Overview of current developments in haptic APIs

Petr Kadleček
Supervised by: Petr Kmoch

Charles University in Prague
Faculty of Mathematics and Physics
Prague / Czech Republic

Abstract

Haptic technology as a key part of human-computer interaction allows us to use sense of touch in virtual reality by kinesthetic feel using force feedback. Increased production of haptic devices in recent years supported the development of many tools and libraries for programming applications with support of haptics. This paper introduces haptic technology and focuses on comparison of haptic application programming interfaces, especially on open-source and cross-platform solutions. We present different types of abstraction layers used in haptic APIs, basic haptic rendering methods and effects as well as a general overview of design concepts used in selected APIs. CHAI 3D haptic library is analyzed in more detail.

Keywords: haptic technology, human-computer interaction, haptic rendering, CHAI 3D, H3D API

1 Introduction

Kinesthetic sense provides us with information about movement and position of our body parts in the environment. We are able to feel various forces in different directions and use this information to determine the size, shape and other characteristics of objects we touch and forces they exert. Haptic modality of human-computer interaction utilizes sense of touch which is generally incorporating hands, upper torso, head and other parts of the body. The purpose of a haptic device is to generate force feedback of a given direction and magnitude in a specified workspace and send the position of a control part of the apparatus to the computer. One of the most valuable applications of haptic devices is in medicine (simulations of surgical operations, teleoperation, virtual palpation [5]). Haptic devices are also valued as assistive technology for visually impaired or blind people [7]. Other applications can be found in military, painting, CAD systems and gaming.

Haptic devices can be generally divided by the dimension of an orientation ability called degrees of freedom (DOF). That is basically translation (3-DOF) and translation combined with rotation (6-DOF). A typical example is a movable grip for 3-DOF devices (e.g. Novint Falcon shown in Figure 1) and a pen on a pivot with the ability to rotate and translate in all three dimensions for 6-DOF devices. There are also 6/3-DOF devices that combine 6-DOF positioning and 3-DOF force feedback (e.g. PHANToM Desktop shown in Figure 1). 7-DOF devices have a scissors snap-on, a thumb-pad or any other extra grip.

Figure 1: PHANToM Desktop (on the left) and Novint Falcon (on the right)

Common comparable properties which can be found in technical specifications of haptic devices include: *workspace* specifying a maximal reach of a touch tool (often measured in inches) and maximal rotation abilities if appropriate, *position resolution* of a touch tool measured in dots per inch (DPI), *maximal force* specified in newton unit or as a force capability in kilograms or pounds and *stiffness* of a haptic device along a degree of freedom measured in newtons per metre.

While force feedback gives a sense of force or generally a kinesthetic feel, tactile and touch sensing is used when one wants to feel pressure, heat or fine textures (and any other sensation felt by the skin). Haptic devices do not usually provide cutaneous sensation and it should not be confused with kinesthetic feel. Technology prototypes using both kinesthetic and tactile feedback have been proposed [6].

Although the sense of touch is not as acute as hearing, its accuracy is somewhere in between sight and hearing. Humans need approximately 1000 Hz frequency of haptic feedback to achieve smooth force perception. If the frequency is smaller than 1 kHz, a haptic stimulus felt by organs of human kinesthesia is unrealistic and may even lead

to system instability, potentially causing injury or damaging the device as stated in [11]. This means that a haptic loop has to be at least 30 times faster compared to minimal real-time computer graphics rendering rates, which demands great optimizations in haptic applications.

The remainder of this paper is organized as follows: In Section 2, we discuss different abstraction layers of haptic APIs. Basics of haptic rendering algorithms, haptic effects and other extensions are presented in Section 3. Sections 4, 5 and 6 are devoted to overview of CHAI 3D, HAPI, H3DAPI and other haptic APIs. The paper concludes with a table of haptic APIs specification and benchmark.

2 Abstraction layers of Haptic APIs

There are various methods of implementing haptic device control into an application ranging from the lowest driver layer to the highest scene graph layer. The most important decision a software architect has to make is a choice of the particular abstraction layer (shown in Figure 2) at which the rest of the application communicates with haptics.

