bContinue boolean var the signal when threads
// are to end
//---------------------------------------------------------------------
#ifdef WIN32
   void KbMon(void *pParam)
#else
   void* KbMon(void *pParam)
#endif
{
   while ( *((bool*)pParam) )
   {
#ifdef WIN32
      Sleep(1000);
#else
     sleep(1000);
#endif

      printf( "Loops: %i\n", gUpdateRateCounter );
      gUpdateRateCounter = 0;

      if ( _kbhit() )
      {
         switch(_getch())
         {
            case 27 :
               // Escape will end the application
               *((bool*)pParam) = false;
               break;
            default :
               // any other key will also end the application
               *((bool*)pParam) = false;
               break;
         }
      }
   }

   // At this point the boolean var is set to false, so the thread loop
   // is exit. Terminate the thread
#ifdef WIN32
   _endthread();
#else
   pthread_exit(0);
#endif
}
```

```
//---------------------------------------------------------------------
//                            M A I N
//
// 11-Master-Slave Main Function
//---------------------------------------------------------------------
int main(void)
{
#ifndef WIN32
   pthread_t tid;
   int err;
#endif

   // First create two instances of the HapticMaster (wrapper) class
   dev1 = haDeviceOpen( IPADDRESS_1 );
   dev2 = haDeviceOpen( IPADDRESS_2 );

   // Problems with connecting to HapticMASTER #1
   if( dev1 == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to 1st device: %s\n", IPADDRESS_1 );
      getchar();
      return HARET_ERROR;
   }

   // Problems with connecting to HapticMASTER #2
   else if ( dev2 == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to 2nd device: %s\n", IPADDRESS_2 );
      getchar();
      return HARET_ERROR;
   }

   // Connection to both HapticMASTER succeeded
   else {

      // Initialize both HapticMASTERs
      InitializeDevice( dev1 );
      InitializeDevice( dev2 );

      haSendCommand(dev1, "set inertia", dInertia, response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev1: set inertia ==> %s\n", response);
         getchar();
         exit(-1);
      }
      haSendCommand(dev2, "set inertia", dInertia, response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev2: set inertia ==> %s\n", response);
         getchar();
         exit(-1);
      }

      // Create the bias force for both HapticMASTERs.
      // Each bias force will be set to fit the counterpart's
      // measured force
      haSendCommand(dev1, "create biasforce myForce", response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev1: create biasforce myForce ==> %s\n", response);
         getchar();
         exit(-1);
      }
      haSendCommand(dev2, "create biasforce myForce", response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev2: create biasforce myForce ==> %s\n", response);
         getchar();
         exit(-1);
      }
```

```
      // Set the bias forces to be 0 on initialization
      haSendCommand(dev1, "set myForce force [0,0,0]", response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev1: set myForce [0,0,0] ==> %s\n", response);
         getchar();
         exit(-1);
      }
      haSendCommand(dev2, "set myForce force [0,0,0]", response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev2: set myForce [0,0,0] ==> %s\n", response);
         getchar();
         exit(-1);
      }

      // Enable the bias forces
      haSendCommand(dev1, "set myForce enable", response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev1: set myForce enable ==> %s\n", response);
         getchar();
         exit(-1);
      }
      haSendCommand(dev2, "set myForce enable", response);
      if ( strstr ( response, "--- ERROR:" ) ) {
         printf("dev2: set myForce enable ==> %s\n", response);
         getchar();
         exit(-1);
      }

      // Start the keyboard monitoring thread
#ifdef WIN32
      _beginthread(KbMon, 0, &bContinue);
#else
     err = pthread_create(&tid, NULL, &KbMon, &bContinue);
#endif

      // While ESC not pressed, go on with the applications
      while ( bContinue )
      {
         gUpdateRateCounter++;

         // Calculate the x, y and z differences in HapticMASTER positions
         haSendCommand(dev1, "get modelpos", response);
         ParseFloatVec(response, measuredPos1.x, measuredPos1.y, measuredPos1.z);
         haSendCommand(dev2, "get modelpos", response);
         ParseFloatVec(response, measuredPos2.x, measuredPos2.y, measuredPos2.z);

         dDiffPos = measuredPos2 - measuredPos1;

         // Calculate the x, y and z differences in HapticMASTER velocities
         haSendCommand(dev1, "get modelvel", response);
         ParseFloatVec(response, measuredVel1.x, measuredVel1.y, measuredVel1.z);
         haSendCommand(dev2, "get modelvel", response);
         ParseFloatVec(response, measuredVel2.x, measuredVel2.y, measuredVel2.z);

         dDiffVel = measuredVel2 - measuredVel1;

         // Calculate the x, y and z spring force and damping of the coupling spring
         dSpringForce = dSpringConst*dDiffPos + dDampFact*dDiffVel;

         // The force to sent to each HapticMASTER is the sum of the
         // calculated coupling springs force and the End-Effector force of
         // the other HapticMASTER
         haSendCommand(dev1, "get measforce", response);
         ParseFloatVec(response, measuredForce1.x, measuredForce1.y, measuredForce1.z);
         haSendCommand(dev2, "get measforce", response);
         ParseFloatVec(response, measuredForce2.x, measuredForce2.y, measuredForce2.z);

         commandedForce1 = measuredForce2 + dSpringForce;
         haSendCommand(dev1, "set myForce force", commandedForce1, response);

         commandedForce2 = measuredForce1 - dSpringForce;
```

```
         haSendCommand(dev2, "set myForce force", commandedForce2, response);
      }
   }

   // When the application finishes, do some cleaning up
   printf("Shutting down application...\n");

   if (dev1) {
      haSendCommand(dev1, "set myForce force [0,0,0]", response);
      haSendCommand(dev1, "remove all", response);
      haSendCommand(dev1, "set state stop", response);
      haDeviceClose(dev1);
   }

