DEVAL – A Device Abstraction Layer for VR/AR

Jan Ohlenburg, Wolfgang Broll, Irma Lindt

Collaborative Virtual and Augmented Environments Department
Fraunhofer FIT
Schloss Birlinghoven,
53754 Sankt Augustin
{jan.ohlenburg, wolfgang.broll, irma.lindt}@fit.fraunhofer.de

Abstract. While software developers for desktop applications can rely on mouse and keyboard as standard input devices, developers of virtual reality (VR) and augmented reality (AR) applications usually have to deal with a large variety of individual interaction devices. Existing device abstraction layers provide a solution to this problem, but are usually limited to a specific set or type of input devices. In this paper we introduce DEVAL – an approach to a device abstraction layer for VR and AR applications. DEVAL is based on a device hierarchy that is not limited to input devices, but naturally extends to output devices.

Keywords: Device Abstraction, Input Devices, Output Devices, Virtual Reality, Augmented Reality

1 Introduction

Multiple input and output (I/O) devices are essential parts of VR and AR applications. They allow users to interact with the environment, modifying its state and properties and perceiving the results of the interactions. For VR/AR applications, a large variety of heterogeneous interaction devices exists. Among them are several tracking systems [9], 3D pointing and mouse-related devices, projection and personal display systems, tactile and force-feedback gloves, optical gesture and mimic recognition systems, speech recognition and synthesis systems as well as less common, application specific sensors and actuators [2].

There are several ways to integrate an interaction device into a VR/AR application. An application can directly connect to a device by sending data to or receiving data from an I/O device using the hardware drivers provided. While this approach offers a high degree of control over device-specific functionality, it couples the application tightly to the I/O device limiting its portability to different hard- and software setups. Additionally, it puts a huge workload to the developer, who would have to deal with an individual interface for each device. Thus, it is common practice for VR/AR applications to decouple the application from specific interaction devices via device abstraction layers.

Device abstraction layers describe sets of devices in abstract device classes. Typically, devices are standardized and structured based on the data types they provide or require (e.g. 3-DOF (degrees of freedom) vs. 6-DOF trackers). Applications that integrate devices via device abstraction layers do not need to be modified if a device is exchanged by a device of the same device type. Ideally, the application even does not need to be restarted but can seamlessly switch at run-time between different I/O devices.

In this paper we present our approach of a device abstraction layer for VR/AR input and output devices extending beyond existing device abstraction layers. Our intention was to include not only common groups of VR/AR interaction devices such as 6-DOF trackers or a space mouse, but also to look at devices that are less frequently considered by device abstraction layers such as temperature sensors or gesture recognition systems, since these devices play a role in upcoming mobile and context-aware VR/AR applications. The result is a general device abstraction layer, that is clearly defined and that can therefore be easily extended by new device types.

The proposed device abstraction layer defines device classes and structures them hierarchically. Devices in this hierarchy inherit the properties and characteristics of their parent classes. One benefit is that applications requiring specific device type functionality may use any device of the corresponding class hierarchy sub-tree. Additionally, characteristics of device subclasses are not hidden, but can be accessed by the application if required.

Another contribution towards more flexible and less device-dependent VR/AR applications are adapters that can be used to transform and filter the data of input and output data.

The paper is structured as follows: Section 2 describes work related to device abstraction layers in the field of VR/AR and in the desktop computer domain. Section 3 gives an overview of the device abstraction layer proposed in this paper and points out our initial objectives. Section 4 deals with input devices and explains the different groups of input devices with their characteristics. Section 5 deals with output devices, respectively. Section 6 explains the concept of adapters. Section 7 concludes the paper and points out possible future directions.

2 Related Work

The requirement to support a wide range of input and output devices is not limited to VR and AR environments. However, within those environments such support is critical, as interaction devices represent an essential part of the user interface and interaction techniques are often employing non-standard interaction devices. In this section, we will focus on related work in the area of VR and AR frameworks, but we will also highlight some fundamental similarities to other environments or libraries providing a flexible support of I/O devices.