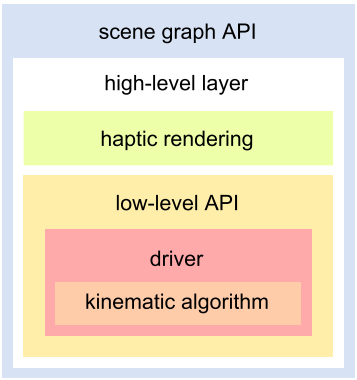

Figure 2: Abstraction layers of haptic APIs

2.1 Driver layer and kinematic algorithm

The lowest layer at which the programmer can communicate with the device is a driver of the operating system. At this layer the driver receives raw data through a serial bus (e.g. USB, IEEE 1394) from encoders that has to be processed with kinematics algorithms to get the data that corresponds to a three-dimensional vector of the haptic tool position in Cartesian coordinates. Kinematic algorithms are often a part of the driver because of specific technical specifications of every device. Manual initialization, opening and closing communication with the device or an inverse kinematics algorithm which computes force data in the application and sends it to the device to compute angles at haptic device joints is also essential. To preserve a smooth haptic response thread handling has to be done. For this reason, an extra haptic thread which calculates physics in the application is necessary.

The driver layer provides the fastest and the most precise response but demands a great effort to get the device working. Support of any other haptic device that has no compatible communication protocol means rewriting a lot of source code.

Manufacturers of haptic devices often provide optimized and well documented drivers in the C or C++ programming language. There are also open source and cross platform drivers that can provide support in officially unsupported operating systems such as Linux or Mac OS.

2.2 Low-level API

While the driver layer communicates in raw data, a low-level API hides the kinematics algorithm implementation from the programmer and allows developers to work directly with position, rotation and force vectors in the application. Many low-level APIs works as a common interface for different drivers which is very helpful when supporting a lot of haptic devices. A device handler is then used for getting information on haptic devices available on the current machine. Reading information from haptic devices may be blocking or non-blocking. Blocking servo loop callback stops the application thread where the function was called and reads the data at the frequency of haptic interface servo loop. A typical haptic application using low-level API is presented in algorithm 1.

Algorithm 1 Application using low-level haptic API

```
Initialize haptic device handler
number of haptic devices ← haptic device handler
if number of haptic devices = 0 then
    Exit
end if
specification of haptic devices ← haptic device handler
Initialize specified haptic device(s)
while Simulation running do
    position ← haptic device {blocking or non-blocking}
    compute force {haptic rendering}
    force → haptic device
end while
```

2.3 High-level layer

A graphical and haptical data representation of a model may be very similar or sometimes even identical. Integration of graphics and haptics into one API is therefore reasonable. There are several different approaches to create high-level API. One of the most intuitive way of incorporating haptics into application is based on calling similar functions that are provided in OpenGL graphics library.

A layer which handles computation of forces for a given model is called a haptic rendering layer. We describe it in more detail in the next section.

2.4 Scene graph API

A scene graph haptic API often uses a tree structure of objects in the virtual world with a specific root node such as a world node. It is possible to apply graphical and haptical properties to an object and set the specific property recursively to its child objects.

A scene graph API often includes low-level APIs for haptics, graphics, physics and audio processing. It provides all the features of low-level APIs and even more by combining them together. Haptic and graphic rendering is essential in the scene graph API oriented on haptics.

The concept of combining low-level APIs into one often creates many drawbacks which the high-level scene graph API implementation may or may not hide from the programmer. Difficulties connected with such a combination of different APIs may result in a thorough problem analysis that may not even be solvable with a feasible effort because the API itself may be proprietary and authors may not support the API any more.

A scene graph haptic API is the best choice for prototyping an application when the speed of development is crucial and performance is not a priority. Support of a scripting language or standard file format representation of a scene helps even more with rapid development.

3 Haptic rendering

One of the most important algorithmic problems associated with haptics is computation of interactions between the haptic tool and virtual objects. Creating a convincing force reaction on a complex object is a nontrivial task that is dependent on data representation. The technique of haptic interaction processing in the virtual scene is called haptic rendering (or haptic display). As in graphic rendering, where the image is composed from a model based on a virtual camera position, the process of haptic rendering returns a force on the basis of a model with which the haptic tool interacts. Creating a good haptic rendering algorithm is a struggle to maintain realistic force feedback without using cumbersome computations which raise memory and CPU requirements.