   if (dev2) {
      haSendCommand(dev2, "set myForce force [0,0,0]", response);
      haSendCommand(dev2, "remove all", response);
      haSendCommand(dev2, "set state stop", response);
      haDeviceClose(dev2);
   }
   return 1;
}
```

Example 11 - Description

In this example, no OpenGL code is found. For each HapticMASTER, a haptic world is created, with a bias force. The program we write simply connects the force readings of the first HapticMASTER with the settings of the bias force of the second HapticMASTER. At the same time, the second HapticMASTER's force inputs are connected to the first HapticMASTER's bias force. The virtual spring is really virtual (it is only calculated and not created in the any of the HapticMASTERs). The spring is used to bring both End Effectors to the same position in the workspace (each HapticMASTER in its own workspace). Once the End Effectors are in the same position, the spring helps compensate for small position / velocity errors.

Ok, let's take a walk through the code.

Global variables

In this example we need to define two IP addresses for the two HapticMASTERs.

dInertia	defines the mass of the End Effector. This will be the same mass for both End Effectors.
dSpringConst	The spring's constant, representing its strength
dDampFact	Damping factor of the spring
bContinue	Boolean that determines whether we stay in the loop or terminate the execution of the program.
dev1	Identifier (handle) of the first HapticMASTER device
dev2	Identifier (handle) of the second HapticMASTER device
response[]	A string to keep the HapticMASTER's response given a specific command.
dSpringForce	A Vector3d object that keeps the current force the virtual spring generates.
dDiffPos	A Vector3d object that keeps the difference between the positions of the two End Effectors.
dDiffVel	A Vector3d object that keeps the difference between the velocities of the two End Effectors.
measuredPos1	A Vector3d object that keeps the measured position of HapticMASTER #1
measuredPos2	A Vector3d object that keeps the measured position of HapticMASTER #2

measuredVel1	A Vector3d object that keeps the measured velocity of HapticMASTER #1
measuredVel2	A Vector3d object that keeps the measured velocity of HapticMASTER #2
measuredForce1	A Vector3d object that keeps the measured force on the force sensor of HapticMASTER #1
measuredForce2	A Vector3d object that keeps the measured force on the force sensor of HapticMASTER #2
commandedForce1	A Vector3d object that keeps the commanded force for HapticMASTER #1. This force is based on the force being measured by the force sensor of HapticMASTER #2
commandedForce2	A Vector3d object that keeps the commanded force for HapticMASTER #2. This force is based on the force being measured by the force sensor of HapticMASTER #1

The KbMon() function

This function checks twice a second if a key was pressed. When a key is pressed on the keyboard, the execution of the program will end, since the variable pointed by the pointer pParam will be set to false. Since the variable *bContinue* is pointed by this pointer, the *bContinue* variable becomes false, which results in leaving the *while* loop in the *main()* function.
The *KbMon()* function runs as a separate thread and therefore executes **parallel** to the *main()* function.

The main() function

The *main()* function tries to open both devices.
The identifier for HapticMASTER #1 is kept in the *dev1* variable.
The identifier for HapticMASTER #2 is kept in the *dev2* variable.

If opening one of the HapticMASTERs failed, an error message is generated and the program terminates. If opening succeeded, we initialize both devices (if searching encoder ends is needed it will automatically take place).
The inertia for both devices is then set. These inertias should be the same for both HapticMASTERs, otherwise the accelerations will be different for the same force input.
Consequently, a bias force effect is created and enabled in both devices. The force is initialized to have no effect on the End Effector (force vector [0,0,0]).
The keyboard monitoring function *KbMon()* is started as a thread and monitors if a key is pressed every 0.5 second.
The main loop is now started, which we quit only if *bContinue* becomes false. Once in the loop, we query the measured positions of both devices. The measured positions are parsed into the *measuredPos1* and *measuredPos2* 3D vectors. The difference between the positions is calculated into the 3D vector *dDiffPos*.

The same is done with the velocities of both devices, only to *measuredVel1, measuredVel2* and *dDiffVel* respectively.
The force that the virtual spring should generate is calculated to compensate for the position and velocity differences:

*dSpringForce = dSpringConst * dDiffPos + dDampFact * dDiffVel*

Now we query the force applied on the force sensor of HapticMASTER #2. This force, compensated by the virtual spring's force (added) is immediately set as the bias force in HapticMASTER #1. Consequently, we query the force applied on the force sensor of HapticMASTER #1, and this force, compensated by the virtual spring's force (subtracted) is set as the bias force in HapticMASTER #2.

When a key is pressed, the *KbMon()* function sets the *bContinue* variable to false, and the while loop is terminated.
We then set, on both devices, the bias force to [0,0,0], remove the bias force effect and set the devices to the **stop** state.

14 Matlab programs

The HapticAPI can be easily used in Matlab programs to communicate with the HapticMASTER. The following three HapticAPI files should be found in the subdirectory where your Matlab program is located:

- haDeviceOpen.m
- haDeviceClose.m
- haDeviceSendString.m

In section 0 the first C++ example program with a graphic variant was introduced, in which we created a haptic world with a single cube. The haptic world was displayed in an OpenGL graphics window, including the haptic cube, the HapticMASTER's workspace and the axes.
We now introduce the Matlab equivalent of this example. We use the HapticAPI library to communicate with the HapticMASTER. Instead of the OpenGL graphic functions, we use the Matlab graphic functions.

Example 12 - Source code

Example 12 : Matlab program

```
% example_12 - Matlab program to communicate with the HapticMASTER device
% ------------------------------------------------------------------
% File    : example_12.m
%
% Purpose : Matlab program to demonstrate the communication with the
%           HapticMASTER device
%
% History:
%  2010-11-08  EH    created
% ------------------------------------------------------------------

% close all, clear all
close all;
clear all;

%------------------------------------------------------------------
% Global Variables
%------------------------------------------------------------------
hapticMasterDeviceHandle = -1;           % Handle to connected device
hapticMasterIPAddress    = '10.30.203.54';% Device IP Address
retVal                   = 0;            % Return value of the haSendCommand
inertia                  = 2.5;          % Mass of the EndEffector
blockStiffness           = 20000;        % Block stiffness
blockSize                = [0.1,0.5,0.03]; % Block dimensions
workspaceR               = [0.0,0.0,0.0]; % Workspace R axis measurements
workspacePhi             = [0.0,0.0,0.0]; % Workspace Phi axis measurements
workspaceZ               = [0.0,0.0,0.0]; % Workspace Z axis measurements
hmCalibrated             = 0;            % Is HM calibrated?
ch                       = ' ';          % Key pressed in figure view

%------------------------------------------------------------------
% Initialization
%------------------------------------------------------------------
% Use getkey.m as a callback function from the current figure
set( gcf, 'keypressfcn', 'LastKeyPressed = getkey;');
LastKeyPressed = 0;

% load HapticAPI2 library
if ( ~libisloaded('HapticAPI2') )
    loadlibrary( 'HapticAPI2' , 'HapticAPI2.h' );
end

disp('Opening device....');
[hapticMasterDeviceHandle, hapticMasterIPAddress] = calllib( 'HapticAPI2',
'haDeviceOpen', hapticMasterIPAddress );

if (hapticMasterDeviceHandle ~= -1)
   disp ( ['Connected to device ' hapticMasterIPAddress ])

   %------------------------------------------------------------------
   % Query workspace dimensions
   %------------------------------------------------------------------
   [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'get workspace_r' );
   if (retVal ~= 0)
       disp (['--- ERROR: ' response]);
   else
       workspaceR = eval(response);
   end

   [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'get workspace_phi' );
   if (retVal ~= 0)
       disp (['--- ERROR: ' response]);
   else
       workspacePhi = eval(response);
   end
```

```
[retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'get workspace_z' );
if (retVal ~= 0)
    disp (['--- ERROR: ' response]);
else
    workspaceZ = eval(response);
end

%-----------------------------------------------------------------
% Check if HapticMASTER is position calibrated
%-----------------------------------------------------------------
[retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'get
                                    position_calibrated' );
if (retVal ~= 0)
    disp (['--- ERROR: ' response]);
end

%-----------------------------------------------------------------
% If not, initialize HpaticMASTER first (search end stops)
%-----------------------------------------------------------------
if ( strcmp(response,'"false"') )

   [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'set state init' );
   if (retVal ~= 0)
       disp (['--- ERROR: ' response]);
   end

   while (~hmCalibrated)
      [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'get state' );
      if (retVal ~= 0)
          disp (['--- ERROR: ' response]);
      end

      if ( strcmp(response,'"stop"') )
         hmCalibrated = 1;
      end
   end
end

%-----------------------------------------------------------------
% Create the haptic world (inertia, block, state force)
%-----------------------------------------------------------------
[retVal, response] = haSendCommand( hapticMasterDeviceHandle, ['set inertia '
                                    num2str(inertia)] );
if (retVal ~= 0)
   disp (['--- ERROR: ' response]);
end

