On Device Driver Architectures for Virtual Reality Toolkits

Abstract

One major goal for the development of virtual reality (VR) toolkits is to provide interfaces for novel input or output hardware to support multimodal interaction. The research community has produced several implementations that feature a large variety of device interfaces and abstractions. As a lesson learned from existing approaches, we sum up the requirements for the design of a driver layer that is the basis for a multimodal input and output system in this paper: We derive a general model for driver architectures based on these requirements. This model can be used for reasoning about different implementations of available architectures. As the flow of data through the system is of interest, we take a closer look at common patterns of data processing. Finally, we discuss a number of openly accessible driver architectures currently used for VR development.

1 Introduction

The design and implementation of novel input and output devices are core research areas in VR. Literature about this topic can be roughly divided into two categories. The first describes the implementation details of novel devices, their impact on users, and application sketches. The second category aims at providing abstractions of specific devices that can be used for software system processing, but mostly with an emphasis on generic solutions that implement various interaction paradigms. However, articles that allow a comparison between different implementations of driver architectures are rare. To most, driver integration seems to be solved as soon as data can be accessed by the application software. While this is an almost trivial task, building a stable driver architecture that seamlessly integrates into existing environments and respects VR-specific requirements is not trivial at all. This leads to the question of what the requirements of a VR driver architecture are. Existing de facto standards, for example, the Human Interface Device (HID) specification or the Virtual Reality Peripheral Network (VRPN) library, may lead to the conclusion that there is no need for a specific driver implementation other than one based on these solutions. However, since HID only specifies a general transport protocol and VRPN suffers from latency and congruency problems, several custom implementations exist as a solution to specific requirements. These solutions share common ideas in their implementation, and deal with recurring problems. This paper gathers aspects of the implementation of driver integration architectures as a sum of existing approaches.
In Section 2, we name the requirements that are special to VR applications with regard to device handling. After that, we list common ideas of data processing at the border between the driver layer and the VR application in Section 3, and thereby build a conceptual driver architecture to provide a common nomenclature. In Section 4, we focus on the aspect of data processing. After that, we reflect several existing VR driver architectures and selected aspects of their implementation in Section 5. We conclude with a summary in Section 6.

2 Requirements

This section will list a general set of nonfunctional requirements of VR driver architectures that were gathered by inspecting several existing approaches. They can be categorized into two main groups, data processing and application requirements.

2.1 Requirements of Data Processing

The data processing and device management stage is the common point of contact for device designers and users of VR devices. Three main requirements can be identified for this group.

• Interface Consistency. Any software that integrates different devices has to provide more or less standardized interfaces for the application to access them. A typical example of a VR device is a tracker that records the transformation of a tracking target in real time. Different approaches comprise mechanical, electromagnetic, optical, or acoustical tracking. Each technology and product can differ in update frequency, precision, latency, data format, and content. However, an application programmer should be able to use different devices without adjustments of the code. Therefore, clear and unified interfaces to the device are needed to ensure a consistent configuration across all supported devices. While this requirement is commonly met, it is often implemented with different qualities. Creating a good implementation can be difficult since, for example, novel devices might not fit into an existing scheme and a consistency cannot be maintained.

• Multimodal Data Types. Multimodality is a key aspect to VR applications. For a driver architecture this means that any possible data type should be recordable, and accessible at the data interface. Multimodal data typically differs in type, temporal resolution, and size. As an example, an application could be designed to combine speech input with (6 degrees of freedom) input to select the target for a spoken command. Different devices are used to detect the object selection with different types of data to work on for this selection interface.

• History Support. Temporal correlations between different modalities have to be respected for advanced and natural interaction. It is important to observe the change of a property over time, and not only the most current date. For this purpose, it is beneficial for a driver architecture to support a time series recording of multimodal data. Considering the example of speech input and pointing device, the application may need some time to perform speech recognition in order to determine a spoken command. The associated pose should correspond to the pointing device’s state at the time the command was spoken. If the pose is read after the recognition, it may have changed already and can produce inaccurate selection results.

2.2 Application Requirements

When devices are set up and data are sent, the VR application needs interfaces to access the incoming data. For these special interfaces we identified the following requirements.

• Concurrency Support. As devices often run concurrently with the application, the driver layer should offer a means of synchronizing data access. This includes thread level parallel access, but might as well be implemented as distributed devices.

