Proceedings of EuroHaptics 2004, Munich Germany, June 5-7, 2004.

The Haptik Library - a Component based Architecture for Haptic Devices Access

M. de Pascale, G. de Pascale, D. Prattichizzo, and F. Barbagli

Siena Robotics and Systems Laboratory, University of Siena, Siena, 53100, Italy
[mdepascale, gdepascale, prattichizzo, barbagli]@dii.unisi.it

Abstract. *Haptik* is a component based library recently developed at the University of Siena. It allows easy but powerful low-level access to haptic devices, both in a uniform way, common to all devices, or in a more specialized one. Haptik addresses haptic device access. It is non-invasive and easily usable with existing applications. Its architecture guarantees a binary compatibility of applications with future versions of devices, library and plugins, still maintaining the maximum performance achievable directly with devices SDKs. The library is freely downloadable at http://sirslab.dii.unisi.it/haptic/haptik.htm

1 Introduction

Despite the valuable works done in the last years by the research community in the field of haptic related technologies, yet many problems still prevent wide spreading of haptic-enabled applications and devices. Surprisingly enough one of these problems is the access to haptic devices for software developers. Actually two ways of accessing haptic devices for software developers exist:
- device-specific low-level API written by the manufacturer
- generic high-level

The former are usually very simple APIs, that expose a limited set of features (basically reading the state of the haptic interface and sending forces back), tied to a particular class of devices from the same manufacturer. This category includes, for example, the Delta Haptic Device API from ForceDimension [2], a set of functions for use with the C++ programming language to access Delta family devices, and the DeviceIO functions from SensAble, unfortunately released only as part of the GHOST 4.0 SDK.

On the contrary, the latter are large libraries, untied from a particular manufacturer, and thus able to work with more than one kind of device. Besides haptic rendering, they typically address also graphical and audio related aspects. Many products belong to this second category: the proprietary ReachIn API [3], which was the first haptic/graphic API to be independent from a particular device, the eTouch API [4] developed from Novint, which is a set of open module haptic/graphic/audio libraries which can be freely expanded and modified, and the CHAI Libraries [5] presented at Eurohaptics 2003 but not yet released.

Unfortunately both these solutions for hardware access present some drawbacks.The use of specific APIs bounds applications to a single device type, or

compels developers to use many different APIs, typically very far from each other. What normally happens is that developers have to rewrite some code which works as a common interface to devices from different manufacturers. Moreover, each of this API requires its own device-specific drivers and libraries. Even if an application has been written to use many device types, it cannot run on systems that do not have all of the used APIs installed. For example, an application written to use both Ghost API and Delta API, will be statically linked to some DLLs from both SDKs. For the application to be executed it's thus mandatory the presence of all of these DLLs, otherwise the operating system will raise a runtime link error. Therefore even if application can correctly work with only one of the two supported devices, the running system must have installed both APIs.

Generic libraries, on the contrary, provide standardized access to different devices. Moreover they typically fulfil even other requirements of visio/haptic applications, providing rendering primitives, collision detection algorithms and physical models. All the best known libraries of this kind ship in the form of C++ class hierarchy [3–5], heavily relying on inheritance and polymorphism to guarantee a certain amount of flexibility and to support callbacks. This structure could lead to some problems: Adding haptic rendering to an existing application with its own graphic system requires some effort either to adopt the library's one or to make the library working with a different graphic system. Moreover if application already has its own class hierarchy, it becomes necessary to merge them: C++ class hierarchy merging is pretty easy to implement, but sometimes can make source code less clear and uniform[1], and additionally may slow down the compiled code, even for parts apparently not directly involved (e.g. using multiple virtual derivation). Obviously these problems do not arise with scratch-written code, and a visio hapti application can be comfortably built up with only few hours of development. Unfortunately, to the best of our knowledge, no one of these libraries directly support last generation graphic technologies such as programmable GPUs. Therefore, sometimes they have too much stuff, sometimes they do not have enough. However these libraries had a great value in the research and educational fields since they allow to easily test new algorithms and to develop non trivial applications in a short time. They show their limits whereas cutting-edge technologies must be used and very high performances must be achieved.

Moreover there are some other portability and compatibility issues, common to both kinds of libraries, mainly related to software engineering design. For example, potential modifications to this libraries in future releases require to recompile or even to modify the application source code, preventing any form of binary compatibility.

In this paper we present a library that, using a component based architecture, overcomes these limitations, achieving many benefits such as device independency, driver version transparency and application's binary compatibility with future devices, SDKs and library versions.

[1] This is true especially if the two class hierarchies use different coding conventions.