[retVal, response] = haSendCommand( hapticMasterDeviceHandle,
                                    'create block myBlock' );
if (retVal ~= 0)
   disp (['--- ERROR: ' response]);
end

[retVal, response] = haSendCommand( hapticMasterDeviceHandle,
                                  ['set myBlock stiffness ' num2str(blockStiffness)] );
if (retVal ~= 0)
   disp (['--- ERROR: ' response]);
end

[retVal, response] = haSendCommand( hapticMasterDeviceHandle, ['set myBlock size ['
                                    num2str(blockSize(1)) ',' num2str(blockSize(2))
                                    ',' num2str(blockSize(3)) ']'] );
if (retVal ~= 0)
   disp (['--- ERROR: ' response]);
end

[retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'set myBlock enable' );
if (retVal ~= 0)
   disp (['--- ERROR: ' response]);
end
```

```
[retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'set state force' );
if (retVal ~= 0)
    disp (['--- ERROR: ' response]);
else
    disp (response);
end

%---------------------------------------------------------------------
% Global drawing settings
%---------------------------------------------------------------------
scene1;       % set up perspective axes, equal scaling, and a light
axis vis3d    % prevents zooming to fill screen when rotating in 3D
axis off;     % Don't draw axes
set(gca, 'CameraViewAngle', 3.5)   % Camera Field Of View
set(gcf, 'color', [0.0 0.0 0.25]); % Background color dark blue
rotate3d on;

%---------------------------------------------------------------------
% Create a graphic block
%---------------------------------------------------------------------
% Set up block in X to front, Y to right, Z up (Mesh object)
BlockX = [ ...
   0 0 0 0 0
   0 0 0 0 0
   1 1 1 1 1
   1 1 1 1 1 ];
BlockY = [ ...
   0 1 1 0 0
   0 1 1 0 0
   0 1 1 0 0
   0 1 1 0 0 ];
BlockZ = [ ...
   0 0 0 0 0
   0 0 1 1 0
   0 0 1 1 0
   0 0 0 0 0 ];

% Resize and translate (move) the block to match the haptic block's dimensions
BlockX = blockSize(1) * BlockX - blockSize(1)/2;
BlockY = blockSize(2) * BlockY - blockSize(2)/2;
BlockZ = blockSize(3) * BlockZ - blockSize(3)/2;

% Draw the block's surfaces - hBlock will be the handle to that object
% (Do not use surfl calls, they add light objects every time)
hBlock = surf( BlockX, BlockY, BlockZ);

% Set block's properties
surfset1( [0.00 0.66 0.60], hBlock); % surface color
set( hBlock, 'facelighting','flat'); % do not smooth

%---------------------------------------------------------------------
% Draw workspace
%---------------------------------------------------------------------
% 4x Vertical workspace lines
northEast = [(workspaceR(2) + workspaceR(1))*cos(workspacePhi(3)) -
             workspaceR(2),(workspaceR(2) + workspaceR(1))*sin(workspacePhi(3))];
northWest = [(workspaceR(2) + workspaceR(1))*cos(workspacePhi(1)) -
             workspaceR(2),(workspaceR(2) + workspaceR(1))*sin(workspacePhi(1))];
southEast = [(workspaceR(2) + workspaceR(3))*cos(workspacePhi(3)) -
             workspaceR(2),(workspaceR(2) + workspaceR(3))*sin(workspacePhi(3))];
southWest = [(workspaceR(2) + workspaceR(3))*cos(workspacePhi(1)) -
             workspaceR(2),(workspaceR(2) + workspaceR(3))*sin(workspacePhi(1))];
line([northEast(1) northEast(1)],[northEast(2) northEast(2)],[workspaceZ(1)
             workspaceZ(3)],'Color',[1 1 0]);
line([northWest(1) northWest(1)],[northWest(2) northWest(2)],[workspaceZ(1)
             workspaceZ(3)],'Color',[1 1 0]);
line([southEast(1) southEast(1)],[southEast(2) southEast(2)],[workspaceZ(1)
             workspaceZ(3)],'Color',[1 1 0]);
line([southWest(1) southWest(1)],[southWest(2) southWest(2)],[workspaceZ(1)
             workspaceZ(3)],'Color',[1 1 0]);
```

```
% horizontal workpace lines
line([northEast(1) southEast(1)],[northEast(2) southEast(2)],[workspaceZ(1)
             workspaceZ(1)],'Color',[1 1 0]);
line([northEast(1) southEast(1)],[northEast(2) southEast(2)],[workspaceZ(3)
             workspaceZ(3)],'Color',[1 1 0]);
line([northWest(1) southWest(1)],[northWest(2) southWest(2)],[workspaceZ(1)
             workspaceZ(1)],'Color',[1 1 0]);
line([northWest(1) southWest(1)],[northWest(2) southWest(2)],[workspaceZ(3)
             workspaceZ(3)],'Color',[1 1 0]);

% 4x horizontal arcs
currentAngle = workspacePhi(1) / pi * 180.0;
deltaAngle = (abs(workspacePhi(3) - workspacePhi(1)) / 15) / pi * 180.0;

for (i = 1:16)
   innerCoordX(i) = (workspaceR(2) + workspaceR(1)) * cos(currentAngle / 180.0 * pi) -
                     workspaceR(2);
   innerCoordY(i) = (workspaceR(2) + workspaceR(1)) * sin(currentAngle / 180.0 * pi);

   outerCoordX(i) = (workspaceR(2) + workspaceR(3)) * cos(currentAngle / 180.0 * pi) -
                    workspaceR(2);
   outerCoordY(i) = (workspaceR(2) + workspaceR(3)) * sin(currentAngle / 180.0 * pi);

   currentAngle = currentAngle + deltaAngle;
end

for (i = 1:15)
   line([ innerCoordX(i) innerCoordX(i+1)],[ innerCoordY(i)  innerCoordY(i+1)],[
        workspaceZ(3) workspaceZ(3)],'Color',[1 1 0]);
   line([ innerCoordX(i) innerCoordX(i+1)],[ innerCoordY(i)  innerCoordY(i+1)],[
        workspaceZ(1) workspaceZ(1)],'Color',[1 1 0]);
   line([ outerCoordX(i) outerCoordX(i+1)],[ outerCoordY(i)  outerCoordY(i+1)],[
        workspaceZ(3) workspaceZ(3)],'Color',[1 1 0]);
   line([ outerCoordX(i) outerCoordX(i+1)],[ outerCoordY(i)  outerCoordY(i+1)],[
        workspaceZ(1) workspaceZ(1)],'Color',[1 1 0]);
end

% X, Y, Z axes
line([ 0.0  0.0],[ 0.0  0.0],[-0.5  0.5],'Color','b');
line([ 0.0  0.0],[-0.5  0.5],[ 0.0  0.0],'Color','g');
line([-0.5  0.5],[ 0.0  0.0],[ 0.0  0.0],'Color','r');

% Marker to represent the End Effector, hee will be its handle
hee = plot3(0.0, 0.0, 0.0, 'marker', 'o');
set(hee, 'color', [0.91 0.44 0.00]);