• Congruency Control. VR systems often use many different components that cooperatively solve interaction tasks, and thus it is important to know
Figure 1. Core elements of the abstract driver architecture and their relationships. Stars define the multiplicity and edge labels define the relationships between the terms.

if each component in this setup is working on the same data. The driver layer should offer a controlling mechanism to ensure these data congruency. Although this requirement is often not regarded as important, it is mandatory for flawless use of different modalities. A simple time-stamping for individual data items should be provided at least.

• **Transformation Support.** The driver architecture should offer standardized means to perform transformations on the data. A common example is the pose, that is, position and orientation, received from the tracking hardware, which has to be transformed into an application reference frame.

• **Functional Decomposition.** For a successful device abstraction, the interface should provide methods to query the properties and functions of a device. Individual properties of each device can then be accessed and combined in order to create multimodal data sources, for example. Depending on the implementation, these properties can reflect high-level functionality, such as the support of haptic feedback, or basic data types such as axes, scalars, and digital values.

• **Low Latency Data Access.** Device data are a central component of interaction processing and are accessed frequently throughout the application run. Therefore, data access should not result in additional system latency, other than fetching of the data from the device.

• **Controllable Latency.** Although latency cannot be fully avoided, a driver architecture should be able to identify the amount of latency that was added as a consequence of the data processing. Ideally, the application can query information about the age of the data and system overhead cost. This additional information can be used in the process of latency compensation.

3 **Driver Architecture**

This section outlines the driver architecture abstracted from existing approaches, which will be explored more fully in Section 5. The purpose of this section is to distill common characteristics and interesting features. It serves as a conceptual design for reasoning about driver architectures in general. Figure 1 provides an overview of the elements of the architecture that are subsequently explained.

The **driver repository** manages existing drivers and provides access to them. A **driver** is the central element that is connected to a physical device over a data bus. It is linked to a **decoder**, which again has access to **sensor data** memory to which it writes. The sensor data are accessed by client code through a **transcoder**, which defines a structured access point to the accumulated raw data of the driver. Afterward, the data are used by the **transform** stage, where it is processed to meet the application’s needs. The transformation stage can also access the **aspects** of a driver in order to gain information about it, or to write data back to the driver. Finally, the data from the transformation stage can be used by the **simulation loop** of the actual VR application.
All existing VR driver architectures define these stages more or less explicitly. Usually, the discrimination between decoding and transcoding is not visible, and the transformation step is often performed at the application level and not by the driver architecture. Even mixtures can be observed, where the steps of decoding, transcoding, and transforming are intertwined.

The driver architecture can be roughly divided into two parts, the driver side and the client side. To transport data between them, the sensor data and aspect components form an exchange interface that is accessed at varying frequencies from both sides. The boundary between the driver architecture and the VR toolkit is not fixed. The implicit transcoding and transforming on the driver level is a popular technique used by many VR systems to make devices easily exchangeable, but it can also be performed on application level. The integration of the transformation stage into the driver architecture increases the usability, but also has drawbacks. It may be necessary to undo the transformation at the application level, for example, to retrieve the original values.

VR applications often run in a distributed setup, so that device data have to be distributed among different systems. This step of remote data sending can be handled at different points in the process, for example, by distributing the data after the decoding, transcoding, or transformation stage. This implementation detail is thus not explicitly modeled in the presented architecture.

### 3.1 Driver and Repository

Drivers are the interface between hardware and driver architecture. They provide means to handle the input and output of the associated device. This task is very device-specific, so additional abstractions have to be made to reach interface consistency.

Inside the repository, many instances of drivers of the same or different type are stored. Their existence and availability can be checked by client code. In some implementations, the repository is only an addressing scheme used to find individual drivers.

### 3.2 Decoder

The decoder collects the device data from the driver’s bus and converts data into meaningful chunks of information. This step is very device-specific and usually requires operating system support. The decoder may be integrated into the driver, but some approaches decouple it in order to allow an implementation to dynamically exchange the decoding. This decoupling is needed, for example, to implement drivers based on a meta-standard such as HID, which uses the same data transport mechanism for a family of different devices.

Time-stamping is typically handled during decoding. It is a central aspect and enables a method of relating samples from different sources. Typical use cases for time stamps include prediction, motion analysis, and reaction-time recording. We assume that, even in a distributed case, a global clock exists that is used for stamping. The resolution of the time stamps for a VR setup should have at least millisecond resolution.