OpenTracker [8] provides an object-oriented approach to access input devices, and to fuse, to filter, or to transform their input. While the approach in general is applicable to arbitrary devices, it has a clear focus on tracking devices. Using an XML configuration it allows a flexible combination and processing of tracker data

flows by defining a behavior graph, where nodes generate output upon one or several inputs from sources or other nodes. Combiners and adaptors used in our approach provide a more flexible but also more complex approach to combine or separate data streams. In contrast to our approach, OpenTracker-based applications do not use a unified API, but rather access the appropriate nodes directly as no inheritance of devices is used. Similar to our approach devices may be accessed using a server push, a polling, or a fixed frequency approach. Other similarities are the cross-platform approach and the support of decoupled simulation.

Gadgeteer [3] is a device management system. Similar to our approach, applications may access individual devices through rather generic device types, allowing for replacing the device without any modifications to the actual application. Other similarities include the cross-platform support, the ability to deal with device failure, and the possible distribution of the input devices across several computers. In contrast to our approach where the device class hierarchy uses simple generalized devices as a basis and inherits or composes more complex ones from those, Gadgeteer sub-divides input devices according to their data type (analog, digital, position, etc.). While this approach allows easy replacements of one device of a particular data type by another one of the same type, it does not take advantage of the possible relation between devices by inheritance.

Additionally, both approaches -OpenTracker and Gadgeteer- are limited to pure input devices. Contrary, our approach extends to ouput devices, allowing supporting combined input/output devices. Further, our approach fully integrates streaming devices, not supported by those two approaches.

VRPN [10] implements a network-transparent interface between application programs and physical devices and allows for a dynamic discovery of interaction devices. VRPN is an open-source project and the current version supports a wide variety of input as well as output devices. In contrast to our approach the device hierarchy is rather broad, limiting the exchangeability of devices required by a VR/AR application.

Microsoft provides two device abstraction layer APIs as part of DirectX, which are related to this work. DirectShow [7] is an abstraction layer for streaming media such as video and audio. It allows accessing a wide variety of different devices to be handled equally, no matter if the source is an internet stream, a file or direct input from a device. The second abstraction layer of DirectX for mouse, keyboard and joystick (including force feedback) devices is the DirectInput API. It is basically to allow direct access to these devices by bypassing the Windows messaging mechanisms. The API provides functionalities like iterating through the available devices and acquiring the state of a device. Our approach is quite similar to the DirectInput API, by using the Broker of the Morgan framework; an application can iterate through the devices of a specified type and access them directly. Our approach also goes beyond this abstraction since each device may have an extended interface of its own to provide additional functionality.

3 The Device Concept – An Overview

The common understanding of a device is that it is a piece of hardware attached to a computer. While such a definition is quite intuitive, it is not very precise and even restrictive when it comes to a general description and classification of arbitrary input and output devices. Thus, in this paper, we define devices as follows:

A device is a combination of a hardware component and a software component, sending or receiving data. The software component may contain a driver, a library, or a software development kit.

Including software in this definition provides a useful generalization as for computer devices software actually is an integral component as it would not be possible to access the hardware otherwise. However, in many cases the software even provides the characteristic functionality of a device. A typical example of such a device is a speech recognizer. While capturing of the sound of course requires some piece of hardware (at least a microphone), the more important part is actually the software, which recognizes the speech and translates it into some input, which can be further processed by the application.

The goal of our approach was to provide a general taxonomy covering all (input/output) devices used in VR and AR environments. A realization based on this classification should enable application developers to realize VR and AR applications faster and more efficient, providing a significant higher flexibility regarding the devices actually used. The main requirements for achieving these goals were:

- The approach should allow any new device or device type, not already part of the classification to naturally extend it, without requiring any changes to the original taxonomy (i.e. it should be obvious how and where a new device fits into the existing hierarchy).
- Where devices can be sub-divided into logical sub-units or may only be used in part, this should be reflected by the device hierarchy.
- Users should be able to replace one device by another of similar functionality or even a set of other devices at runtime (i.e. the application developer does not need to be aware of the particular device).
- It should be possible to connect devices to any machine in the system, running an arbitrary operating system.