%-----------------------------------------------------------------
% Update EndEffector's position until ESC key is pressed
%-----------------------------------------------------------------
while ( LastKeyPressed ~= 27 )

   % Query End Effector's position
   [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'get measpos' );
   if (retVal == 0)
   else
      disp (['--- ERROR: ' response]);
   end

   measpos = eval(response);

   % Update graphical End Effector's position
   set(hee, 'xdata', measpos(1));
   set(hee, 'ydata', measpos(2));
   set(hee, 'zdata', measpos(3));

   % Repeatedly update figure
   drawnow
end
```

```
   %-----------------------------------------------------------------
   % Terminate program execution
   %-----------------------------------------------------------------
   [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'remove all' );
   if (retVal ~= 0)
      disp (['--- ERROR: ' response]);
   else
      disp (response);
   end

   [retVal, response] = haSendCommand( hapticMasterDeviceHandle, 'set state stop' );
   if (retVal ~= 0)
      disp (['--- ERROR: ' response]);
   else
      disp (response);
   end

   retVal = calllib( 'HapticAPI2', 'haDeviceClose', hapticMasterDeviceHandle );
else
   disp ('Error, unable to connect to device')
end
```

Example 12 – Description

Our Matlab program (example_12.m) uses the HapticAPI Matlab files (Matlab must know the path to the three files mentioned above). In addition, we use four self written Matlab functions we use to draw the graphics, get keyboard inputs from the user and send commands to the HapticMASTER.
Feel free to use these functions / rewrite them in your Matlab programs as well. Following is a thorough explanation of the 4 Matlab functions we use and the code of example "12-Matlab Program".

Getkey.m

Getkey.m **returns the ASCII of the key last pressed by the user** when the Figure window had the focus. It asks the *gcbo* object for the *'currentcharacter'* property and returns this value.

Scene1.m

Scene1.m **sets the perspective projection and lighting** in the scene.
The *'projection'* property of the *gca* object is set to *'perspective'*.
The viewing angle is set to 110 degrees away from the Y-Z plane, and 20 degrees away from the X-Y plane.
The camera light is created with 35 degrees azimuth and 20 degrees elevation relative to the camera.
The rest of the commands set the grid and axis properties.

Surfset1.m

Surfset1.m **sets the surface properties for a given object**.
The *mycolor* parameter determines the color of the surface.
The *h* parameter is the handle to the object.
If no parameters are supplied, yellow will be the default color and will affect all objects in the scene.
If only the color parameter is supplied, all objects in the scene will be assigned this color.
Consequently, the surface properties of all faces of the given object are being set as follows:

'facecolor' is set to the supplied color.
'facealpha' (the opacity of the face) is set to 0.99. For some reason, values lower than 1.0 result in much faster 3D rotations.
Finally, the ambient and specular properties of the surface are set.

example_12.m

Global Variables

hapticMasterDeviceHandle	Handle to the HapticMASTER device
hapticMasterIPAddress	A string representing the IP address
inertia	Virtual mass of the End Effector
blockStiffness	Block stiffness
blockSize	Block dimensions (X, Y, Z)
workspaceR	R axis extents queried from HapticMASTER
workspacePhi	Phi axis extents queried from HapticMASTER
workspaceZ	Z axis extents queried from HapticMASTER
hmCalibrated	Is the HapticMASTER position calibrated?
ch	ASCII code of the last pressed key

Initialization

First, the *'keypressfcn'* (**Key pressed function**) of the object *gcf* is **registered** and set to '*LastKeyPressed* = *getkey*'. When a key is pressed and the focus is on the Figure window, the *getkey()* function is called (defined in getkey.m). This value is returned and saved in the *LastKeyPressed* variable.

Next, we **open the HapticMASTER device** by calling the haDeviceOpen*()* function:

hapticMasterDeviceHandle = haDeviceOpen*(hapticMasterIPAddress);*

The function haDeviceOpen*()* expects the string behind hapticMasterIPAddress.

The function haDeviceOpen *()* returns a *hapticMasterDeviceHandle* (an tcp/ip connection), which is from now on the handle.

Now we can send string commands to the HapticMASTER. We do this using the *haDeviceSendString()* function, which expects:

- The handle to the HapticMASTER device
- The string command.

The command is then executed by the HapticMASTER and returns the response string.

In case of success a success message is returned.
In case of failure, an error message is returned (starting with "--- Error: ").

We always check the *responce* if it starts with "--- Error: ". We can print a message if necessary, otherwise we do nothing.

The first three commands ('get workspace_r', 'get workspace_phi', and 'get workspace_z') **query the extents of the three joint axes of the HapticMASTER** (R, Phi and Z). When the commands succeed, we copy the

response string to the *workspaceR, workspacePhi* and *workspaceZ* respectively. Since the returned string *response*, in case of success, is a Matlab formatted vector, we can use the *eval()* function to immediately translate this string into a real Matlab vector. The string '[0.1,0.4,0.6]' will be translated into the Matlab vector [0.1,0.4,0.6] and then assigned into the variable on the left side of the assignment.

Next, we **check whether the HapticMASTER is position calibrated**. We do this by sending the command 'get position_calibrated'. Now we compare the *response* string with the string 'false' (use the *strcmp()* function to compare strings). If the two strings equal, it means the HapticMASTER is not position calibrated. In this case, we **send the command 'set state init'**. The HapticMASTER will search for the end stops in all three axes. When the search process ends, the state will automatically transit to the STOP state.
This is the reason we now wait in a while loop, until the STOP state is reached. In the loop we constantly query the current state of the HapticMASTER ('get state'). After each query we compare the *response* string to 'stop'. When it is indeed 'stop', we exit the while loop by setting the Boolean *hmCalibrated* to 1 (true).