Time stamps can be applied at different places during decoding. Since the time stamps are taken at different steps during the driver data processing, they can vary significantly depending on factors such as operating system scheduling policy. Time-sensitive algorithms, for example, for latency compensation, may suffer from a bad choice.

- **Device-Sourced.** The time stamp is delivered by the device and indicates the moment of incidence of the recorded event. Devices typically use internal timing hardware to provide time stamps. This can make it difficult to compare them to a global clock.
- **Detection-Sourced.** The driver is capturing the moment in time when the data are first detected, that is, the time the data arrive over a transport connection.
- **Delivery-Sourced.** This time stamp indicates the moment when data are delivered to the application by putting data in a reading queue.

Apart from the time of creation, the data time stamp can also be seen in relation to other readings from the device.

- **Per-Value-Snapshot.** The time stamp of the data is related to the moment when data were decoded.
from a block of new data. In this scenario, all values of the current state of the device have a slightly different time stamp. For example, when a device has more than one button, and the buttons are decoded one after the other, a new time stamp is assigned to each individual button value upon decoding.

- **Per-Reading-Snapshot.** A time stamp indicates a unique point in time that is valid for the complete state of the device. Samples from the same device and decoding step can thus be selected by exactly matching the time stamp.

### 3.3 Sensor Data

After decoding the device stream, information is stored in the sensor data memory. This can either be located in driver or client memory. The sensor data memory is an important element, as it is generally accessed with high frequency, either by client code polling for new data or by drivers that are writing new data. Implementations have to critically inspect the design of the structure to avoid performance penalties or processing flaws, which will be discussed further in Section 4. Additionally, the choice of implementation also determines the possibility of storing and processing multimodal data. Usually, implementations use a closed type system, allowing only data types that are commonly needed in VR applications, such as positions and triggers. A few approaches allow arbitrarily typed storage of sensor data. Other architectures solve the problem by encoding device values as arrays of base types, for example, floats or Boolean values. However, this requires a reconstruction of abstract data types on the application level with specific knowledge of the setup.

### 3.4 Transcodes

The transcoding stage defines the system border of the driver architecture to the VR system, as it provides structured access to the data collected during the decoding phase by the driver code. Typical interfaces for querying a current value from the device within a VR system are as follows.

- **Symbolic Access with Anonymous Return Type.** The driver is queried by passing a token or symbol and an anonymous memory blob to the driver. This instructs the driver code to write the information queried by the token into the given memory. Many straightforward designed VR driver toolkits adapt this interface and use compile-time defined tokens for data access, and then use setup-specific knowledge to interpret the result.

- **Explicit Functional Access with Known Return Type.** The driver code provides methods or functions that give access to individual data elements that the driver has collected. A dedicated access function with known return type exists for each data element. However, in a language where the set of functions is compile-time defined, a lot of different functions may be needed to provide all required access methods.

- **Symbolic Functional Access with Known Return Type.** The driver offers an interface where a getter method for a named value can be queried using a symbol. The driver code implements a dictionary of methods that allows typed access to driver data in a unified way.

The choice of an access interface usually determines integration aspects of the driver architecture into existing VR systems. The symbolic access promises a flexible extension scheme with regard to driver implementation because a driver can inspect the passed token and interpret it within its capabilities. Usually, the set of symbols is compile-time limited, so client code can access a closed set of properties from a driver. Obviously, this interface is not very extensible. For example, it is hard to encode behavior that is not covered by the present set of symbols. The explicit functional access is more flexible, as it allows for consideration of individual capabilities directly at the interface of the driver. The symbolic functional access is the most flexible in terms of usage and extensibility. However, client code usually needs to add further abstractions for a homogeneous processing using that interface. Such code requires advanced infrastructure to be built, as parts of the architecture have to support reflection interfaces and management entities that handle registration and access of the
dictionaries at runtime. Thus, very few approaches support this type of access.

### 3.5 Transformation

The transformation stage comprises methods to get data from the transcoding stage, combining data with additional sources of information, such as other devices or system objects, and converting data to a desired format. The following transformation structures can be identified.