There are basically two accepted standard methods that are implemented in high-level haptic APIs for 3-DOF haptic rendering: God-object method by Zilles et al. [15] and Virtual proxy method by Ruspini et al. [11]. It should be noted that even though there are many articles concerning 6-DOF haptic rendering, there is no standard widely-used implementation.

The maximal stiffness capability along any degree of freedom is limited on every haptic device. Therefore, a user may move a haptic tool with a force which lets them penetrate into a rigid body or any kind of object. Haptic rendering algorithms are trying to solve this problem by exerting an adequate force that is pushing a haptic tool away from the object.

3.1 Penalty based methods

The simplest type of haptic rendering technique specifies a force vector for every point in a scene by calculating the nearest resting position of a haptic tool also represented as a point. If the haptic interface point is outside the object, the resulting force is zero, otherwise the force vector has a magnitude proportional to the penetration distance. This kind of method is also called *vector field method* [15] or *penalty based method* [11]. This technique, however, has many drawbacks which make it useless for at least plausible simulations. As this method does not save a history of haptic interface point movement, discrete space of haptic servo updates may result in unnoticed penetration through an object in one haptic loop step as shown in Figure 3 on the left. Another pop-through problem may come up when penetration is too deep and the desired nearest resting point is on the other side of the object as shown in Figure 3 on the right.

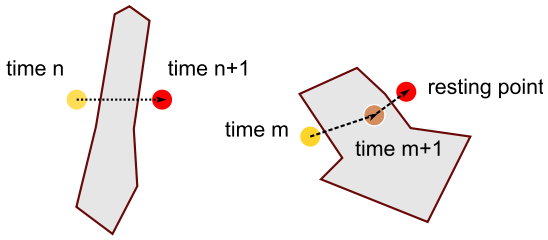

Figure 3: Pop-through problems with penalty based methods

3.2 God-object method

To solve pop-through problems mentioned in penalty based methods a God-object method was proposed [15]. The God-object represents a virtual point in the scene that is not able to penetrate into rigid bodies and thus behaves correctly. A position of the God-object is updated in every haptic loop step.

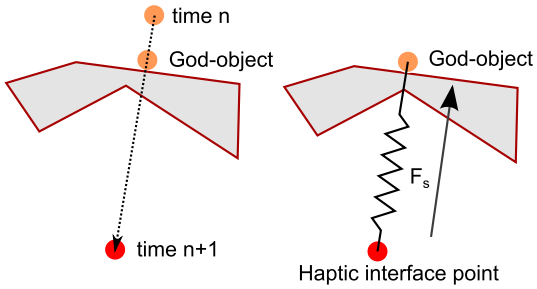

Figure 4: God-object

If the haptic interface point (HIP) penetrates into an object, the movement of god-object towards HIP is constrained by a surface of this object and the resulting force is calculated by simulating an ideal mass-less spring (shown

in Figure 4) which is, according to Hooke's law, defined as follows:

$$F_s = -k\Delta x = -k(x_{\text{HIP}} - x_{\text{GodObject}}) \quad (1)$$

where Δx is a displacement of spring and k is a spring constant defining the stiffness of the surface.

The God-object method can be easily extended [14] to support static and dynamic friction on rigid bodies which is essential to achieve realistic haptic stimulus. Haptic shading, an analogous algorithm to Phong shading can be applied on force feedback on surface normals to create an effect of smooth surface. Another association to computer graphics is in the use of textures. A haptic texture mapped on the object can be used to simulate different kinds of materials such as wood, stone or metal. Realistic haptic texture rendering has been investigated in [3].

3.3 Virtual proxy method

Polygonal meshes often contain small surface gaps because of low-quality digitization or non-precise modeling. When the god-object enters a mesh through a small gap, the user gets stuck inside the mesh until he finds the gap again. To resolve the problem we either fill in small gaps in the process of loading the mesh or we set a radius of the god-object in collision detection with constraint planes. The Virtual proxy method [11] proposes to treat a presentation of the haptic tool in the virtual environment as a sphere (as shown in Figure 5). Extensions discussed in God-object method are applied simply by moving the proxy and thus changing the resulting force.