When the HapticMASTER is position calibrated, we go on by **creating the haptic world** for our example.
We set the inertia by sending the 'set inertia…' command. Here it is interesting to see how you should send a command in combination with a numerical parameter. The function *haSendCommand()* expects a string as a command. We actually construct the string in such a way that it reads the numeric parameter and concatenates its value to the command string. The result is one string that is being sent as the command. The way to concatenate the string is as follows:

[*constant_string* num2str(*variable*)]

The angled brackets ([and]) are part of the parameter!
If *constant_string* is 'set inertia ' (with a trailing white space) and *variable* equals 3.0 we send the following command:

['set inertia ' num2str(inertia)]

This translates to a single string: **'set inertia 3.0'**, which is exactly the command we want to send. If we change the *inertia* variable at the top of the program, the string changes accordingly. Of course you can also simply send the string 'set inertia 3.0' but then if you change the *inertia* variable, the string stays the same, and thus does not influence the End Effector's virtual mass.

Following, we **create a block effect** with the name *myBlock*, we **set its stiffness**, and **size**, and we **enable it**.

We then send the 'set state force' command, which **sets the HapticMASTER to the FORCE state**, in which forces are rendered and the HapticMASTER listens to the force sensor inputs.

The code that we see afterwards is the code for **drawing the scene** using Matlab graphic functions. We set the scene by calling the *scene1* function. We then turn the axis off, set the camera view angle, the background color and the option to rotate the scene.

Next, the mesh matrices for the *BlockX, BlockY* and *BlockZ* are defined. These matrices are then resized according to the size of the block in the X, Y, and Z axes and shifted (translated) to the center of the block.

The *surf()* function creates the block's surfaces based on the mesh matrices and returns a handle to the block object (*hBlock*).
We give the block a color using the *surfset1()* function (which expects a color and a handle to the object), and override the lighting properties of *hBlock* to 'flat' (no smoothing of the face will take occur).

Next, the **workspace is drawn**:
The X and Y coordinates of the four corners of the top face of the workspace are calculated according to the Phi and R extents of the HapticMASTER (queried earlier). The points are saved in the northEast, northWest, southEast, and southWest vectors respectively. The four vertical lines are drawn by connecting the eight points vertically. The Z coordinates are determined by the queried Z extents.
Four horizontal lines are drawn by connecting the eight points horizontally.

Consequently, the four horizontal arcs are drawn:
Each arc is divided into 16 line segments, which are drawn in groups of four (top inner radius, top outer radius, bottom inner radius and bottom outer radius).

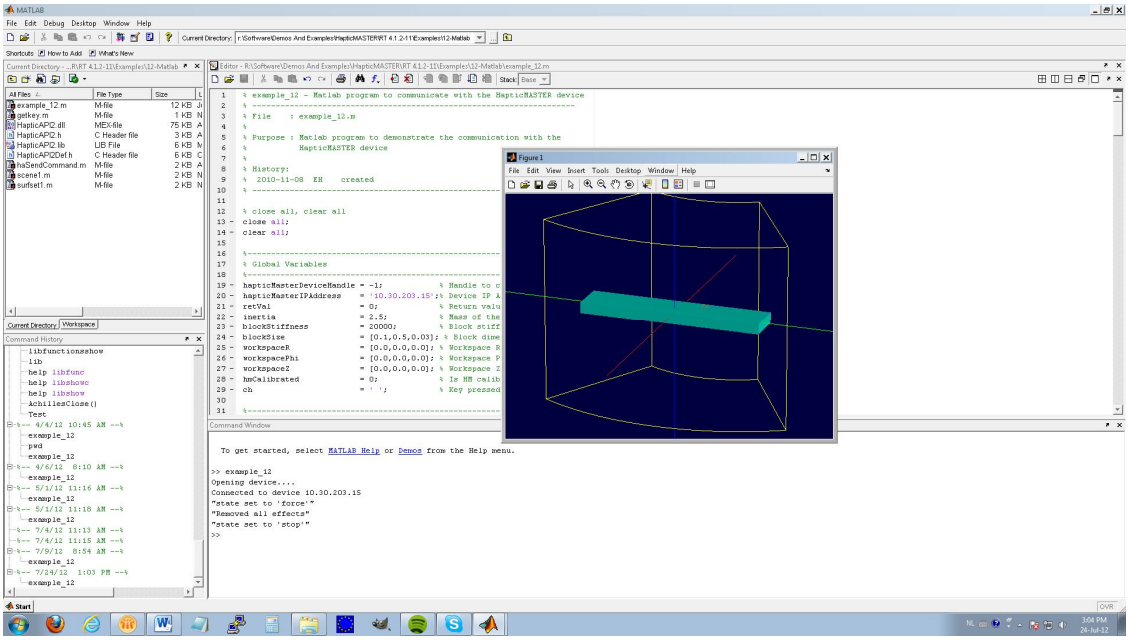
Finally, **the X, Y and Z axes are drawn** in the colors blue, green and red respectively.

Next, we **create an orange 'O' marker**, using the *plot3()* function which returns a handle to the marker object, namely the *hee* variable (handle to the End Effector).

From this moment on, the scene is created, both haptically and graphically. In a 'while' loop, we constantly query the End Effector's position and update the position where we draw the orange 'O' marker. This loop is repeated until the ESC key is pressed on the keyboard. When this happens, the variable *LastKeyPressed* equals to 27 (ASCII code for the ESC key) and the while loop terminates.

Upon termination of the program, we **remove all objects** and effects (command 'remove all'), **set the HapticMASTER to the STOP state** (command 'set state stop'), and **close the HapticMASTER device** (function *calllib()* with the function '*haDeviceClose*').

You are more than welcome to play with the dimensions of the block (change the *blockSize* vector values). Since we derive both the haptic object and the graphical object from this vector, you will notice they are both influenced by the change. This example demonstrates how you can easily write Matlab programs to communicate with the HapticMASTER device. You may have noticed the extra "complication" in the code originates from the graphics code, and not from the HapticMASTER API. You can try translating the examples introduced in this programming manual from C++ programs to Matlab programs.



15 Real-time Data Logger

The Real-time computer that the HapticMASTER utilizes runs in 2048 Hz. Each 1/2048 of a second, the forces, positions, velocities and accelerations are recalculated. In the examples shown in this manual, you have seen you can use the *haSendCommand()* of the HapticAPI to query the forces, positions, velocities etc. from the HapticMASTER.
However, programs running on a client PC which query that data are executing in a much lower frequency (normally 50 Hz). In many cases, you want to be able to query the HapticMASTER for various variables in the high frequency that it provides in Real-time. In this chapter we introduce a technique to achieve this goal. The **Realtime Data Logger** is capable of logging up to 16 variables in 2048 Hz, and send the samples to the client PC via an Ethernet connection. If you need to monitor more than 16 variables, you can create multiple Real Time Data Loggers.
The following example creates a Real-time Data Logger which logs the measured force Z, measured pos Z and model velocity Z, 2048 times per second during approximately 2 seconds.

The real-time data logger writes the data into a large buffer that can hold samples for up to 3 seconds. The program running on the PC takes care of constantly flushing the buffer and writing the data to an output file. Due to constantly flushing the real-time buffer into the host's buffer, the real-time buffer DOES NOT get full. If you do not flush the real-time buffer, it will become full after 3 seconds and the data logger will stop logging. The output file is saved on the client PC and can be later read by a Matlab program for further analysis and diagnosis of the data.

Example 13 - source code