- **Single Stage Transformations.** A common approach is the use of interfaces that allow for the modification of a single data once it is ready to be processed. This is enough to convert a device- or setup-specific data, for example, by transforming a pose to a common reference frame. Sometimes, these transformations can be specified using a configuration file or during runtime. Attempts to enhance this type of transformation have to be implemented on the application level. This can lead to error-prone and nonreusable code. It is not very applicable for multimodal processing and is mainly suited for simple spatial transformations.

- **Pipes and Filters Transformation.** An extension of the single stage transformation is the pipes and filters approach, where a number of transformation steps can be chained together to produce more complex transformations. Pipes and filters architectures are well discussed in the literature (e.g., Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996). In the scope of VR, it must be noted that a pipes and filters approach—though more flexible than single stage transforms—typically only inspect changes of data from a single data source to a single sink. The approach shows fundamental problems of recursive processing and clean M:N data distribution.

- **Data Flow Networks.** A data flow network provides a customizable processing framework where driver data are inserted into a network of data sources, processing nodes, and data sinks. Data flow networks offer an accepted approach to device data transformation in the scope of VR, and are well documented in the literature (e.g., Haan & Post, 2008; Figueroa, Bischof, Boulanger, Hoover, & Taylor, 2008). They are usually implemented on the application level rather than being part of the driver architecture. Some approaches even use data flow networks for application modeling, raising the need for support of cyclic graphs and multimodal data processing and more complex state control (e.g., Allard et al., 2004; Reitmayr & Schmalstieg, 2001).

### 3.6 Application

The results of the transformation step are finally needed in the application. On a coarse level, we distinguish push and pull approaches. Push approaches result in a callback as soon as new data are transformed, typically with local context of the data delivered. In pull approaches, the application initiates a pull of data, fetching new data that are ready to be exchanged. Device abstraction is often created by the definition of registers by the application. Registers are frequently implemented as a publish-subscribe mechanism (Buschmann et al., 1996). The register approach is a powerful metaphor, as it defines a boundary between the device layer and the application logic and provides clear constraints on the value of the single registers that are needed for application processing.

### 3.7 Aspects

Driver aspects are an important means to classify the quality and types of services that a device can provide. They also serve as a basis for the functional handling of the devices within a VR system. In the presented conceptual context, aspects model a driver’s features, for example, force feedback, the presence of workspace bounds, or a local coordinate frame. The functional decomposition of devices allows for the use of individual properties of the device in new contexts. Not only can event sources be modeled as aspects, but also static information, for example, the serial number of the device. Three ways of implementing aspects can be identified.
Assenmacher et al.

Figure 2. A static class layout for the type constrained implementation of driver aspects. Functional decomposition of the driver capabilities is achieved by inheritance.

Figure 3. A typical static class layout for the dynamically associated interface of driver aspects. In this example, the user code can use the specialized interface for extended functionality, for example the definition of device dependent force effects.

- **Inquisitive Interface.** The inquisitive interface is characterized by an implementation that provides a common interface for querying all types of supported capabilities. Typically a single routine is used, passing a query token and a memory blob to store its results to.

- **Type Constrained Interface.** Another way to implement aspects is by viewing them as an integral part of the device using the aspect as a base type. In this sense, the functional aspects become part of the device implementation in an is-a-relation, as shown in Figure 2. A problem of this approach is that there are more driver classes than for the other approaches. Possibly every existing combination of properties has to be inherited. For example, a joystick that supports force feedback differs typologically from a joystick without force feedback.

- **Dynamic Aspect Association.** Aspects can be handled as dynamic attachments to a driver instance. A driver class can hold any number of functional interfaces, as shown in Figure 3. This allows the configuration of a driver to be easily changed, even during runtime. A drawback of this approach is that aspect instances always have to be bound to a specific instance of a particular driver and cannot be used polymorphically where drivers can be used.

4 Data Processing

When developing a driver architecture, one important issue is the efficient processing of data from the device to the application. This proceeds in several stages, which are discussed in the following. The implementation of each stage can be very different and depends on the requirement of the specific toolkit.

4.1 Data Sampling

The current state of the device sensors has to be determined at a dedicated point during the application run. Two operation modes for state sampling are common in VR systems.

- **Poll Sampling.** With poll sampling, or synchronous update, the client queries the device driver for the current state of the device. The driver then queries the device and provides the requested information.

- **Stream Sampling.** In this variant, data from a device are continuously streamed to the driver. The assumption is that it is cheaper to stream and
then sample the current state than to invest in the overhead of polling. This variant is also called asynchronous update.

