Visual Debugger for Single-Point-Contact Haptic Rendering

Christoph Fünfzig[1], Kerstin Müller[2], Gudrun Albrecht[3]

[1] LE2I–MGSI, UMR CNRS 5158, Université de Bourgogne, France
[2] Computer Graphics and Visualization, TU Kaiserslautern, Germany
[3] LAMAV, FR CNRS 2956, Université de Valenciennes et du Hainaut-Cambrésis, France
Christoph.Fuenfzig@u-bourgogne.fr
Kerstin.Mueller@cs.uni-kl.de
Gudrun.Albrecht@univ-valenciennes.fr

Abstract: Haptic applications are difficult to debug due to their high update rate and many factors influencing their execution.
In this paper, we describe a practical visual debugger for single-point-of-contact haptic devices of impedance-type. The debugger can easily be incorporated into the running haptic application. The visualization shows the position trajectory with timing information and associated data like goal positions and computed feedback forces. Also, there are several options for in detail analysis of the feedback force applied at each time instance. We show with several use cases taken from practical experience that the system is well suited for locating common and intricate problems of haptic applications.

1 Introduction

Haptic applications have two characteristics. They are interactive with a human user in the loop, and they have realtime requirements as they operate at a 1kHz rate. Both make these applications difficult to debug and difficult to compare. Problems of a specific haptic rendering algorithm might occur only for certain input sequences and geometric configurations.

Concerning the device type, we work with an impedance-type haptic device and a single point of contact between the haptic probe and the rendered object. Impedance-type haptic devices measure the endpoint motion (position or velocity) and output a force in response. Using the opposite causality, admittance-type devices measure the applied force and output a motion according to the virtual environment being rendered [HM07]. Examples of impedance-type are shown in Figure 1, the SensAble Phantom Omni and the NOVINT Falcon. In our experiments, we have used the NOVINT Falcon parallel device. It has a $(4\,\text{inch})^3$ (approx. $(10.16\,\text{cm})^3$) workspace with $2\,\text{lb}$-capable (approx. $8.9\,N$) actuators and $400\,\text{dpi}$ (approx. $157.48\,\text{dpcm}$ resolution) sensors. The standard procedure for

[1]The first author would like to thank the Conseil Régional de Bourgogne for support in a postdoc scholarship of the year 2008/2009.

debugging consists of writing out data files of trajectories and force feedbacks. Standard visualization packages (like Matlab) might be employed for visual examination. But due to the tight coupling of 3D position data and the generated 3D force feedback, this approach usually does not provide enough insight. In order to ease development and analysis, we developed a customized graphical debugging tool for haptic applications. The tool can be integrated into existing haptic applications at the C/C++ source code level. Such an in-system debugger reduces the turnaround time and is very comfortable to use.

We will not cover haptics with several contact points and admittance-type devices here. The extension to several contact points is very natural, and occurs with devices having multiple physical contact points, like data gloves, or devices with a single contact point using virtual contact shapes, like the Phantom Premium 6DOF. Admittance-type devices are less commonly used as they are more expensive to build. Devices of this type need a continuous control unit to even hold position in freespace areas. Whereas impedance control takes no action in freespace areas but applies maximum forces for stiff walls. These devices therefore do not need a continuous control unit.

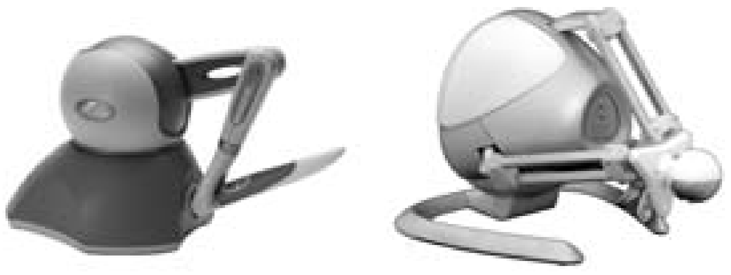

Figure 1: Haptic Devices: the SensAble PHANTOM Omni® is a serial 6DOF input, 3DOF output device (http://www.sensable.com), the NOVINT Falcon® is a parallel 3DOF input/output device (http://home.novint.com/products/sdk.php). Photos are courtesy of SensAble and NOVINT.