Our approach has been realized as part of our Morgan VR/AR framework [6] [1] development. Consequently CORBA is used as the general communication mechanism. This results not only in a device hierarchy, but also in an appropriate inheritance of the corresponding interfaces. Developers are free to choose a rather general interface of an abstract base class, allowing any other device inherited from this interface to be used alternatively, or to apply a specific interface, potentially offering some device-specific features, but restricting exchangeability. Typically, all devices actually used within a specific setup are configured using a configuration file. A propriety text based description exists in addition to an XML-based description. Independent of this configuration, new devices may be added or replaced at run-time.

In our approach all devices are derived from one device base class, providing general interfaces required for input as well as for output devices. This includes

setting or querying a device label and querying the operational state of a device, or specific device features (the latter using a universal XML-based query format).

In the following sections the details of our approach regarding the concepts and realization of input and output devices will be described in detail.

4 Input Devices

Input devices represent the most prominent subgroup in the device hierarchy. This is also reflected by the fact that most existing device abstraction layers are actually restricted to this particular group of devices. Input devices cover the whole range of devices used for interaction or for providing sensor information. Although, the definition of an input device may seem obvious, we define it for completeness based on our definition of devices.

An input device is a device, sending data into the system, based upon input from reality.

In general, new input data will be provided by such a device either at specific intervals or in an unpredictable manner. System components interested in input data from the device may either query the current state of the device on a regular basis or at the particular time when the specific data is required, or the device may send the input data upon availability of new data. The first alternative may lead to unnecessary communication resulting in higher latencies due to the round-trips necessary. However, especially components, which do not need regular updates can significantly reduce the update rate by deciding themselves if and when they actually need an update of the current status. This especially applies to applications executed in an environment depending on a low bandwidth connection or temporary disconnections such as smartphones. The second alternative requires the device to keep a list of interested components and to forward the input data whenever new information is available. This minimizes unnecessary data communication and makes it easier for the interested components to receive the data. However, devices using a rather small bandwidth connection might be flooded by too many updates. In our approach we support both alternatives, giving the application developer the freedom to choose the most appropriate one. Hence, system components or applications may subscribe themselves at the input device and will receive state changes regularly using the *publish-subscribe* pattern [3]. Additionally, the current state can be queried without the need of a subscription.

In order to reduce the communication load, subscribers may also choose rather to specify a maximum or a minimum update rate resulting in a minimum or a maximum period between two subsequent updates transmitted. This however, is only applicable to specific devices (see Section 4.2). Using queries rather than the device push mechanism implies another important difference: The queryable state of an input device is the information available at the time of the query, e.g. the current tracked location of an object for a tracking device. However, depending on the individual device, the input data may not result in a state change. One example is a speech (command) recognition device. While the command recognized will be transmitted to

components subscribed; it will not become part of the queryable state and thus cannot be accessed by this mechanism.

We distinguish two major subcategories of input devices: *Discrete Input Devices* e.g. a keyboard (see Section 4.1) and *Continuous Input Devices* e.g. a 6-DOF tracker (see Section 4.2). See **Fig. 1** for an overview of the input device hierarchy.