Most implementations choose stream sampling to retrieve new data. Stream sampling has the advantage of capturing the device data at the update rate of the device or transport layer. This introduces the cost of dealing with explicit synchronization and decisions about the buffering between the driver layer and the transcoding interface. Not all devices are capable of stream sampling, for example, when they are event-triggered. Ideally, a driver architecture should support both notions of update processing, depending on application and device requirements.

### 4.2 Data Exchange

Once the data are available at the driver interface, data have to be exchanged with the application. For data transfer from driver to client, two common strategies can be identified.

- **Producer-Consumer Exchange.** In the producer-consumer exchange scheme, the driver is collecting data coming from the device, until a consumer fetches the accumulated data. This approach usually requires a buffering scheme with a locking mechanism. It implicitly assumes a 1:1 relationship between the producer and the consumer. With every readout, the data are moved to the application memory and are no longer available at the driver.

- **Window Exchange.** The window exchange collects a number of samples that can be read out in time- or sample-windows starting from a selected time stamp. The approach requires synchronization during the write and the read phases.

The producer-consumer exchange is the simpler type of data exchange and easier to implement and manage. The window exchange scheme is more suited for concurrent setups with multiple readers, since otherwise data distribution needs to be handled on the application level.

### 4.3 Granularity

Another interesting issue is the granularity with which the device data can be accessed by the client code.

- **Message-Level Access.** This access strategy allows data describing the whole state of the device to be read, that is, as a set of data values that are valid for a single time stamp. For example, the position and orientation of a sensor body is sampled at a distinct point in time and they can only be accessed as a pair.

- **Per-Attribute Access.** This strategy allows a single property from a data record to be fetched, for example, reading the position of a sensor without inspecting its orientation. The shortcoming of this method is that one may lose the temporal connection between different attributes of the same record, for example, when the orientation data are overwritten while reading the position data.

- **Time Stamp Based Access.** In order to allow time-based queries and history access, the driver can store time stamps for each data sample. This is helpful for temporal analysis, such as prediction of a tracked sensor, or for correlation of different devices for multimodal interaction.

### 4.4 Exchange Data Structures

When the strategy for data exchange is set, one still needs suitable data structures for the exchange process. Among the existing approaches, three main categories can be distinguished.

- **Single-State Properties.** Drivers offer just a single snapshot of their state properties. A new snapshot will overwrite the last one. The nature of this structure is that it is present at the time of access, without relation to other data, for example, from previous points in time or from other devices.

- **Queues.** In order to avoid missing incoming data, some architectures push new data into a queue. This queue is constantly inspected and emptied by the application. This results in a producer-consumer exchange scheme.
scheme where more than one item can be produced and consumed per iteration.

- **Histories.** Histories, in contrast to queues, provide access to data as a time series. Each entry in a history is time-stamped, and histories are traversed instead of emptied. This, unlike queues, keeps the local past persistent between evaluations by client code.

When only the latest sample is of interest, the single-state property approach can be used. However, as the requirements state, multimodal interactions require temporal information. Queues are a suitable approach to this if only a 1:1 relation between driver and consumers is required. In a 1:N communication scheme, one consumer usually handles the distribution to the remaining consumers, shifting the distribution problem from the driver to the application layer.

The single-state property is often found in existing libraries, as it is easy to implement. Its use in combination with stream sampling is problematic, as the congruency requirement can be violated if new data arrive in between the access of different clients. When combined with per-attribute access interfaces—as is often done—this is even more error prone, since different attributes can change between attempts to access them. Histories conserve temporal relations of individual samples and provide a solid base for multimodal interaction. Additionally, a 1:N writer to reader relation can be implemented.

### 5 Example Driver Architecture Implementations

Currently, there is not much literature dealing explicitly with the modeling of device driver architectures for VR systems. Most related work focuses on device abstraction as a precondition for generic interaction support. In the following, we will present some examples for existing implementations of driver architectures, which are usually coupled to a VR toolkit. Whenever possible, we consulted source code as the basis for our summary on the toolkit. Since the capabilities of the individual implementations may change frequently, we supply the version number of the inspected source code. In case we do not supply a version number, all statements are based on the cited publication.