2 Related Work

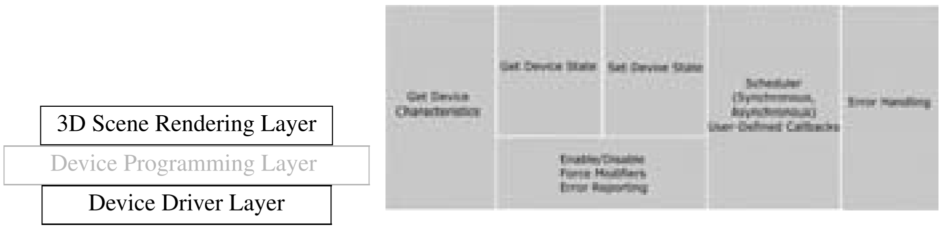
3D Scene Rendering Layer
Device Programming Layer
Device Driver Layer
Get Device Characteristics
Get Device State
Set Device State
Enable/Disable Force Modifiers Error Reporting
Scheduler (Synchronous, Asynchronous) User-Defined Callbacks
Error Handling

Figure 2: Haptic system's software structure. SensAble's HDAPI (right, [Sen05]) and Novint's HDAL ([Nov08]) are similar but are not source-code compatible.

The haptic system's software is commonly structured in layers as shown in Figure 2. The abstraction increases from bottom to top. Basic functionality is available in the *Device Programming Layer*, which consists of haptic thread functions, device state query, and device state setting [Nov08, Sen05].

Most works cover the performance aspect of haptic applications. About comparing and benchmarking haptic rendering algorithms, also some work is available. In [RBC05], a common software framework is proposed which normalizes all factors, on which an haptic application depends. They formalize the notion of one haptic algorithm being faster than another. Ruffaldi et al [RME$^+$06] add physical ground truth to this comparison. They measure the geometry of a physical object, and measure an input sequence with force responses to create a database of ground truth data. Haptic rendering algorithms are then compared by their simulated forces on input sequences taken from this database. The thesis [Cao06] also aims at benchmarking haptic applications. It describes the design of a simulation module, which is able to generate reproducible position input sequences to feed into the haptic algorithm under analysis. Several input models are presented that vary in the required user inputs, like path-based model (recorded space curve), function-based model (space curve defined by functional sections) and adaptive model (curves filled inbetween penetration points). The author shortly mentions an analysis module, which is intended for the visualization of the acquired data but details of the visualization are not available.

3 Data Acquisition for Debugging

For debugging, we need to know all device variables in the workspace: position $x_d(i)$ (or velocity), and the device output force $f_d(i)$. Additionally, it is helpful to know the force semantics in the simulated environment. This force usually results from a distance to a goal position $g(i)$ or a penetration depth with respect to a surface contact point (SCP) $g(i)$ (Figure 3). All device variables occur as sequences over $i \in \mathbb{N}$. In the following, we omit the variable subscripts.

Depending on the computation time for the virtual environment simulation, the measurement $\{x(i), f(i), g(i)\}$ occurs at a certain point in time $t(i)$. The sampling time $t(i) - t(i-1)$, for $i \in \mathbb{N}$, is about 1ms.

We store the measurements in a ring buffer of fixed size n, which contains all measurements in a certain time interval $[ts = t(j), te = t(j+n)]$. This storage organization is fixed size and fast enough so that a single measurement of size 10 doubles (3 for position, goal, force each and 1 for the corresponding time value) can be stored away without changing the simulation timings significantly. Furthermore, note that the time interval is irregularly sampled, and the interval width $te - ts$ can vary. This is the case because the sample times are given by the force simulation in the virtual environment. The computation requires a varying time depending on query position and environment state. The device then exerts the last force handed to the API at a rate of 1kHz in the feedback loop (zero-order hold semantics).