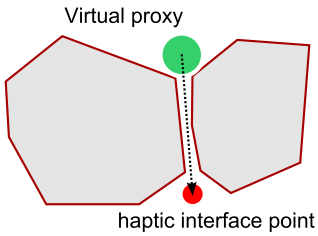

Figure 5: Virtual Proxy

In the remainder of this paper, we will examine several common haptic APIs in more detail.

4 CHAI 3D

CHAI 3D [4] is a scene graph API written in the C++ programming language with aim to create a modular, open source and cross platform haptic API with a wide support of different haptic devices (and a virtual device working on Microsoft Windows platform). CHAI 3D is licensed under GNU General Public License (GPL) version 2 but also offers a Professional Edition License. The main reason to create CHAI 3D was that all available APIs developed by manufacturers of haptic devices were proprietary and supported only the one specific device or a group of devices from the manufacturer.

The scene graph capabilities of CHAI 3D mainly focus on haptics combined with graphics. It does not include any extra visual or sound effects but it does propose lightweight and compact functionality. CHAI 3D is definitely not the API with tons of functions ready for the implementation of sophisticated applications. It is rather the API for academic and research use where the extra functionality can be easily added.

Though the API manual or tutorials do not yet exist, the source code is very well documented and is very easy to read and scan through. The reference guide generated by a Doxygen documentation system could serve as a quick guide over the source code but it is not a comprehensive source of learning CHAI 3D. Authors of CHAI 3D recommend to learn by the examples in packages for different platforms. This method gives the learner a decent overview of the API but does not allow to fully understand some fundamental characteristics of the API which makes the learner read part of the API source code eventually.

4.1 Low-level use of API

Though the CHAI 3D library is a scene graph API, use of CHAI 3D as a low-level communication layer is convenient. CHAI 3D provides support of many devices and an easy to use device handler *cHapticDeviceHandler*. Every device is then treated as a generic haptic device *cGenericHapticDevice* with basic ability to get a position, set a force, device communication opening, initialization and closing.

4.2 CHAI 3D scene graph

A scene graph of CHAI 3D contains standard shapes, meshes, virtual cameras and lights. The main unit of all objects in the scene graph is a *cGenericObject* class which inherits from a general abstract type *cGenericType*. The generic object creates a tree structure of objects using a standard template vector class of children objects in a *m_children* member. All methods for object modification or property setting allow propagation to children by setting an optional function parameter *a_affectChildren*, which is by default set to false. CHAI 3D scene graph has one root node class for every object in the scene called *cWorld*. This class is essential for further communication with graphics and haptics.

The API contains only three standard object shapes (two implicit surface objects): sphere (*cShapeSphere*) defined by a radius, torus (*cShapeTorus*) defined by an inside and an outside radius and line (*cShapeLine*). Beside standard shapes implemented in CHAI 3D API, it is possible to load complex meshes in OBJ and 3DS formats.

4.3 Haptic tool

The scene graph representation of a haptic device is called a tool. An abstract class defining all tools in the scene graph is *cGenericTool*. The only specific tool that CHAI 3D provides at this time is a 3-DOF tool identified as a *cGeneric3dofPointer*. 6-DOF force rendering algorithms are not supported.

The generic tool is also a generic object which means that the tool has its position, rotation and all other object properties. The tool itself needs only a pointer to the haptic device from a device handler. It manages all the initialization automatically by calling a *start* method. A *stop* method does the opposite.

The default device mesh of the generic 3-DOF pointer displays the tool as a sphere. God-object algorithm with variable radius is used for the haptic force rendering for which there are two meshes representing the tool:

- a device mesh (*m_deviceMesh*) which represents the real current position of the haptic device touch tool
- a proxy mesh (*m_proxyMesh*) which represents a model of the haptic interface in the virtual environment

The force model is also defined as the abstract model (with a generic class *cGenericPointForceAlgo*) split into *cProxyPointForceAlgo* and *cPotentialFieldForceAlgo* classes. The *cProxyPointForceAlgo* class implements the God-object method and *cPotentialFieldForceAlgo* class process local interaction relating to haptic effects.