```
//---------------------------------------------------------------------
//        1 3   -   R E A L   T I M E   D A T A   L O G G E R
//
// This example demostrates using the realtime data logger.
// It logs the measured force Z, measured position Z and model velocity C
// for 2 seconds and writes it to an output file.
// This file can then be opened by Matlab and the information inside can
// be further analized.
//---------------------------------------------------------------------

#include "HapticAPI.h"
#include "HapticMASTER.h"
#include <math.h>

#ifdef WIN32
   #include <process.h>
   #include <conio.h>
   #include <io.h>
#else
    #include <pthread.h>
   #include <sys/time.h> /* struct timeval, select() */
   /* ICANON, ECHO, TCSANOW, struct termios */
   #include <termios.h> /* tcgetattr(), tcsetattr() */
   #include <stdlib.h> /* atexit(), exit() */
   #include <unistd.h> /* read() */
   #include <stdio.h> /* printf() */

   static struct termios g_old_kbd_mode;
   /*****************************************************************************
   *****************************************************************************/
   static void cooked(void)
   {
      tcsetattr(0, TCSANOW, &g_old_kbd_mode);
   }
   /*****************************************************************************
   *****************************************************************************/
   static void raw(void)
   {
      static char init;
   /**/
      struct termios new_kbd_mode;

      if(init)
         return;
   /* get current state of console */
      tcgetattr(0, &g_old_kbd_mode);
   /* put keyboard (stdin, actually) in raw, unbuffered mode */
      memcpy(&new_kbd_mode, &g_old_kbd_mode, sizeof(struct termios));
      new_kbd_mode.c_lflag &= ~(ICANON | ECHO);
      new_kbd_mode.c_cc[VTIME] = 0;
      new_kbd_mode.c_cc[VMIN] = 1;
      tcsetattr(0, TCSANOW, &new_kbd_mode);
   /* when we exit, go back to normal, "cooked" mode */
      atexit(cooked);

      init = 1;
   }
```

```
   /*************************************************************************
   *************************************************************************/
   static int _kbhit(void)
   {
      struct timeval timeout;
      fd_set read_handles;
      int status;

      raw();
   /* check stdin (fd 0) for activity */
      FD_ZERO(&read_handles);
      FD_SET(0, &read_handles);
      timeout.tv_sec = timeout.tv_usec = 0;
      status = select(0 + 1, &read_handles, NULL, NULL, &timeout);
      if(status < 0)
      {
         printf("select() failed in kbhit()\n");
         exit(1);
      }
      return status;
   }
   /*************************************************************************
   *************************************************************************/
   static int _getch(void)
   {
      unsigned char temp;

      raw();
   /* stdin = fd 0 */
      if(read(0, &temp, 1) != 1)
         return 0;
      return temp;
   }
#endif
#include <stdio.h>
#include <stdlib.h>

#define IPADDRESS "10.30.203.12"
#define SECONDS_TO_LOG 5.0

#define PosX 0
#define PosY 1
#define PosZ 2

// Set the variables to their default values
double dInertia = 3.5;
bool bContinue = true;

// HapticMASTER handles
long dev = 0;
char response[100];

float** theMatrix;
int matrixColumnCount = 0;
FILE *fp;

//---------------------------------------------------------------------
//                          K B   M O N
//
// This function monitors the keyboard for any keys being hit.
// It runs in a thread for as long as the boolean parameter
// given with the call to this function is true.
// When this boolean variable is set to false, on application termination,
// this tread will end itself
// In this app we use the bContinue boolean var the signal when threads
// are to end
//---------------------------------------------------------------------
#ifdef WIN32
   void KbMon(void *pParam)
#else
   void* KbMon(void *pParam)
```

```
#endif
{
   while ( *((bool*)pParam) )
   {
#ifdef WIN32
      Sleep(500);
#else
     sleep(500);
#endif

      if ( _kbhit() )
      {
         switch(_getch())
         {
            case 27 :
               // Escape will end the application
               *((bool*)pParam) = false;
               break;
            default :
               // any other key will also end the application
               *((bool*)pParam) = false;
               break;
         }
      }
   }

   // At this point the boolean var is set to false, so the thread loop
   // is exit. Terminate the thread
#ifdef WIN32
   _endthread();
#else
   pthread_exit(0);
#endif
}


//---------------------------------------------------------------------
//                            M A I N
//
// 13-Real-Time-Data-Logger Main Function
//---------------------------------------------------------------------
int main(void)
{
#ifndef WIN32
   pthread_t tid;
   int err;
#endif

   // Open output file
   if( ( fp = fopen("output1.m", "w") ) == NULL ) {
      printf("Cannot open file.\n");
      exit(1);
   }

   // First create two instances of the HapticMaster (wrapper) class
   dev = haDeviceOpen( IPADDRESS );

   // Problems with connecting to the HapticMASTER
   if( dev == HARET_ERROR ) {
      printf( "--- ERROR: Unable to connect to device: %s\n", IPADDRESS );
      getchar();
      return HARET_ERROR;
   }

   // Connection to HapticMASTER succeeded
   else {

      // Initialize the HapticMASTER
      InitializeDevice( dev );

      haSendCommand(dev, "set state force", response);
```

```
      printf("set state force ==> %s\n", response);

      haSendCommand(dev, "set inertia", dInertia, response);
      printf("set inertia %g ==> %s\n", dInertia, response);

      // Start the keyboard monitoring thread
#ifdef WIN32
      _beginthread(KbMon, 0, &bContinue);
#else
     err = pthread_create(&tid, NULL, &KbMon, &bContinue);
#endif
      long logger = haDataLoggerOpen ( IPADDRESS );

//      haDataLoggerConfigure ( logger, 128, 1 );
      haDataLoggerAddParameter ( logger, "samplenr", &matrixColumnCount );
      haDataLoggerAddParameter ( logger, "sampletime", &matrixColumnCount );
      haDataLoggerAddParameter ( logger, "#HapticMASTER.Measured force.
                                  Force Sensor Z.output", &matrixColumnCount );
      haDataLoggerAddParameter ( logger, "#HapticMASTER.Model.Cartesian.pos Z",
                                  &matrixColumnCount );
      haDataLoggerAddParameter ( logger, "#HapticMASTER.Model.Cartesian.vel Z",
                                  &matrixColumnCount );

      haDataLoggerAllocMatrix ( 512, matrixColumnCount, theMatrix );

      printf( "running a data logger session...\n" );
      haDataLoggerStart(logger);

      fprintf ( fp, "%% Column 1: samplenr\n" );
      fprintf ( fp, "%% Column 2: sampletime\n" );
      fprintf ( fp, "%% Column 3: #HapticMASTER.Measured force.Force Sensor Z.output\n"
);
      fprintf ( fp, "%% Column 4: #HapticMASTER.Model.Cartesian.pos Z\n" );
      fprintf ( fp, "%% Column 5: #HapticMASTER.Model.Cartesian.vel Z\n" );

      unsigned int rowsFlushed = 0;
      unsigned int totalRowsFlushed = 0;

      // While logging time not elapsed, go on with the applications
      while ( bContinue )
      {
         // Flush the matrix and query number of rows flushed
         rowsFlushed = haDataLoggerFlushMatrix(logger, theMatrix);
         totalRowsFlushed += rowsFlushed;

         printf ( "rows: %i\n", rowsFlushed );

         if (rowsFlushed != -1) {
            // Iterate through all flushed rows
            for (unsigned int i = 0; i < rowsFlushed; i++) {

