The JHapticGUI Library

Intent

Force feedback devices are widely used for a diverse range of application including 3D modeling, medical simulations, games and virtual sculpture. Normally these applications are based on a 3D vector graphics (e.g. OpenGL) scene which is “augmented” with sense of touch, namely the possibility to touch the virtual objects that are rendered on the screen.

However, a relatively unexplored field is the use of such devices to provide a haptic modality of interaction with everyday programs featuring a 2D graphical user interface made of buttons, lists, menus etc.

Such a modality, besides opening up new ways of interacting with graphical user interfaces, would have an impact on the access to common software by people with visual impairments.

The JHapticGUI library allows to **embedd force feedback devices in the development of Java GUI based programs**.

When programming graphical user interfaces, the challenges to address are somewhat different from 3D vector graphics ones and have mostly to do with handling and syncronizing a number of threads in order to keep the computation running and the interface reactive at the same time, thereby letting the user still have control on it.

Whereas haptic devices normally expose native programming API’s (C/C++), the Java language is an attractive choice for GUI programming, allowing for a nice trade-off between performance of the programs execution and quick development times. Besides that, the Java language sports a nice mechanism for incorporating native libraries into Java code called **Java Native Interface** (JNI).

The JHapticGUI library leverages the JNI mechanism to easily allow the implementation of GUI based programs, that can be operated using force feedback devices.

Features

- The jHapticGUI library provides a clean and abstract way to interface with haptic devices from Java code;
- It comes as a small library of Java classes and a few C++ files to include in your code, in order for your haptic scene implementation to fit in the framework;
- It is completely device independent, in fact it leaves the whole haptic API programming to the client code and focuses on the communication and thread syncronization between the C++ code and the Java code;
- It is especially - but not exclusively - suitable for GUI programming;
- It provides an abstraction mechanism on the inter-thread communication protocol by means of a general purpose message exchange system. The details of communication implementation are hidden to the client code but the content of the message is left to be defined to the application;
- It is released under the GNU General Public license.

Architecture

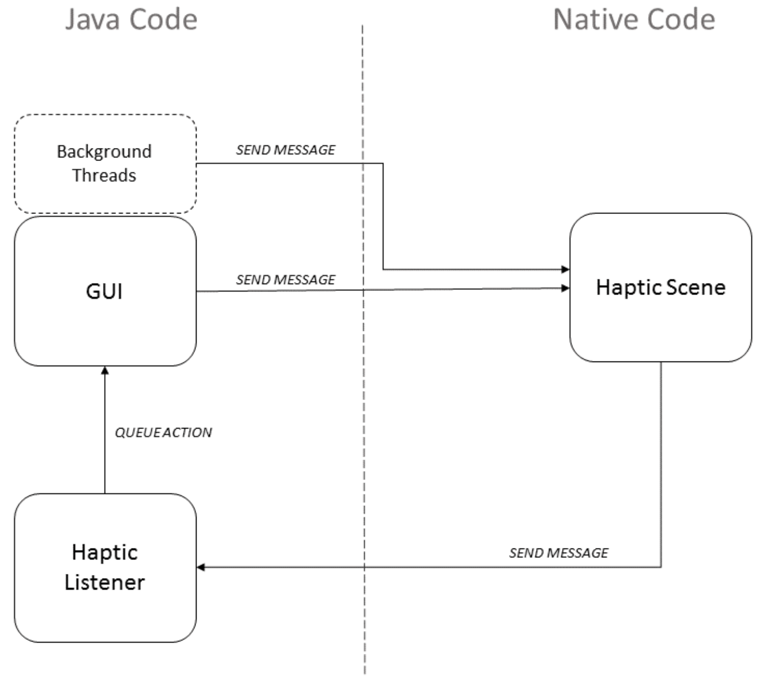

Figure 1: The Architecture of the jHapticGUI library

The jHapticGUI library comes in two parts:

jHapticGUI.jar that contains java classes to interface with the haptic device from the Java code;

A set of C++ files to help you integrate the haptic device API native code into the Java code.

Figure 1 summarizes the architecture of the library. Each box represents a thread in execution. The communication between the java code threads and the native code thread is carried out by means of **messages exchange**.