An overall force contains assigned local haptic effects and interaction forces computed on the base of haptic device properties (e.g. stiffness), a position relative to an interaction projected point on the interacting object surface and a best new position of the proxy model in the proxy point force algorithm. Interaction detection is not always precise especially in complex meshes and the proxy model gets sometimes stuck and generates excessive force.

The tool works in a workspace set by a radius. It is possible to change the radius and position of the workspace and its rotation relative to the scene. The tool is often attached to the camera so that the workspace corresponds to the view of the camera. A schema of haptic tool interaction process is shown in Figure 6.

4.4 Haptic effects

The CHAI 3D scene graph provides a set of haptic effects that can be assigned to implicit surface objects [12]. These effects are computed using a local interaction *computeLocalInteraction* method of each object. The mesh or any other complex object without overridden *computeLocalInteraction* method is not able to apply haptic effects because there is no way how to compute an interaction projected point from a generic object algorithm. Only the proxy point algorithm is used for these objects to calculate forces.

Haptic effects with the base abstract class *cGenericEffect* in the API are as follows:

- Magnetic model effect *cEffectMagnet* provides a magnetic field effect near the object
- Stick-slip effect *cEffectStickSlip* provides an effect of sliding one object on another with sticking caused by friction (e.g. rubber on a desk)
- Surface effect *cEffectSurface* provides a basic surface effect of a tool pushing against the object
- Vibrations effect *cEffectVibration* provides an effect of a vibration with a specific frequency and amplitude
- Viscosity effect *cEffectViscosity* provides an effect of a tool moving through a fluid

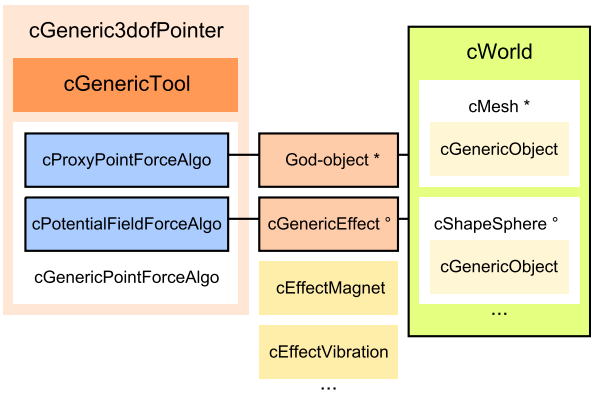

Figure 6: Schema of a haptic tool interaction process in CHAI 3D - effects can be applied only on implicit surface objects, God-object method is used for mesh objects in cWorld, as denoted by asterisk and circle

All effects are very sensitive to a good setting of properties such as a maximal stiffness of the haptic device. A relatively small change of effect properties can make a great difference in the effect perception and sometimes even a different driver may result in a different effect behavior.

4.5 ODE module

The CHAI 3D library does not implement its own rigid body dynamics simulation. There is, however, a module that connects the CHAI 3D scene graph with the Open Dynamics Engine (ODE) library.

Communication of CHAI 3D and ODE is handled by *cODE*, *cODEWorld* and *cODEGenericBody* classes. The API contains precompiled ODE libraries for both dynamic and static linking with double precision. Preprocessors definitions need to be set correctly in order to run an application properly without runtime errors.

Every object in the ODE simulation has to be added to a specific ODE world. Such an object is defined as an ODE generic body with properties of physical simulation and a CHAI 3D body image model of the scene graph. The ODE world is a generic object which behaves as a child object in the standard parent world object but has a list of *bodies* instead of a list of *children*. However, all recursive algorithms in CHAI 3D look up *children* list in the scene graph. For instance, it is therefore not possible to assign a haptic effect to an object in the ODE simulation because the rendering algorithm is using the mentioned recursion through children list. A fix of this behavior can be found in [10].