               for (int j = 0; j < matrixColumnCount; j++) {
                  // Write flushed matrix to the output file
                  fprintf ( fp, "%g ", theMatrix[j][i]);
               }
               if ( theMatrix[1][i] >= SECONDS_TO_LOG ) {
                  bContinue = false;
               }
               fprintf ( fp, "\n" );
            }
         }
         else {
            printf ("Error flushing the RT DataLogger.
                     Please press any key to continue\n");
            getchar();
            bContinue = false;
         }
      }
   }
```

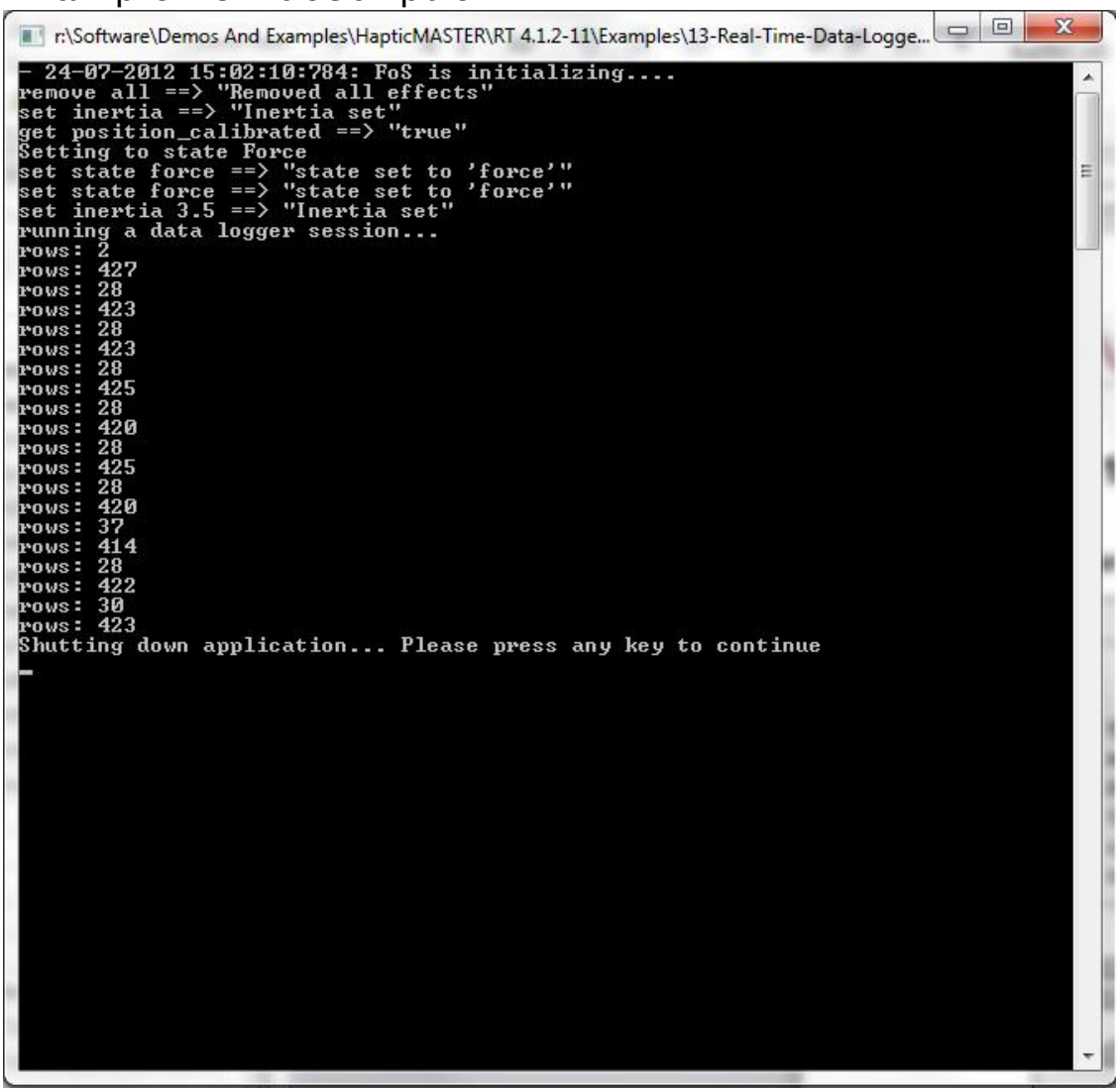
```
    if (dev) {
       haSendCommand(dev, "remove all", response);
       haSendCommand(dev, "set state stop", response);
       haDeviceClose(dev);
    }

    // When the application finishes, do some cleaning up
    printf("Shutting down application... Please press any key to continue\n");
    getchar();

    fclose(fp);

