OgreHaptics Manual v0.2 ('Cyta')

Jorrit de Vries (jorrit@jorritdevries.com)

January 6, 2009

Contents

Chapter 1

Introduction

This manual is intended to give you an overview of the components of OgreHaptics, a library aiming to ease the integration of commercially available haptic interfaces with the OGRE 3D graphics rendering engine (http://www.ogre3d.org/), and their usage. Before getting to these components however, here follows a brief introduction for those unfamiliar with the concept of haptic rendering.

1.1 Haptic Rendering

Haptic, from the Greek words *haphe* and *haptesthai*, refers to the sense of contact or touch. Haptic rendering refers to the algorithms and software used to display this sense through input/output interfaces. Through these interfaces the user can feel and interact with virtual three-dimensional objects. This manual will use the term haptic device for such an interface.

Haptic rendering, or the rendering of forces, can be performed in several different ways. Surface rendering, in which a force generated by the collision between the *end-effector* and a virtual object is displayed to the user, is one of the most compelling forms of haptic interaction. Another form is the rendering of environmental force effects, such as force fields or the viscosity of (a part of) the virtual space through which the user moves the end-effector. The current version of OgreHaptics only implements the rendering of force effects, with surface rendering planned for the next release.

Due to the human perception capabilities to process tactile information the rendering of forces through the haptic device needs to be executed at a rate of approx. 1000 Hz in order to present smooth haptic display. This is a significantly higher rate than the rate for which the graphical display needs to be updated in order to display smooth motion. The difference in update frequency between the two outputs needs to be addressed.

1.2 Multithreading

To maintain the continuous high rate of force rendering, the application makes use of multiple threads, e.g. a haptic thread and one or more threads which we will call the client thread. In the client thread usually the graphic display is rendered.

To prevent conflicts in accessing attributes used by multiple threads, so-called race conditions, OgreHaptics implements the means to provide thread safe data synchronisation between the client and the haptic thread. This ensures data used by the haptic thread as well as in the client, i.e. the transformation of the end-effector, can be safely used by the latter. Since the haptic thread needs to run at a much higher pace, it follows that the execution of the client thread will always be blocked when data is synchronised.

Through the use of multiple threads OgreHaptics follows the current trend in hardware development, in which the use of multiple cores is becoming more common.

1.3 Spaces

A virtual environment consists of several transformation spaces combined to display a virtual object correctly on screen. Vertices defining the geometry of a virtual are transformed in the local object space. The object itself is transformed to be placed correctly in the world space. OgreHaptics uses a similar concept for haptic rendering.

A typical haptic device has a limited physical workspace in which the end-effector can be moved around and in which the rendered force can be optimally displayed. For example, the Falcon® device by Novint Technologies has a physical workspace of 120x120x120 mm. Should the application allow the user to move the virtual end-effector around the entire virtual environment, which might consist of hundreds of thousands of units in each dimension, the user will have a very hard time tracking the virtual end-effector, since the end-effector moves with numerous world units per mm movement in the device workspace.

To limit the virtual workspace dimension OgreHaptics provides the means to define a subset of the world to be mapped against the physical workspace of the device. The box describing the dimensions of this subset is called the touch space, positioned around the center of the world. To be able to reach all parts of the virtual environment, the client application can set a transformation matrix to transform the touch workspace to world space coordinates.

The transformation from world space to device workspace is as follows.

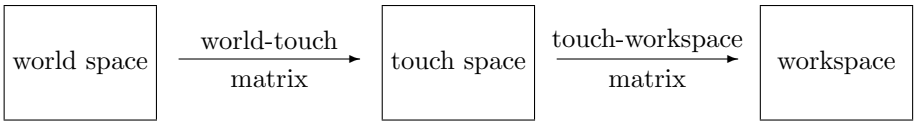

Figure 1.1: Transformation matrices

Chapter 2

Core Objects

This chapter provides you with short descriptions of the core objects used by OgreHaptics to perform haptic rendering. Shown below is an UML diagram giving an overview of the core objects and how they relate to each other.