The ODE module enables creation of a dynamic box, sphere, capsule and a mesh from an assigned CHAI 3D body image model. Static planes are also available. A global gravity can be set as a three-dimensional vector describing a force. Calling an ODE world *updateDynamics* method with a step time function parameter updates the simulation. Though the implementation of dynamics into the scene graph is simple, a programmer still has to work with the ODE world as a separate world and encounters a lot of disadvantages when using recursive scene graph algorithms.

4.6 GEL module

The haptic technology utilizes an implementation of a deformable body simulation more than any other technology. CHAI 3D provides a module to create such deformable objects in the scene graph which uses the GEL dynamics engine developed at Stanford University.

As in the ODE module, the GEL module is implemented as a separate world (*cGELWorld*) of deformable objects. The main idea behind the deformation is a skeleton model made of nodes (*cGELSkeletonNode*) and links (*cGELSkeletonLink*) between them. Nodes are represented as spheres with a given radius and mass connected with elastic links with spring physics defined by elongation, flexion and torsion properties (as shown in Figure 7). Every node has its physical properties (linear damping, angular damping, gravity field definition) and provides methods to control force and torque.

The GEL module provides a simple way to add deformable objects to the scene graph, but integration of the GEL dynamics engine in the lower layer of the scene graph with automated skeleton modeling would considerably enhance the high level use of CHAI 3D.

5 H3D API and HAPI

H3D API [1] is a high level scene graph API developed by SenseGraphics. H3D API uses HAPI as a low-level layer for haptics, OpenGL for graphics and the X3D XML-based file format to represent the scene. The library is written in the C++ programming language and is licensed under GNU GPL v2. Closed source license for commercial use is also available.

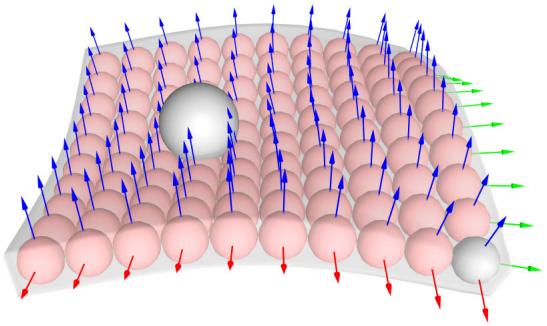

Figure 7: CHAI 3D GEL module example

5.1 X3D

The most interesting feature H3D API provides is scene definition in X3D file format. The whole scene with a camera set, lights, primitive objects, complex meshes, textures, etc. is defined as XML nodes. As X3D is originally web-based technology, a texture or any other object loaded from a file can have a URL path.

The haptic device is defined through a *DeviceInfo* node with the haptic renderer specification, position calibration and the proxy model appearance. H3D API implements all HAPI haptic rendering functionality to the X3D specification. For instance, to add a frictional surface effect to the shape in the scene, a XML node *FrictionalSurface* is added to the appearance node of the shape with appropriate properties.

H3D API also supports X3D routes which makes it possible to read data from one source and route it to a specified destination. That is for instance routing the position of the mouse from the *MouseSensor* node to the shape node position. A *PythonScript* node allows to route data from X3D to Python programming language functions.

5.2 Python interface

H3D API propose a very unique way of haptic programming using Python scripts on top of the X3D scene definition. A Python interface to the H3D API implements X3D creation and write functions, special bindable node access (haptic device info, viewpoint, etc.) and X3D field types so that it is possible to create a comprehensive application just using the X3D and Python when there is no reason to develop efficient real-time application.

5.3 Scene graph and C++

H3D API is not only the Python and X3D. The entire application can be written in the C++ programming language

for better performance. The C++ code allows to parse X3D strings which makes it easier to create objects or set materials in C++. This method should be used only in initialization of the scene because real-time X3D parsing in a graphics loop of the application would lower the performance.

H3D API is a perfect tool to create fast prototypes of applications using haptics. Python and X3D is available for a very rapid development and C++ for higher performance applications.

5.4 HAPI

HAPI [1] is a new complex open source high-level haptic API also developed by SenseGraphics licensed under GNU GPL v2. As with the H3D API, a closed source license is also available. HAPI is written in the C++ programming language and works on all major operating systems: Microsoft Windows, Linux and Mac OS.