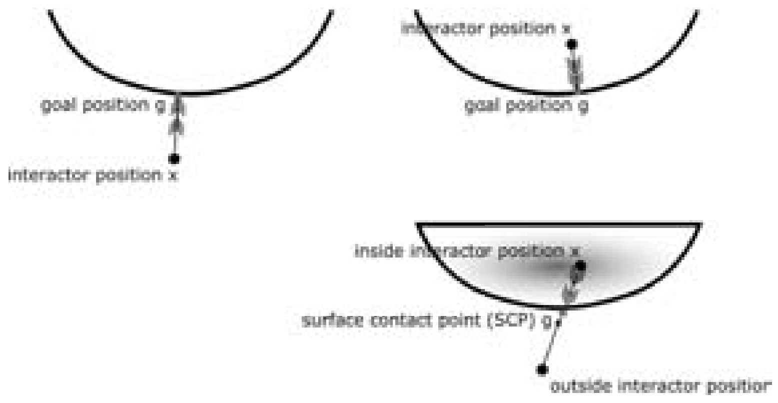
goal position g
interactor position x
interactor position x
goal position g
inside interactor position x
surface contact point (SCP) g
outside interactor positior

Figure 3: Haptic force employed for rendering. A spring force restores the interactor position to the goal position. For solid object rendering, this is called the surface contact point.

4 Data Visualization

For the data visualization, we want to achieve the following goals:

1. position and goal points are semantically connected, and this should be explicitly visualized,
2. forces apply to their corresponding interactor positions, and their direction and relative magnitude should be easy to catch,
3. time information is very important as many common application errors are due to false timing.

From the position sequence, we must compute derivative and second derivative in order to show velocity and acceleration on the curve. We compute them by a least-squares fit of a functional quadratic (Figure 4) to the three components of the position. We have experimented with the neighborhood half-size $s \in N$, $s > 0$, i.e., the number of data points before and after the current position used in the least-squares fit.

For goals 1 to 3, we have chosen options from a set of visualization styles. During result presentation, the debugger keeps a current time in its time interval $[ts = t(j), te = t(j + n)]$ of stored data. We render the position sequence x as a space curve connected by illuminated, colored line segments, as proposed by [ZSH96]. At the data points, we show the velocity magnitude there by the illuminance of the color. Explicit widgets to show the velocity direction are omitted as this is clear from the curve direction alone (Figure 5, left). For goals 1 and 2, we connect each position point with its corresponding goal point by a line segment and change from black to red color to show its direction (Figure 5, right). Similarly, we draw the force vector originating at each position point and change from black to green color to show direction (Figure 5, right). To reduce clutter, both goal positions and force vectors can be switched on and off, and only a time subinterval is shown. The user interaction to define a view on the trajectory is simplified in the way

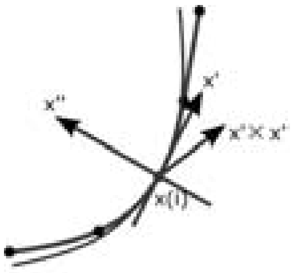

Figure 4: Least squares fit of a functional quadratic at time $t(i)$ to approximate velocity and acceleration of the position sequence. The neighborhood half-size used for the fit is $s = 2$ in this case. Note that the apex of the functional quadratic is always at time $t(i)$ but it does not need to pass through the point $x(i)$!

that the user looks at a current point t_c in time and rotates around the x- and y-axis of the device's coordinate system. This focus point in time must be moved forward or backward to an analysis spot by the user, but it strongly prevents spatial disorientation.