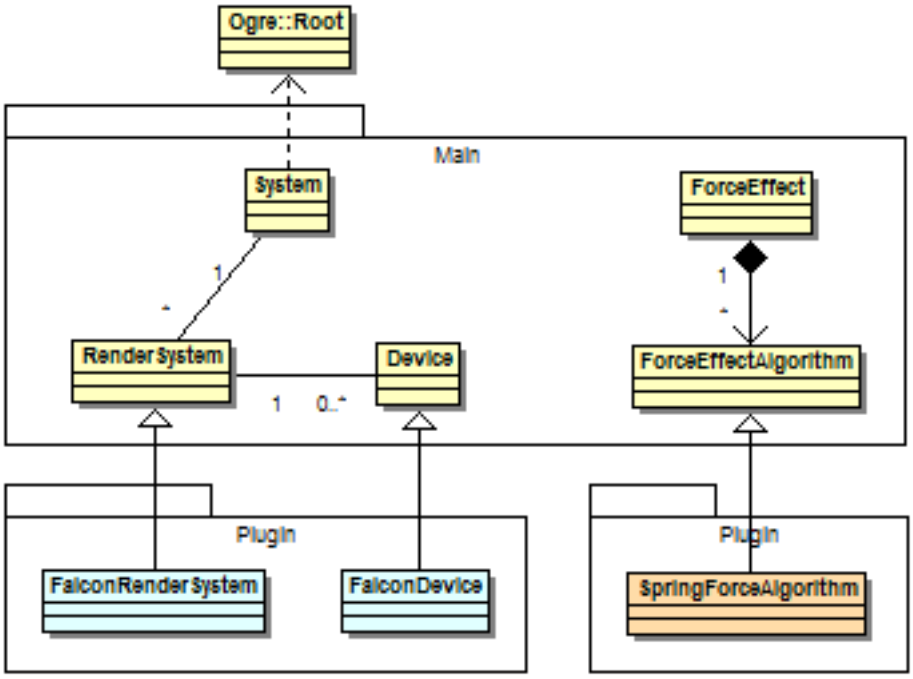

Figure 2.1: UML overview

More details on these objects can be found in the following sections.

2.1 System

The *System* object is the main entry point to the OgreHaptics system. This object *must* be the first one of the OgreHaptics objects to be created and the

last one to be destroyed. It relies on the Ogre::Root class being instantiated.

Through the System object plugins, i.e. for adding haptic rendering devices to be available to the application, are loaded, resource managers are initialised when needed and it also serves for creating new devices and updating the whole system every graphical rendering frame.

2.2 RenderSystem

The *RenderSystem* is an abstract base class which defines the interface of for an haptics API. It is currently implemented for the OpenHaptics™ SDK (http://www.sensable.com/) to be used with PHANToM® devices and the HDAL™ SDK (http://www.novint.com/) to be used with the Falcon® device, but can be extended to make other types of devices available. It is reponsible for managing the scheduler used by the haptic thread for the specific API's.

A typical application does not communicate with the RenderSystem object directly – only in the cases when you want to access advanced methods such as scheduling callbacks from the haptic thread you need access to the RenderSystem directly.

2.3 Device

The *Device* object defines an interface to the underlying haptics API needed to control the haptic device supported by that API. An instance of the Device class represent a single physical haptic device.

Through the Device object you can control the input and the output of a haptic device, such as the settings of the touch space (see 1.3) to be mapped to the device workspace and the transformation of the touch space, and query the current state of the device as well as register to listen to events generated by the device.

Instances of the Device class are created through the System object. It is advisable to first obtain a list of information about available/connected devices, and instantiate a new Device using information from that list. If you know which API will be used for communicating with the haptic device(s), it is also possible to instantiate zero-based indexed devices for that API.

2.4 ForceEffect

The *ForceEffect* object provides the means to create environmental forces such as viscosity, springs and rumble. These forces are environmental, because they are not related to a particular entity or shape in the virtual environment.

A force effect consists of one or more *ForceEffectAlgorithm* subclasses, which implement the actual algorithms used for rendering forces. Through composition using multiple algorithms complex effects can be created. Since the calculations are executed in the haptic thread, care must be taken when modifying parameters on algorithms used by an actively rendering force effect. To update parameters in a thread-safe manner, use the **ForceEffect::setAlgorithmParameter** operations, which will provide the mechanism

to update the algorithms safely through **System::update**. This is done to prevent the rather expensive locking of data every haptic frame.

The *ForceEffectManager* object controls the list of available effects. Force effects can be either created programmatically using **ForceEffectManager::getSingleton().createEffect**, and add named algorithms through **ForceEffect::addAlgorithm** and tweak settings fo both algorithm(s) and the created effect, or by specifying the effect and algorithms in a 'script' which is loaded at runtime.

To apply a force effect you call the **ForceEffect::start** method and give it a pointer to the device you want apply the calculated force to.

Chapter 3

Scripts

OgreHaptics drives some of its features through scripts in order to make it easier to set up. The scripts are simply plain text files which can be edited in any standard text editor, and modifying them immediately takes effect on your OgreHaptics-based applications, without any need to recompile. This makes prototyping a lot faster. Here are the items that OgreHaptics lets you script:

ForceEffect Scripts for the creation of force effects (section 3.1)

3.1 ForceEffect Scripts

ForceEffect scripts allow you to define effects used for haptic display which can be reused and modified easily without having to hard-code the settings in your source code. This allows a quick turnaround on any changes you make. Force effects which are defined in scripts are used as templates, and multiple actual effects can be created from them at run-time.

3.1.1 Loading Scripts

Force effect scripts are loaded at initialisation time by the system: by default it looks in all common resource locations (see **Ogre::Root::addResourceLocation**) for files with the '.forceeffect' extension and parses them. If you want to parse files with a different extension, use the **ForceEffectManager::getSingleton().parseAllSources** method with your own extension, or if you want to parse an individual file, use **ForceEffectManager::getSingleton().parseScript**.

Once scripts have been parsed, your code is free to instantiate effects based on them using the **ForceEffectManager::createEffect** method which can take both a name for the new effect, and the name of the template to base it on (this template name is in the script).