2 Haptik Library

The Haptik Library is a small library with a component[2] based architecture providing uniform access to haptic devices. It does not contain graphic primitives nor physical algorithms or complex class hierarchies, but instead provides a set of interfaces that hide differences between devices to the applications. Moreover the library has been built with a high flexible infrastructure of dynamically loadable plugins. Actually Haptik Library is implemented only on Windows OSes, mainly due to lack of support from haptic device manufacturers for other platforms, and so in what follows we will talk of DLLs. Obviously enough the same mechanisms can be easily achieved on other systems supporting forms of dynamic linking libraries. As soon as device manufacturers began to support other OSes the Haptik Library will be ported to other platforms. Moreover, being component based, the Haptik Library is a cross language solution.

2.1 Library Overview

Differently from existing libraries, the Haptik Library exposes haptic devices as components. To get access to a device an application does not instantiate a certain class from a hierarchy, but instead requests an interface from the library. The application can explicitly points out the device to access (a chosen one, the first available from a certain manufacturer, the default one, etc) and the type of interface it would use. Following is a sample of the minimal code that an application uses when working with Haptik library:

```
device = Haptik.GetDeviceInterface( HAPTIK_DEFAULT_DEVICE,
                                    HAPTIK_IHAPTIKDEVICE);
device->Init(object,callback);
device->Start();
```

This code simply requests to the library an interface of type *IHaptikDevice* to access the default device (i.e. the first available). Then some actions are requested directly to device (i.e. setting up callback system and starting reading state and applying forces). As components in fact, each device typically exposes more than an interface. A standard interface, common to all kind of devices, is always supported. In addition some other custom interfaces are implemented, through which special features of that device are made available. Therefore an interface can be thought as a sort of abstract device, which can be "played" by many different real devices. In this way applications can uniformly interact with very different devices. For example through the common interface position and orientation of a device can be read. The actually accessed device could be one that has a stylus with 6DoFs or it could be a simpler one with only 3DoFs. In the latter case device implementation simulates orientation via software, making differences between devices transparent to applications. With regards with the

[2] Haptik library is based on the Component Model, which is the abstract model powering technologies such as OMG CORBA [6], Microsoft COM [7], as well as SUN JavaBeans. Therefore at least a minimal knowledge on the subject is required to truly understand the advantages deriving from the use of the library. See [8] for a brief overview.

example code, *IHaptikDevice* is a simple standard interface which exposes the capabilities of 3DoF devices, and is guaranteed to be implemented by any device. Special features of each device can be exposed through custom interfaces specifically designedto reflect enhanced capabilities available only on that device. Either in this case, where an interface effectively reflects a single particular device, the use of interfaces gives some benefits: a device-specific interface can be implemented even on simpler devices, maybe simulating the missing capabilities via software. In this way it allows the usage of customized applications that in an old scenario will work only with that device. Moreover any device can ultimately be simulated completely via software, exposing both common and custom interfaces to be used for debugging purposes, as teaching tools or for simple demonstration on mobile computers with no need of haptic hardware. For clarity sake, in the above sample code we could imagine application requesting an interface of kind *IPhantomDevice* or *IDeltaDevice*, to exploit peculiarities of hardware.
Furthermore the pluggable structure of the library (described in the next section) allows easily adding support for new devices, replacing old plugins with never versions, and using custom-written plugins for specific needs. All this achieved without requiring recompilation neither for the library nor for the application. In fact even the library itself and plugins are implemented as components, thus old plugins will continue to work unchanged (at binary level) with new version of the library even if the library/plugins communication protocol will be eventually completely changed.

2.2 Library Structure and Behaviour

The Haptik Library consists of its main dll, *haptik.dll*, which is statically linked by the application, and a series of additional dlls, loaded transparently at runtime, each one containing a plugin. In fact the library dll performs only few services, exposing methods for device enumeration, info/status querying and interface requesting, while the plugins effectively implements the routines necessary for a particular class of devices.
At runtime, and only when needed, the Haptik Library tries to dynamically load and initialize plugins (Fig. 1). The loading of some plugins could fail: in fact plugins are typically statically linked to some device specific dlls that could be missing. For example a plugin that expose phantom devices is tied to the presence of the related drivers, and if these are not installed on the running system, than the OS will refuse to load the plugin's dll, due to the unresolved references. In this case the plugin is simply skipped and will not prevent the application from running.
Moreover a fallback system is implemented inside the library to address versioning problems: for example, let's imagine that two plugins, phantom40.dll and phantom31.dll, both implement support for phantom devices, but have been written with different versions of the GHOST SDK.
At runtime if phantom40.dll could not be loaded, than prior version are tried. In this way applications always use the most recent drivers but keep working unchanged on systems with older ones. As previously stated this dll dependency is one of the problems with prior haptic libraries, but with this architecture only

a plugins is affected, while the application could run despite of drivers and libraries dlls presence and version. After the loading phase, all plugins are queried about supported device and all these information are uniformly exposed. Application can now request an interface to a device and start using it. Using internal reference count the library knows when it's safe to unload unused plugins, thus guaranteeing no useless waste of memory (even virtual/paged one).