The **haptic scene** box is the thread running the native code that handles the haptic device [1]. It receives messages from the GUI thread and it sends messages to the haptic listener.

The **GUI box** is the *Event Dispatching Thread* [2] (EDT) the thread that draws the graphical user interface on the screen. It sends messages to the haptic device upon user actions (e.g the user clicks on a button).

[1]in fact this might be made out of more than one thread under the hood, but as far as this document is concerned it can be assumed that there is just one thread executing.

[2]the EDT is specific to the *Java Swing* libraries, however the concept can be extended to other GUI libraries that use the same principles of the EDT, namely a dedicated thread that handles the user input, delegates time demanding tasks to other threads, and has a queue of events to acquire the results of such taks from the other threads and display them to the user.

The **backgound thread** is optional and has been added for sake completeness. It sends messages to the haptic device upon events that are not generated directly by the user through the GUI (e.g. a new packet of data arriving from the network).

The **haptic listener** box is an auxiliary thread that lives in the Java space and accepts incoming messages from the haptic device. When a message arrives, involving an updated to the GUI, an event is pushed in the EDT queue to be displayed.

The existence of the haptic listener frees up the EDT from the duty of listening to incoming messages from the haptic device, as this would affect the reactivity to the user input. In case the incoming message doesn't require any update to graphical user interface, the haptic listener can take care itself of handling it.

Let's see the thread communication in action with a couple of examples.

Haptic Browser

Suppose we have a haptic browser, that diplays menu entries in web pages as touchable 3D shapes. When a new page is loaded in the browser, the GUI thread sends a message to the haptic scene with the list of the menu entries in it. Upon receiving the message, the haptic scene renders the items as a set of shapes arranged one next to the other. Then the user grabs the haptic device and starts exploring the haptic scene. Each time the user touches a shape, a message is sent to the haptic listener with a unique id, which links the shape to a specific item. The haptic listener receives the messages from the haptic device and first plays a sound to give feedback to the user, then pushes an event to the GUI, passing on the id of the item. When the EDT processes the event, the menu item is selected in the interface so if - for example - now the user hits the space bar, the web page pointed by that menu item is opened on a new tab.

Haptic xy-pad

An xy-pad is a common tool in music production to control two musical parameters at the same time, via the MIDI protocol. Let's see how a haptic device based xy-pad could be designed. The GUI displays a graphic representation of the xy-pad. The haptic scene is made up of one single shape that can be a square or a circle. When the user touches and scrubs on the shape, a message, containing the coordinates of the user touching position, is sent to the haptic listener. The haptic listener, upon receiving the message, sends a MIDI message to an attached synthesizer and pushes an event to the EDT to update the graphics with the new position. When the user wants to switch the haptic shape - from square to circle or vice versa - she clicks on a button on the GUI and a message is sent from the GUI to the haptic device, resulting in the shape switch.

These are of course trivial examples, but they serve to give an idea of how the differen parts of the architecture collaborate with each other, in order to give the user a haptic experience of tasks that would normally be performed with mouse and keyboard only.

Messages

A message is a bundle of data exchanged by the jHapticGUI threads. It is made out of three fields:

command : which specifies the effect the message is going to have. The type of this field is *String*

arguments : optional arguments of the command. The type of this field is also *String* and it's intended to be the string representation of any type of data. Indeed, the data sent as argument could be of any type - int, float, arrays, often spacial coordinates - and this choice allows for maximum flexibility in the number and type of arguments, without the need of defining ad-hoc signatures on both sides of the code for each message.

ID : an (optional) unique identifier of the object, the command refers to. Rendering an interface in haptics, means there are a number of GUI objects that are represented in the haptics code as well. Since these two views share the same content, it makes sense to have an id to address the very same objects, when exchanging commands between each other. The type of this field is *int*.

The way the exchange is implemented is trasparent to the user: the library provides dedicated functions on both sides to send and receive messagges. Let's see more in detail the main public classes both on the Java and on the C++ side.