3.1.2 Format

Several effects may be defined in a single script. The script format is pseudo-C++, with sections delimited by curly braces ('{', '}'), and comments indicated

by starting a line with '//' (note, no nested form comments allowed). The general format is shown in the example below.

```
force_effect Examples/ConstantForce
{
    sustain_mode temporal
    sustain_duration 200
    fade_in_duration 75
    fade_out_duration 100

    algorithm Constant
    {
        magnitude 11
        direction 0 0 1
    }
}
```

Every force effect in the script must start with 'force_effect' to define the type of object which is about to be created. This must be done, since OGRE provides means to combine multiple types of scripts in one file, the *.os file. After the object definition the name is written, in this example 'Examples/-ConstantForce', which must be globally unique. It may include path characters to logically divide up your force effects and to avoid duplicate names, but the system does not treat the name as hierarchical, just as a string.

A force effect script can have top-level attributes set using the scripting commands available, such as 'fade_in_duration' to set the time in milliseconds in which the force effect will fade in. Algorithms are added as nested definitions within the script. The parameters available in the algorithm sections are entirely dependent on the type of algorithm.

3.1.3 Top level ForceEffect Attributes

sustain_mode A force effect can be started in two different modes for the sustaining state. In *temporal* mode an effect will run as long as the set duration, in *persistant* mode an effect will run until it is stopped explicitly.

Possible values are 'temporal' and 'persistant'. Defaults to 'persistant'.

sustain_duration Describes the duration of the sustaining state in milliseconds of an effect in sustain mode 'temporal'. Defaults to 0.

fade_in_duration Describes the duration in milliseconds by which the force effect will be faded in linearly. Defaults to 0.

fade_out_duration Describes the duration in milliseconds by which the force effect will be faded out linearly. Defaults to 0.

3.1.4 Algorithms

An *algorithm* section in your force effect script defines the type of algorithms used for rendering forces to a haptic device. A force effect can have an unlimited number of algorithms, although the more algorithms are used, the more expensive the force effect will be to render.

OgreHaptics provides several algorithm types to be used, but more types can be added through plugins. Algorithms are registered with a unique name, and you can use that name to define an algorithm to be used by the force effect. The following list describes the attributes of the algorithms implemented in the default plugin.

Constant Used to render a constant force in a single direction using the formula $F = s \cdot \vec{v}$, where s is the magnitude of the force and $\vec{v}$, which is of unit length, the direction.

magnitude Describes the magnitude of the force in N. Defaults to 0.

direction Describes the direction of the force in device workspace coordinates. Vector will be normalised. Defaults to a unit-z vector.

SawtoothWave Renders a force using the formula $F = ((2 \cdot (t \cdot -(\lfloor t \cdot f + \frac{1}{2} \rfloor) \cdot A + D) \cdot \vec{v}$, where t is time in seconds since the start of the wave, f the number of periods per second, A is the amplitude, D the offset and $\vec{v}$ the direction of the output.

amplitude Describes the amplitude of the sine wave. Defaults to 0.

offset Describes the offset of the sine wave. Defaults to 0.

frequency Describes the frequency of the sine wave in Hz. Defaults to 0.

inverse_ramping Whether this sawtooth wave is ramping in reverse. Defaults to false.

direction Describes the direction over which the sine wave oscillates in device workspace coordinates. Vector will be normalised. Defaults to a unit-z vector.

SineWave Renders a force using the formula $F = (A \cdot \sin(\omega t \cdot \theta) + D) \cdot \vec{v}$, where A is the amplitude, ω the angular frequency in radians per second, θ the phase shift, D the offset and $\vec{v}$ the direction of the output.

amplitude Describes the amplitude of the sine wave. Defaults to 0.

offset Describes the offset of the sine wave. Defaults to 0.

frequency Describes the frequency of the sine wave in Hz. Defaults to 0.

phase_shift Describes the phase shifting of the sine wave in degrees. Defaults to 0.

direction Describes the direction over which the sine wave oscillates in device workspace coordinates. Vector will be normalised. Defaults to a unit-z vector.

Spring Used for rendering a spring force using Hooke's law: $F = k \cdot \vec{v}$, where k is the stiffness of the spring in N/mm and $\vec{v}$ is the displacement from the equilibrium/anchor position.

stiffness Describes the stiffness of the spring in N/mm. Defaults to 0.

anchor_position Describes the anchor or equilibrium position in device workspace coordinates. Defaults to the center.

Viscous Renders a force based on Stokes' law: $F = 6\pi \cdot \mu \cdot R \cdot V$, where μ is the fluid's viscosity (in $Pa \cdot s$), R is the radius of the particle moving around in the fluid (in m) and V is the velocity of the particle.

viscosity Describes the viscosity of a fluid in $Pa \cdot s$. Defaults to 0.

radius Describes the radius of the sphere moving through the fluid in m. Defaults to 0.001m (1mm).

Chapter 4

Known Issues

A number of implemented features are not running as expected and will hopefully be overcome in the future. Here is the list of issues that we are aware of:

- The ViscousForceAlgorithm is causing an HD_EXCEEDED_MAXIMUM_VELOCITY error using the OpenHaptics™ SDK. It can be run with extremely low values, but we are looking for another implementation.