    return 1;
}
```

Example 13 – description



Global Variables

Variables are as usual in all previous examples.

bContinue	serves as a Boolean to determine whether we leave the loop
theMatrix	A pointer to the matrix that will hold the data sampled by the real-time data logger. The function *haDataLoggerCreateMatrix()* allocates the memory for the matrix itself (discussed later)
matrixColumnCount	An integer to keep track of the total number of columns in the matrix. This variable is automatically changed when we add parameters to the data logger.
fp	A pointer to a FILE handle. We use this pointer to write the flushed data directly into an output file.

The function KbMon()

This is a callback function that is called when a key was pressed.
See example “11-Master Slave” for details.

The main() function

The main() function first **opens an output file** with the name *output1.m*
This file will be used to write the logged data flushed from the buffer.

Next, the **HapticMASTER device is opened** (*haDeviceOpen*) and the *dev* handle is returned.

If opening the device succeeded, we **set the state to FORCE** ("set state force"), **set the virtual mass** ("set inertia …") and **register the callback function** *KbMon()* should a key will be pressed.

We then **create a data logger** by calling the *haDataLoggerOpen()* and supplying the IP address. This function returns a long integer *logger* which will be used from now on as the handle to the data logger.

Consequently, we **add the parameters** we want the data logger to log for us. We do this by calling the
haDataLoggerAddParameter (logger, [parameter name], matrixColumnCount) function.

logger	The handle to the data logger,
[parameter name]	Replace with a string that determines the name of the variable to be sampled.
matrixColumnCount	The value of this variable is automatically changed by the function to keep track of the total number of columns that are now found in the matrix.

Next, we **allocate memory for the matrix** that will keep the flushed data:
haDataLoggerCreateMatrix(512, matrixColumnCount, theMatrix);

512	The number of rows (samples) in the created matrix.
matrixColumnCount	The number of columns in the created matrix.
theMatrix	Pointer to the matrix that we allocate.

Finally, we have to **start the data logger**. We do this by calling the *haDataLoggerStart()* function and providing the handle to the logger.

From this moment on, the buffers of the real-time data logger are constantly filled with data, 2048 times per second. If you are not flushing these buffers, they become full after approximately 3 seconds. It is your responsibility to flush the buffers into the previously created matrix. If you don’t do that, the data logger stops logging when its buffers become full.
Flushing the buffer into the matrix takes place by calling the *haDataLoggerFlushMatrix(logger, theMatrix)*

logger	The handle to the data logger
theMatrix	Pointer to the matrix that receives the flushed data

This function returns an unsigned integer, which tells us how many rows were flushed from the real-time buffer to the locally created buffer.
This implements a non-blocking call to the flush function, which means that no delays are introduced by calling the haDataLoggerFlushMatrix function. If at the moment of calling the function 10 rows were available in the real-time buffer, you will get these 10 rows and the return value of the function will be 10. You can then copy these 10 rows in a loop into to the output file.
Once we finished copying the rows to the output file, we flush the real-time buffer again into our local buffer and we repeat this until the duration we want to log is reached. We compare the sample time (being queried by the realtime data logger), found in the 2nd column of the table (theMatrix[1][i]) with the duration we want to log. When this time is elapsed, we signal a stop to the loop by setting bContinue to false. The rest of the rows flushed from the real-time buffer are copied to the output file as well and the next time we check the loop variable bContinue, we quit the loop.
Another way to terminate the loop is pressing the ESC key, this results in the KbMon function being called, which sets the bContinue variable to false, if indeed ESC was pressed.

Once outside the loop, we remove all effects ("remove all"), set the state to stop ("set state stop"), and close the device (haDeviceClose(dev)).
Finally, we close the output file and terminate the program.

Appendix A. Utility functions in HapticMaster.h and HapticMasterOpenGl.h

In order to make the programming task even easier, we provide you with a HapticMASTER utility library. Including the *HapticMaster.h* (and optionally *HapticMasterOpenGl.h*) utility library provides the programmer with some useful defines and utility functions.
The example programs we introduce in this manual all use this utility library. We encourage you to use it in your programs as well.

HapticMaster.h

Defines

Useful defines such as *Pi, Pi2, DegPerRad, and RadPerDeg* are available for the programmer's use.
OpenGL materials for the X, Y and Z axes and for the drawing of the workspace are included as well.

InitializeDevice()

The *InitializeDevice()* function initializes the HapticMASTER device.
In order to this, the function follows these steps:

- The inertia of the End Effector (its virtual mass) is set to 3.0 Kg. This is done by calling the *haDeviceSendString()* function with the "set inertia 3.0" command.
- It is checked if the HapticMASTER is "position calibrated" (In other words – the HapticMASTER knows its boundary positions). This is done by calling the *haDeviceSendString()* function with the "get position_calibrated" command. If the HapticMASTER is position calibrated, the string "true" is returned in the *response* variable. If it is not position calibrated, the string "false" is returned.
- If the HapticMASTER returned the string "true" as a response for the "get position_calibrated" command, the execution of the function proceeds with step 6.
- If the HapticMASTER returned the string "false" as a response for the "get position_calibrated" command, a calibration process starts. This is done by calling *haDeviceSendString()* with the "set state init" command. See section 6 for more details about the HapticMASTER's states.
- When the calibration process is completed, the state is automatically changed to STOP, meaning the HapticMASTER holds its current position. The function *InitializeDevice()* waits now in a loop until the STOP state is reached. This is done by repeatedly querying the current state of the HapticMASTER – namely by calling *haDeviceSendString()* with the "get

state" command. When a STOP state is reached, execution goes on with step 6.

- The HapticMASTER's state is set to Force, meaning the force sensor is active and one can move the HapticMASTER's arm by providing force inputs. Setting the HapticMASTER's state is done by calling *haDeviceSendString()* with the "set state force" command.

Four haSendCommand() variants

In general, in order to send a command to the HapticMASTER device, you use the *haDeviceSendString()* function. This function expects a device identifier, the command input string, and the output string to which the response can be copied. The command input string is a **single string**, which describes the complete command, including the arguments (size, position, stiffness etc.). However, it is often desired to supply the HapticMASTER dynamic commands, in which the arguments of the command are read from a variable and not directly coded in the command. By working with dynamic commands, you can make sure both the graphical world and haptic world are derived from the same variables. If you change these variables in the code and recompile the program, the graphic world and the haptic world are correlated with each other, since they both read the arguments from this variable. This is the reason we introduce four variants of the *haSendCommand()* function. Each variant wraps the *haDeviceSendString()* function in such a way that the supplied arguments are patched into the command input string.

The four variants of the *haSendCommand()* function are as follows:

- No arguments
- 1 double argument
- 3 double arguments
- 1 Vector3D argument

Variant 1 requires no preprocessing at all – it is simply calling *haDeviceSendString()* with the same string.
All other variants use the *sprintf()* function in order to patch the argument(s) into the string *tempString* and with this string the function *haDeviceSendString()* is called.
With variant 4 you can supply a single Vector3D object and all 3 coordinates of the vector are patched into the command.

Notice all four variants of the function have the name *haSendCommand*. C++ looks at the type and number of arguments that you supply when calling the function, and automatically determines which of the four variants is to be called.

ParseFloatVec()

The *ParseFloatVec()* function helps the programmer with parsing string representing float vectors by writing the 3 string components into 3 float variables.

Usage example:

The command was:

command = "get measpos"

Possible response:

response = "[0.23,0.1,0.0]"

Calling *ParseFloatVec copies this string response into 3 floats:*

```
double floatsVec[3];       // An array to hold three floats
```

ParseFloatVec(response, floatVec[0], floatVec[1], floatVec[2])

➔ floatsVec[0] = 0.23
floatsVec[1] = 0.1
floatsVec[2] = 0.0

Please notice *response* is a string – no calculations can be performed on this response string. The components of *floatsVec* are float numbers and therefore can be used in numeric calculations and assignments.

BreakResponse()

Use this function to break a concatenated response string from the HapticMASTER. A concatenated response string is returned from the HapticMASTER when concatenated commands were sent to it (separated by the ';' character). The function expects 3 parameters:

- A pointer to the output string which will hold the broken response.
- A pointer to the input string which holds the concatenated response.
- An integer telling the function which of the sub-responses is to be broken from *inputString* into *outputString*.

Usage example:

The concatenated command was:

command = "set mySpring enable;get measpos"

Possible concatenated response:

response = "effect enabled;[0.23,0.1,0.0]"

Calling *BreakResponse()* can be used to break the response into either *"effect enabled"* or *"[0.23,0.1,0.0]"*:

BreakResponse(outputString, response, 1) ➔ *outputString = "effect enabled"*

BreakResponse(outputString, response, 2) ➔ *outputString = "[0.23,0.1,0.0]"*

outputString can then be further processed as a single response string from the HapticMASTER.

HapticMasterOpenGl.h

DrawAxes()

This function draws the axes of the HapticMaster's 3D Cartesian workspace.

- A red line for the X-axis
- A green line for the Y-axis
- A blue line for the Z-axis

CreatePoints()

This function queries the HapticMASTER for its boundaries on the R, Phi and Z axes. These queried values are kept in the variables *MinRadius, MaxRadius, LeftSideArc, RightSideArc, PosHeight and NegHeight. The 3D* coordinates of all arc points of the workspace are then calculated. The points are saved in two vectors – *FrontArc[][3]* and *BackArc[][3]* for the outside arc and the inside arc of the workspace. The arc is constructed of *iNrSegments* line segments, which is currently set to 10.

DrawWorkspace()

This function uses the point coordinates calculated in the function *CreatePoints()* to draw the workspace. The workspace consists of four horizontal arcs (each consisting of 10 line segments) and two vertical quads – one for the left side of the workspace, one for the right side of the workspace. The drawn workspace fits perfectly your specific HapticMASTER, since the values are queried from the HapticMASTER itself.
The function expects a handle to the device and an integer determining the color the workspace will be drawn.

If the integer is 1, the color will be Magenta (Bordeaux).
If the integer is 2, the color will be Green.
For all other values, the color will be Yellow.

Appendix B. Document Revision History

version	date	by	comments
0.1	2001-12-19	BR	First draft completed
1.1	2003-04-02		Added caution text
	2003-07-21		Added Master-Slave example
1.2	2003-08-06		Added Linux Example
2.0	2010-12-09	EH	Updated whole document to fit String commands API, Eyal Halm
2.1	2011-01-01	EH	- First Hello Haptic World example, then Hello Graphic World. - Examples with better names. - Concatenating commands
2.2	2012-09-20	EH	Updated Programmer's manual to fit Real-time 4.1.2-11 and further
2.3	2013-11-25	MG	Updated Programmer's manual to fit Real-time 4.2.
2.4	2014-05-06	EH	- Explanation for HapticAPI library for Linux 32-bit and 64-bit added. - Source code updated in the manual