2.3 Performance considerations

Some considerations are needed about the potential performance penalty added by such a runtime loading scheme and by interface invocations. We state that actually there is no overhead with respect to a statically-linked object-based code. In fact, once an application obtains an interface to a device, the library is completely stepped over and not traversed by interface invocations, which instead go directly through plugin code (Fig 1).
Moreover, as already stated, dynamically-loading permits a low memory footprint, which could perhaps help performances. Even the low overhead added by interface invocations, with respect to direct method invocations, is negligible. In fact using the event-driven mode, methods from an interface are invoked only during initialization of the device while, for the rest of time, communication is performed by way of callbacks, exactly in the same way as other procedural or object based libraries. In addition, using low-level non-invasive callback techniques, the Haptik Library does not require modifying the application class hierarchies, and at each cycle performs a direct address invocation instead of a virtual call as with classic inheritance-based callback techniques. We can thus state that using Haptik Library an application run at least as fast as using other libraries or even faster.

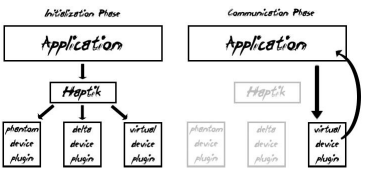

Fig. 1. Invocation paths

2.4 Advantages and Intended Audience

Due to its architecture several advantages arise from the usage of the Haptik Library, especially if compared to existing device-specific or generic libraries:
- Haptik allows for an easy Low Level access, with no need for tricks to circumvent the API, while still exposing some high-level features like device enumeration, default device selection, haptic-thread management and synchronization.
- Haptik has a fast learning curve with respect to high level libraries: due to their complexity they require at least a brief knowledge of the class hierarchy to be used. Even accessing devices, without using the high level stuff, requires a certain set up. With Haptik few lines of code are enough to use a device.
- Haptik is not-invasive: it supports both polling and event driven behaviours and as callbacks it can both invoke procedures or non-virtual methods on any object without requiring inheritance. In addition it's only about haptic: following the holy principle that "if you do not use something you do not have to pay for

it" only device-access problem is addressed. Therefore it could be easily used in conjunction with other libraries for 3D rendering, collision detection and physical simulation, chosen or implemented by the developers.
- Haptik is truly open and extensible: anyone can design new interfaces and implement plugins for any device.
- Haptik offers the highest degree of compatibility and portability achievable: thanks to component based architecture, all these advantages are guaranteed at binary level. Therefore an application written (and compiled) to run with Haptik Library will always work, on every system, no matter what kind of drivers are installed, no matter if devices are present or not. The most recent hardware and drivers will be automatically used. The same executable will continue to run on newer version of library and plugins, even if new completely different interfaces were introduced, and will work on devices not yet released at application build time. In addition the used component architecture allows for future code portability to other platforms.
Altought Haptik Library is not intended to replace high level libraries, its usage is thus advisable:
- To add haptic rendering to already existing applications
- To develop high-end visio/haptic applications or research applications that need an own graphic engine
- For developers who want an easy but powerful low-level access to devices

Conclusions and Acknowledgments

Current version of Haptik Library is being successfully used at the University of Siena, both as a teaching tool in haptic related classes and for high-end visio-haptic demos development, by students and researchers. We hope to see it spread after the EuroHaptics presentation. Obviously enough the Haptik Library is a completely open project and we hope to see feedback and cooperation from interested researchers and manufacturers to obtain a widely agreed specification of the common interfaces. We aim to a scenario where each manufacturer designs a custom interface for its devices, and ships, besides its own technologies, a related plugin, while the whole haptic community defines common interfaces and protocols. This research was partially supported by the italian MIUR and by Fondazione MPS.

References

1. Sensable Technologies - http://www.sensable.com
2. Force Dimension - http://www.forcedimension.com
3. ReachIn - http://www.reachin.se
4. The e-touch project - http://www.etouch3d.org
5. F. Conti et Al: "The CHAI Libraries" in Proc. EuroHaptics2003
6. OMG: "CORBA: Common Object Request Broker Architecture" tech report 1999
7. Microsoft Corporation: "Microsoft Developers Network" - http://www.msdn.com
8. S.Williams et al: "The Component Object Model: A Technical Overview" MSDN
9. M. de Pascale: "CMOS: A Component Model for Operating Systems" Thesis 2002