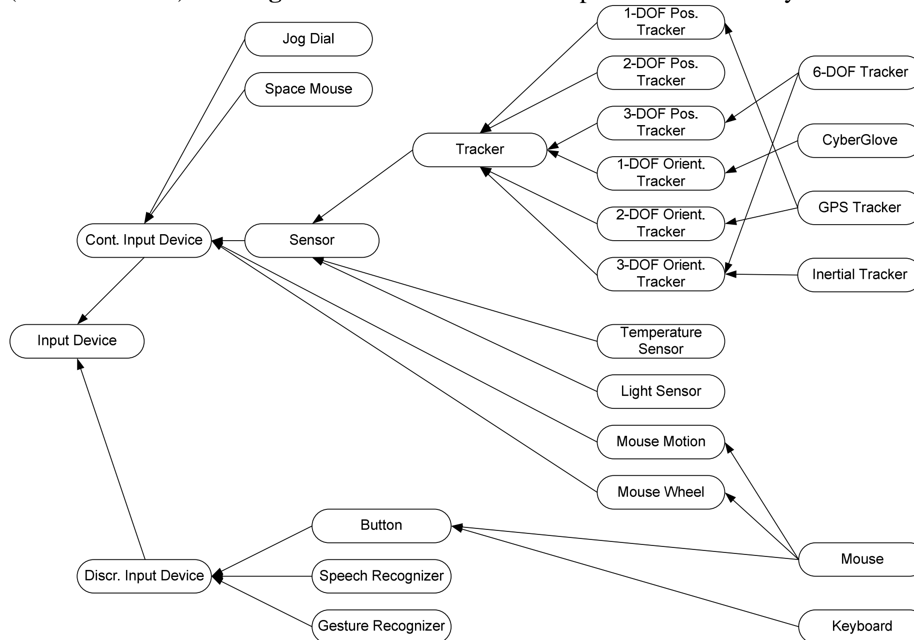

Fig. 1. The hierarchy for input devices. Interfaces for common device classes are inherited by derived device classes.

4.1 Discrete Input Devices

Some input devices generate input data from a finite set of values. Often those values are non-numerical, but rather enumerable, e.g. a light switch may either be *on* or *off* or a button may be *pressed* or *released*. However, the cardinality of the set of values is not restricted to two. Imagine for instance the input selector of an amplifier. It may have six different states {tape, cd, aux, dvd, microphone, tuner}. We define such devices as *Discrete Input Devices*:

> *A discrete input device is an input device, providing decisive input data values from a finite set of discrete values.*

The following devices are currently supported as part of this hierarchy, i.e. the classes representing these devices are derived from the abstract class *DiscreteInputDevice* (compare **Fig. 1**): *Button, GestureRecognizer and SpeechRecognizer*. Button may be e.g. a mouse button or a keyboard button with the

states *pressed* and *released*. The gesture and speech recognizer are able to recognize a finite set of hand and finger gestures or spoken commands, respectively. The latter two devices actually do not have a state, which represents the current gesture or word, but they issue an event after recognizing it.

4.2 Continuous Input Devices

In contrast to *Discrete Input Devices*, *Continuous Input Devices* are characterized by continuous input values continuously issued during the usage of the device. The input data may be any value within a certain range of numerical values. The volume knob of an amplifier for instance may have a state within the range [0…100]. Hence we define *Continuous Input Device* as follows:

> *A continuous input device is an input device, continuously providing input data values within a certain (continuous) range.*

The data type of continuous input devices may be any continuous data type. Examples include but are not limited to integer and floating point numbers, 3D vectors, n x n matrices, etc. The state of a temperature sensor may be a value within the range [-20°C…40°C] or the input from a slider may result in integer values within the range [0…3]. The data values provided by continuous input devices have a limited decisiveness, as intermediate values are typically not decisive and may be omitted (e.g. to reduce network traffic). Continuous input devices allow transformations and filtering on their data, which is not possible for discrete input devices.

As mentioned above, in our approach subscribers may choose a maximum update frequency. However, restricting the update frequency usually is only useful for Continuous Input Devices as skipping of a state change in a Discrete Input Device will quite frequently render the transmitted data useless. Beside the maximum update frequency, continuous input devices provide the possibility to specify a desired update frequency. As many of these devices can be sampled or can be configured to deliver events at an application dependent frequency (at least within a certain range), this allows to access a specific device in an optimized manner. An application rendering a 3D scene at a fixed frame rate for example, may choose this frequency for a specific input device to achieve a steady execution. However, the desired frequency of course must not exceed the maximum update frequency (if specified).