HAPI is one of the most active haptic APIs supporting devices from Sensable, Force Dimension, Novint and Moog FCS Robotics. There are four haptic rendering algorithms available: God-object algorithm (described in Section 3), Ruspini algorithm - Virtual proxy method, CHAI 3D rendering, OpenHaptics rendering.

HAPI provides not only the basic device handling, but there is also a number of haptic force effects, surface effects, collision detection, primitive shape creation and thread handling. A very specific functionality is graphics rendering based shape creation. It allows a programmer to create haptic shapes using standard OpenGL drawing functions. A *FeedbackBufferCollector* class collects all triangles that are rendered via the OpenGL library.

HAPI is very well documented with an accompanying manual, reference manual generated by Doxygen documentation system and a lot of examples of all features. The source code of a basic device handling application written in HAPI using the *AnyHapticsDevice* class has just about 20 lines. HAPI can be downloaded as a Windows Installer or as the source code.

The manual and examples make HAPI very easy to use. HAPI is one of the best choice of commercial and non-commercial high-level APIs with a very good support from authors and can be also used as a low-level API.

6 Other haptic APIs

OpenHaptics[13] is a commercial software development toolkit designed for SensAble devices. The toolkit contains scene graph API for rapid development, high-level and low-level APIs and support for integration of haptics into existing applications. OpenHaptics is also available in Academic Edition.

There are many low-level APIs designed for specific devices: HDAL [8] (Novint Haptic Device Abstraction Layer) which is a commercial closed source SDK

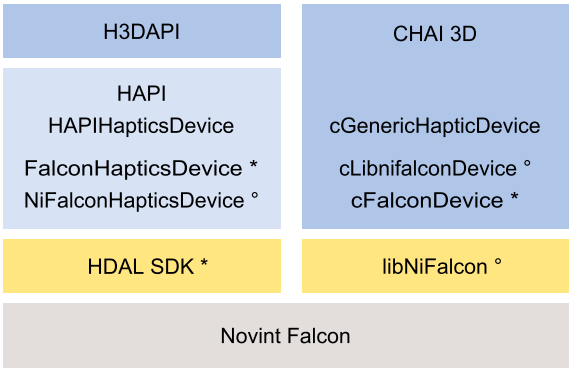

Figure 8: Haptic API abstraction layers for Novint Falcon. HDAL SDK wrapping classes are denoted by asterisk, libNiFalcon wrapping classes are denoted by circle

for Novint Falcon device working only on Microsoft Windows, libNiFalcon [9] - an open-source driver for Novint Falcon working on all major platforms or JTouchToolkit [2] (HDAL SDK and OpenHaptics HDAPI/HLAPI wrapper for Java platform). Example of haptic API abstract layers for Novint Falcon device is shown in Figure 8 (experimental implementation of libNiFalcon into CHAI 3D is a part of [10]).

Conclusion

We have introduced haptic technology and discussed aspects of programming with haptics. We have shown that there are many ways how to add support of haptic technology into an application using different abstraction layer of haptic APIs varying from haptic device driver, low-level APIs to high-level scene graph APIs. We have presented basic methods of 3-DOF haptic rendering - specifically the God-object method and Virtual proxy method which are used in high-level APIs such as CHAI 3D, HAPI or OpenHaptics. Relevant parts of CHAI 3D haptic library have been analyzed in detail including haptic tool, haptic effects or ODE and GEL module support. Very active haptic APIs HAPI and H3D API have also been analyzed. H3D API brings a possibility to create haptic applications in declarative programming language X3D with an interface to Python programming language. Another commercial device specific haptic APIs were mentioned such as HDAL SDK or OpenHaptics.