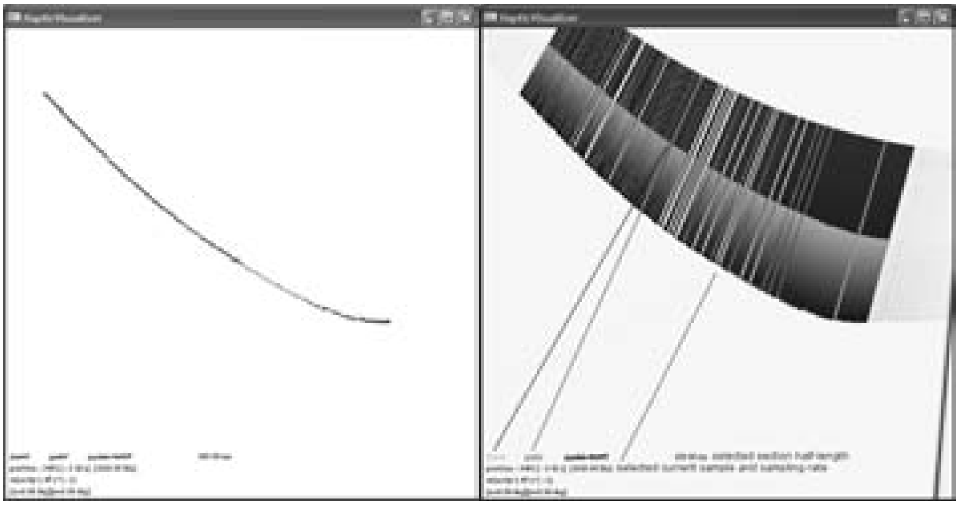

Figure 5: Visualization of a 200ms section of the position sequence shows the velocity magnitude at the samples and the time progression (from light to dark). Optionally, the goal positions (a minimum distance point on a cubic Bézier curve in this case) and the forces can be added to the visualization.

Problem 3 is how to show time on the curve given by x. At current time t_c, we show a time subinterval $[t_c - r, t_c + r]$ of length $2r$ on the curve in opaque style ($\alpha = 1$) and render the rest with a constant, large transparency $\alpha_{\min} \ll 1$. Because the human eye is not sensitive to a gradually changing transparency, we use this abrupt change of transparency intentionally. In order to show time and curve progression, the first half $[t_c - r, t_c]$ is rendered in violett color, and the second half $[t_c, t_c + r]$ is rendered in blue color.

5 Conclusion

In this paper, we presented a visual debugger for single-point-of-contact haptic systems. Our customized graphical debugging tool records the position trajectory and associated data like goal positions and feedback forces inside the running haptic system. Several options exist for the in-detail analysis of the data including the timing information. For a better turnaround time and an improved convenience, we built it as an in-system tool that can be integrated into the developed haptic application at the C/C++ source code level. With minor additions to the API (i.e., goal position), it is also possible to integrate it into the haptics programming environment below the application level. In our experiments, the tool has shown to be especially useful for analysing haptic rendering problems. Timing errors can be caused by position information acquired at a too low rate, or the haptic loop being too slow. Such defects can be seen inside the debugger from the timing information provided. When rendering a curve or surface with the haptics device, the desired behavior is a fast approach to the goal position, and a stiff but passive (energy diminishing) reaction to deviations from it. Sampling issues or stability problems can deteriorate the desired sensation. They result from too large forces at the available sampling rate. The debugger helps to spot this very common problem, and to resolve it by changing the spring and damping constants. Force continuity problems are usually caused by principle problems of the force-computing algorithm. They can be sensed at the device, and the debugger is able to mark suspicious points in the data stream graphically.

As future work, we want to extend the debugger to multiple-point-of-contact devices, in which case we have to additionally visualize orientation data and torques. Furtheron, an accurate model of the haptic device's dynamics could provide a detailed analysis by model-based prediction.

References

[Cao06] Xiao Rui Cao. A Framework for Benchmarking Haptic Systems. Master's final project thesis, School of Computing Science, Simon Fraser University, April 2006.

[HM07] V. Hayward and K.E. MacLean. Do It Yourself Haptics, Part-1. *IEEE Robotics and Automation Magazine*, (4):88–104, 2007.

[Nov08] Novint Inc. *Haptic Device Abstraction Layer (HDAL) Programmer's Guide*, 2.0.0 edition, Feb 2008.

[RBC05] Chris Raymaekers, Joan De Boeck, and Karin Coninx. An Empirical Approach for the Evaluation of Haptic Algorithms. In *WHC '05*, pages 567–568, 2005.

[RME$^+$06] Emanuele Ruffaldi, Dan Morris, Timothy Edmunds, Federico Barbagli, and Dinesh K.Pai. Standardized Evaluation of Haptic Rendering Systems. In *HAPTICS '06*, pages 33–41, 2006.

[Sen05] SensAble Technologies Inc. *OpenHaptics Toolkit Programmer's Guide*, 2.0 edition, Jul 2005.

[ZSH96] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Interactive Visualization of 3D-Vector Fields using Illuminated Streamlines. In *IEEE Visualization*, pages 107–113, 1996.