The most important sub-hierarchy of continuous input devices are sensor devices:

> *A sensor device is a continuous input devices estimating or measuring a real property and allowing for calibration (in order to provide the desired input)*

The measurement of the property will influence the current state of a Sensor. A typical example for a Sensor device (which is not a Tracker – see below) is a thermometer. Sensors have to be calibrated in order to work properly. However, this does not necessarily have be done by the developer or user, but could have been done by the manufacturer of the device. Nevertheless the software (interface) should allow for a re-calibration or re-adjustment. This is not true for the other continuous input devices, such as the mouse motion device. The mouse motion device provides

(depending on the used driver) relative position information about the mouse's movement, but it does not have an absolute position, since it can be picked up and placed somewhere else without changing its state.

Trackers are a subclass of Sensor Devices, which estimate the position or orientation of a real object in 1-3 degrees of freedom (DOF) – a 6 DOF tracker is in our hierarchy a combination of a 3 DOF position and a 3 DOF orientation tracker. They are essential for all kind of VR and AR systems as they require to track the user's location allowing to render the appropriate viewpoint dependent view as well as realizing 3D input such as wands, 3D pointers, or tangible interfaces. All trackers share the same common interface, allowing for setting a transformation matrix containing the individual calibration data. Thus, we define tracking devices (trackers) as follows:

> *A tracking device is a sensor device whose measured property is a position or orientation (1-3 DOF each)*

As already mentioned, tracking devices, which combine position and orientation tracking derive from the appropriate classes, e.g. a GPS tracker derives from 1 DOF position tracker (altitude) and 2 DOF orientation tracker (a direction vector representing longitude and latitude).

Since all devices also provide their data through their derived interfaces, one of the major advantages of this hierarchy is that it allows direct subscription to the interfaces of the actually desired or required device. For instance, a subscriber may only be interested in 3 DOF position data of an object; it will only receive this information and does not have to care whether the object is maybe tracked by a 6 DOF tracker.

5 Output Devices

Output devices represent the second large sub-hierarchy of devices. They are able to represent or to emit information. Again, we define it for completeness based on our definition of devices as follows:

> *An output device is a device, receiving data from the system, affecting its output to reality.*

It either puts the device into a new state or it invokes the device to emit this information, e.g. the state of a relay is set or a speech synthesizer emits the words. Similar to input devices, the state of the output devices can be defined as the information currently represented by the device at the current time. Accordingly, the state is not necessarily identical to the latest output data sent to the device. Similar to input devices, the state of output devices is queryable.

In general, the sub-hierarchy of the output device resembles that of input devices. This also extends to the individual data types and data values. Actually, for many input devices a corresponding output device exists. While for instance a 3-DOF position tracker provides location information, a 3-DOF positioner (output device) allows positioning an object within a specified 3D coordinate system.

According to the sub-hierarchy for input devices, we also distinguish between discrete output devices and continuous output devices. An example for a discrete

output device is a switch, which may represent two or more states, e.g. {on, off} or {low, medium, high}. Examples for continuous output devices are relays, positioners and rotators. A concrete example of a positioner and a rotator is the CyberForce, a force feedback device, which is able to control the position of the hand and the angle of each joint. An industrial robot is another example.

6 Adapters

Adapters [3] represent a powerful software design pattern [4], which can be used in this hierarchy as a software component for converting between different interfaces. Adapters allow for using a device in a different way and/or with a different interface than the one originally provided. Thus, an adapter makes a device behaving like a different one. In order to achieve this, an adapter inherits from the destination interface and from the subscriber of the source interface, adapting the input interface to the desired output interface.

For example, an application uses a 3-DOF position subscription to a 6 DOF tracking device to update its camera position, e.g. using an ARToolkit [5] based 6-DOF tracking. In order to be able to use a GPS tracker instead, an adapter can be used to convert the information of the GPS tracker (2D orientation and 1D position) into a 3 DOF position tracking device and publish this data through the interface of a 6 DOF tracker (see **Fig. 2**). This enables the application to subscribe to the GPS tracker adapter. Thus, the two devices may be exchanged without any modification to the application.