Main Classes

HapticDevice.java

This class is a placeholder of the haptic device from the Java code side. Once this class is created (by an dedicated factory method) the haptic device has been started and initialized and it's ready to receive commands to draw the haptic scene. Sending a message to the haptic device is as simple as calling the following method on a `HapticDevice` object.

```
sendMessage(String command, String args, int ID)
```

The call can be made by any thread in the Java space, the library makes sure that the message command is executed in the haptic thread. Furthermore, the call guarantees a swift execution by using a synchronized queue; it can therefore be used in the EDT, without the risk of freezing the ineterface. See the "Messages" section for a detailed explanation of the arguments.

HapticListener.java

An abstract class implementig the haptic listener thread. Clients of the library must provide their own concrete instance of this class, by overriding the abstract method:

```
messageReceived(String command, String args, int ID)
```

See the "Messages" section for a detailed explanation of the arguments. This callback is called in a dedicated thread - see Figure 1 - each time a new message arrives from the haptic thread. Therefore any code that modifies the graphical user interface must be explicitly queued for execution in the EDT [3].

The haptic listener thread needn't explicitly be started by the client code. Just create your instance of the HapticListener class and pass it to the haptic device factory method.

HapticScene.h

If the message infrastructure and thread handling is done for you by the jHapticGUI library, the specific effect of each message is what is left for you to implement. On the haptic side, this means creating a class that inherits from the abstract class `HapticScene` and that implements the haptic domain logic.

Note that the jHapticGUI library is not device specific. You can therefore use any haptic device[4] and any API as long as they are based on an OpenGL virtual scene.

Let's see the methods that need to be implemented by the concrete subclass of `HapticScene`:

`virtual bool initHaptics(void)`

Tries and initializes the haptic device, returns `false` if the initialization fails, `true` otherwise.

`virtual void initGL(void)`

Initializes the OpenGL window, after the haptics has been successfully initialized.

`virtual unsigned int processMessage(const Message & m)`

Handles an incoming message from the Java thread. Returns an update code if the message alters the scene. The code is passed to `drawScene` for updating the scene accordingly. `Message` is a simple struct included in the C++ code delivered with the library, that contains three fields as per the "Messages" section.

`virtual void beginFrame(void)`

Called at the beginning of each haptic frame.

`virtual void endFrame(void)`

Called at the end of each haptic frame.

[3]for more info, see `javax.swing.SwingUtilities.invokeLater(Runnable doRun)` in the official Java Core API

[4]the library has been tested with the OpenHaptics® Academic Toolkit library and the HAPI (`http://www.h3dapi.org`) library, respectively on a Geomagic® Touch$^{\text{TM}}$ device and a Novint Falcon® device

```
virtual void drawCursor(void)
```

Draws the haptic cursor, namely the haptic device proxy.

```
virtual void drawScene(unsigned int messageUpdate,
            unsigned int callbacksUpdate)
```

Draws one frame, both in graphics and in haptics, of the scene. `messageUpdate` and `callbacksUpdate` are unsigned integer codes, specifying what needs to be updated in the haptic and graphic scenes. `messageUpdate` is the value returned by `processMessage` and flags that a message from the Java thread alters the scene, which must therefore be updated. `callbacksUpdate` is the value returned by `checkCallbacks` and flags that a haptic event happened that alters the scene, which must therefore be updated. Note that the instance fields of the `HapticScene`'s subclass might be a better place for some kind of information about how to draw the scene. Especially information that needs to be read at each haptic/graphic frame, such the position of an object. `messageUpdate` and `callbacksUpdate` are useful to flag one-time updates, without having to keep track of that in your instance variables. Their use is however not mandatory.

```
virtual unsigned int checkCallbacks(void)
```

Checks for haptic callbacks, namely functions that are called when a haptic event occurs. Typical haptic events are: proxy touching an object, proxy moving, button pressed etc. Returns an update code if any event changes the scene. The code is passed in the next frame to `drawScene`, for updating the scene accordingly.

```
send(const Message & m)
```

This is not a method of `HapticScene` to be overridden, but rather a pointer to function that is passed to `HapticScene` when it is constructed and that is available to its subclasses. You can use it in your code to send messages to the Java haptic listener thread. `Message` is a simple struct included in the C++ code delivered with the library, that contains three fields as per the "Messages" section.

An Implementation Example

An implementation of the xy-pad example (described above), that shows how to use the library in your code, is provided in the jHapticGUI web page. For more information see `https://code.soundsoftware.ac.uk/projects/jhapticgui`.