A commonly cited work in this area is the MR Toolkit (Shaw, Green, Liang, & Sun, 1993). It defines software engineering requirements for a driver architecture, for example, portability, support of a variable number of devices, flexible configurations, and serves as a basis for application development. A main idea of the MR Toolkit is to provide an explicit transcoding stage between client and server code. Transformations are supported through filters and can be attached right after the transcoding to modify the driver data. The calculation is performed with device speed and within the scope of the driver process. Devices usually run in their own process scope, communicating through shared memory or network with other parts of the VR system. The application stage in the MR Toolkit is implemented as an early register approach. The handling of time-stamping is not discussed in the publication.

One of the most popular and used libraries is VRPN, currently at v7.26 (Taylor et al., 2001), which implements a distributed approach to device handling. Application clients connect to a device server in order to receive new data or to push instructions to the driver server. The authors treat devices as compositions of functional attributes and model them using type constrained interfaces implemented by multiple inheritance. Inherited aspects can be used to query individual channels of incoming data that can be combined to create virtual devices on the application level. It defines a publish-subscribe mechanism to create callbacks when the client code gives computation time to VRPN in the simulation loop. VRPN uses TCP/IP for the data transmission by default, so one can regard the network as a buffer between reader and writer. The events still in the network cannot be traversed during the callbacks. The toolkit supports all types of time stamps, typically using delivery-sourced time stamps with per-value-snapshots. The authors of VRPN do not make strong assumptions about how it should be integrated into client software. As a consequence, there is no real interface for driver...
management or dynamic configuration. This is not the scope of VRPN, as it aims at being integrated into different toolkits that have their own strategies for this. VRPN can integrate custom protocols over a general marshalling interface.

The OpenTracker framework, currently at v2.0 (Reitmayr & Schmalstieg, 2001), shows a tight integration of the device driver architecture with a data flow oriented transformation graph structure. In this approach, multimodal device data are modeled by a dictionary approach, where events that carry information about the device data are passed through the data flow network. This framework demonstrates that it is useful to integrate the interfaces of a transformation stage with the driver level architecture, as it allows a flexible coupling of device data and interaction algorithms. Data source event objects are pushed to the network upon update of the source nodes, implementing a single-state property. Time-stamping is performed on driver level, and existing driver code seems to prefer delivery-sourced time-stamping with per-reading-snapshot. The data flow network in OpenTracker is traversed by concurrent threads that evaluate different parts of the network. The nodes' output and update frequencies are not modeled externally, and conflicts are resolved using node internal local locking strategies.

VRJuggler, currently at v2.2.1 (Bierbaum et al., 2001), provides its own device abstraction called gadgeteer, which can incorporate devices from other driver architectures, for example, VRPN. It is an example for a toolkit-independent device driver architecture that uses buffering schemes to implement sampling strategies. Internally, drivers work as a collection of independent threads that continuously collect buffers and dispatch them to the application. All input data can be time-stamped by driver code. It has the interesting property of being able to time-stamp using intervals, for example, by giving delivery-sourced stamps as a lower bound and detection-sourced time-stamps as an upper bound. Existing code methods seem to use delivery-sourced time stamps and per-value-snapshots. The functional decomposition of devices in VRJuggler is comparable to VRPN’s approach.

IDEAL (Fröhlich & Roth, 2001) is the input and output system of Virtual Design II. It represents a network architecture where drivers run in dedicated processes. New data are delivered to a number of registered applications. Its major contribution is a strategy to reduce network traffic and transmission of device data just in time for a rendering. The reduction of latency is reached by omitting intermediate data, and only necessary data is sent to the application, at a time “slightly before it is needed.” The implementation of the time-stamping is not clear from the publication. In IDEAL, a device is described by a number of input and output dimensions, presumably a variable array of basic types. A filtering step allows for the transformation of the data of the devices to a unified metrical coordinate system. A tightly coupled callback structure is used to push incoming data to the application.

The FLUID toolkit (Ilmonen & Kontkanen, 2002) deals mainly with the implementation of data history processing for multimodal input. This toolkit emphasizes the use of histories for multimodal access and depicts various access patterns for histories. Their history model is implemented as front- and back-buffer queue access. Data are swapped in and out at dedicated points in time, effectively forming a producer-consumer data exchange. The handling of time-stamping is unclear from the publication.