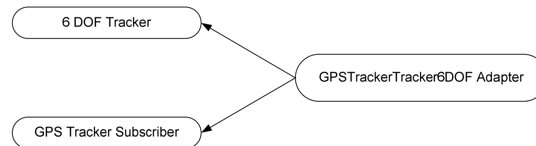

Fig. 2. Inheritance diagram for the GPS tracker to 6 DOF tracker adapter.

Since adapters are not limited to one input device, they can also be used to combine the data of several devices and provide them through one interface. The GPS tracker adapter could e.g. also subscribe to a 3 DOF orientation tracker, e.g. an Intersense InertiaCube, in order to provide the full 6 DOF information of the user.

Filters represent another major application area of adapters. We use for instance adapters to provide interpolation and extrapolation filters to input data. Inheriting the original interface of the input device, this allows adding filter chains between a device and the actual application, without the need to change the application.

7 Conclusions and Future Work

In this paper we presented DEVAL, our approach to a device abstraction layer, developed to provide a universal device hierarchy for VR and AR applications. Our

goal was to provide an approach which would not only classify existing devices, but also easily extend to any new devices in the future. Using inheritance not only for the device classes, but also for the networked interfaces allows users to use devices which did not even exist at the time the application was developed. Further, our approach provides the application developer with the full flexibility on how to receive the input data from a device. We introduced the major subcategories of input devices within our device abstraction layer: discrete input devices, continuous input devices, sensor devices, and tracking devices. We further showed how our approach naturally extends to output devices. We finally showed how adapters can be used to convert and combine input data from several devices, or to apply filters.

In our future work we intend providing support for streaming input and output devices as well as for alternative network distribution channels beside CORBA.

Acknowledgments. Parts of the work described in this paper were performed within the IPerG project and the IPCity project. IPerG and IPCity are partially funded by the European Commission in FP6 (FP6-2003-IST-3-004457 and FP6-2004-IST-4-27571).

References

1. Broll, W., Lindt, I., Ohlenburg, J., Herbst, I., Wittkämper, M., Novotny, T.: An Infrastructure for Realizing Custom-Tailored Augmented Reality User Interfaces. In: IEEE Transactions on Visualization and Computer Graphics, Vol. 11, No. 6, IEEE Educational Activities Department, Piscataway, NJ, USA (2005) 722-733.
2. Burdea, G., Coiffet, P.: Virtual reality technology. Wiley-Interscience, 2004.
3. Gadgeteer, 2005: http://www.vrjuggler.org/gadgeteer, Virtual Reality Applications Center, Iowa State University, Ames, IA.
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
5. Kato, H., Billinghurst, M., Blanding, B., May, R.: ARToolkit. Technical Report. Hiroshima City University, 1999.
6. Ohlenburg, J., Herbst, I., Lindt, I., Fröhlich, T., Broll, W.: The MORGAN Framework: Enabling Dynamic Multi-User AR and VR Projects. In: Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST 2004), pp. 166-169, ACM Press, New York, 2004.
7. Pesce, Mark D.: Programming Microsoft DirectShow for Digital Video and Television, Microsoft Press, 2003.
8. Reitmayr, G., Schmalstieg, D.: An open software architecture for virtual reality interaction. In: Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST 2001), pp. 47-54, ACM Press, New York, 2001.
9. Rolland, J.P., Baillot, Y., Goon, A.A.: A Survey of Tracking Technology for Virtual Environments. In: Fundamentals of Wearable Computers and Augmented Reality. Chapter 3, pp. 67-112, 2001.
10. Taylor, R., Hudson, T, Seeger, A., Weber, H., Juliano, J, Helser, A.: VRPN: A Device-Independent, Network-Transparent VR Peripheral System. In: Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST 2001), pp. 55-61, ACM Press, New York, 2001.