Final comparison of haptic APIs is given in Table 1. Computational load benchmark has been implemented in selected low-level haptic APIs as a simple force field haptic rendering algorithm. Benchmark ran on Intel Atom 330 1.6 GHz dual core CPU and results are shown in figure 9. Some APIs do not support blocking servo calls which make fast haptic rendering algorithms (under 1ms) to create pointless CPU load by polling data in loop. This

API	CHAI3D	HDAL SDK	JTouchTool.	libNiFalcon	HAPI	H3D API	OpenHaptics
Open source	●	○	●	●	●	●	○
Cross platform	●	○	*	●	●	●	●
License	GPL/C	C/N	GPL	BSD	GPL/C	GPL/C	C/A
Development state	●●○	●●●	●○○	●●○	●●●	●●●	●●●
API manual	○	●	○	○	●	●	●
API reference	●	●	●	●	●	●	●
Device range	●●●	●○○	●●○	●○○	●●●	●●●	●●○
Abstraction layer	High/Low	Low	Low	Low/Driver	High/Low	High	High/Low

Table 1: Haptic APIs comparison, C = commercial, N = non-commercial, A = academic, * = partial

behavior is taken into account and some APIs were extra benchmarked with simple polling prevention.

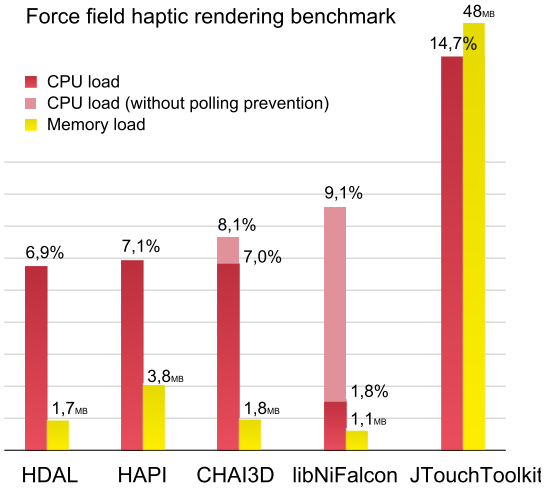

Figure 9: Benchmark based on simple haptic rendering algorithm simulating force field with ideal spring defined by Hooke's law

Acknowledgements

I would like to thank Petr Kmoch for his support and advice throughout the creation of this work.

References

[1] SenseGraphics AB. *H3D API - haptics software development platform*, 2011. `http://www.h3dapi.org/`.

[2] John Archer. *JTouchToolkit*, 2008. `https://jtouchtoolkit.dev.java.net/`.

[3] S. Choi and H.Z. Tan. Toward realistic haptic rendering of surface textures. In *ACM SIGGRAPH 2005 Courses*. ACM, 2005.

[4] Conti Francois et al. *CHAI 3D set of libraries*, 2009. `http://www.chai3d.org/`.

[5] Williams II et al. The virtual haptic back for palpatory training. In *Proceedings of the 6th international conference on Multimodal interfaces*, pages 191–197. ACM, 2004.

[6] M. Fritschi, M.O. Ernst, and M. Buss. Integration of Kinesthetic and Tactile Display–A Modular Design Concept. In *Proceedings of the EuroHaptics*, 2006.

[7] J.P. Fritz, T.P. Way, and K.E. Barner. Haptic representation of scientific data for visually impaired or blind persons. In *Proceedings of the Eleventh Annual Technology and Persons with Disabilities Conference*. CSUN, 1996.

[8] Novint Technologies Inc. *HDAL - Novint Falcon SDK*, 2008. `http://home.novint.com/products/sdk.php`.

[9] Kyle Machulis. *libNiFalcon - open source driver for the Novint Falcon*, 2009. `http://qdot.github.com/libnifalcon/index.html`.

[10] Kadleček Petr. A Practical Survey of Haptic APIs, Bachelor's thesis, Charles University in Prague, Czech Republic, 2010.

[11] Diego C. Ruspini, Krasimir Kolarov, and Oussama Khatib. The haptic display of complex graphical environments. SIGGRAPH '97, pages 345–352, 1997.

[12] Kenneth Salisbury and Christopher Tarr. Haptic rendering of surfaces defined by implicit functions. In *Proceedings of the ASME 6th Annual Symposium*, pages 61–68, 1997.

[13] Sensable. *OpenHaptics software development toolkit* , 2011. `http://www.sensable.com/products-openhaptics-toolkit.htm`.

[14] C. B. Zilles. Haptic Rendering with the Toolhandle Haptic Interface. Master's thesis, Massachusetts Institute of Technology, 1995.

[15] C. B. Zilles and J. K. Salisbury. A constraint-based god-object method for haptic display. IEEE Computer Society, 1995.