DIVERSE (Kelso, Sattersfield, Arsenault, Ketchan, & Kriz, 2003) supports a variety of devices using a base library called dtk, currently at v2.4.18. It has the interesting capability of addressing device drivers over named shared memory segments running in different processes, similar to the MR Toolkit. A filter architecture is used for transformation of the data after decoding. It supports a queuing scheme that allows reading of older values from a device. The queue is implemented by a circular array that is accessed byte-wise. Transcoding is implemented by intermediate data types that introduce more complex memory layouts and ensure block-wise access to the raw storage. Individual samples are time-stamped either as detection-sourced or, when unstamped, automatically as delivery-sourced. Existing driver code seems to prefer delivery-sourced time-stamping with per-value-snapshots.
Syzygy, currently at v1.3.0 (Schaeffer & Goudeseune, 2003), is a VR toolkit using remote devices for application replication or master-slave rendering for a clustered environment. It implements single-state properties for new data, supporting a fixed set of data types, which are triggers (1 DOF), axes (3 DOF) and transformations (6 DOF). It offers an interface to couple several devices in order to construct virtual devices. The authors mention a filtering interface that allows for transformation of the device data, realizing a pipes-and-filter architecture by chaining filter elements. The filtering focuses on normalization steps, for example, to scale the ranges of different devices to a common reference frame. The data sources are abstracted in device nodes. Device sinks represent interfaces to the Syzygy application. This processing can be interpreted as a distinction between decoding and transcoding, where the transcoding is limited to its type system, that is, the different degrees of freedom. Syzygy does not seem to support time-stamping of new data on the driver level.

Chai3D, currently at v1.6.2 (Conti, Morris, Barbagli, & Sewell, 2003), is a development library for software that needs haptic support with visual feedback. It provides a collection of drivers, including simulated, virtual, or proxy devices, for example, GUI panels. The implementation of the driver code for various haptic devices is straightforward and uses a dictionary approach to retrieve different device states in various types. Chai3D is an example of a toolkit that provides inquisitive interface access to the driver data. A recent version (v2.0.0) uses functional decomposition by inheritance. The inquisitive interface is still present, but seems to be abandoned in favor of a type constrained interface. Chai3D does not support time stamping of individual samples on driver level. All input data is modeled as single-state property.

VRui, currently at v1.0.62 (Kreylos et al., 2006), offers a distributed device abstraction layer similar to the MR Toolkit. It provides a high-level transformation stage in a separate layer where device data are connected to produce events. This architecture is an example for explicit integration of the transformation stage into the device abstraction layer. Internally, it uses a global register model that routes device input into an array and a publish-subscribe mechanism is used to notify the application. Between the slot and the output callback list, transformation support can be installed using an arbitrary code piece to convert data to a desired reference frame. As these code pieces can reference other slots, the system introduces an update and dependency check algorithm to avoid endless cycles or missed updates. VRui does not support time stamping of individual device states on the driver level. It offers access to data as a single-state property.

The ViSTA toolkit, currently at v2.0.0 (Assenmacher & Kuhlen, 2008), implements a two-level driver architecture, differentiating between a physical device and a logical device. In ViSTA, device drivers contain a set of logical sensors, each providing a history of snapshots of the device state. The logical device consists of a data flow network for transformation of the device data. Changes to node states in the network can generate events or change the application state. In its current form, the device driver layer supports explicit device-dependent decoding as well as client-sided transcoding. Aspects are implemented as a hierarchy of dynamically associated properties of a driver. ViSTA supports detection-sourced time stamping with per-reading-snapshots on the driver level.

6 Summary

Input and output devices are at the heart of every VR interaction paradigm. On the one hand, they are objects of research, for example when developing novel devices for specific interaction tasks. On the other hand, commercially available devices have to be integrated and used in order to implement even basic VR applications. With this paper, we identify terms needed when discussing an implementation of a device driver architecture for VR systems. Processing of multimodal data types, preserving temporal access across a number of calls, and concurrency-enabled exchange structures are the most important elements presented.

As a conclusion, it can be said that no perfect platform seems to exist which matches all requirements of device handling and data processing as discussed in this paper.
Libraries such as VRPN help in application development as they are easily integrable into existing environments and offer a large number of supported devices. Other approaches may be more suited to deal with specific problems, but are tied to their VR systems and cannot be extracted easily.

A theoretical model for driver architectures helps to qualify and review existing implementations, especially for comparison and reasoning on software engineering aspects.

References


