

OpenTracker - A Flexible Software Design for Three-Dimensional Interaction

Gerhard Reitmayr
University of Cambridge
Cambridge CB1 2PZ, UK
gr281@cam.ac.uk

Dieter Schmalstieg
Graz University of Technology
8010 Graz, Austria
schmalstieg@icg.tu-graz.ac.at

Abstract

Tracking is an indispensable part of any virtual reality and augmented reality application. While the need for quality of tracking, in particular for high performance and fidelity, has led to a large body of past and current research, little attention is typically paid to software engineering aspects of tracking software. To address this issue we describe a software design and implementation that applies the pipes-and-filter architectural pattern to provide a customizable and flexible way of dealing with tracking data and configurations. The contribution of this work cumulates in the development of a generic data flow network library called *OpenTracker* to deal specifically with tracking data. The flexibility of the data flow network approach is demonstrated in a set of development scenarios and prototype applications in the area of mobile augmented reality.

1 Introduction

Tracking is an indispensable part of any virtual reality (VR) or augmented reality (AR) application. Processing of tracking data requires operating of devices, reading specialized network protocols, performing calculations to fuse data from different sources and interpret it to provide multi modal interaction.

As the development of interactive applications moves beyond building a single demonstration setup, a number of requirements appear. Different tracking devices need to be supported because an application setup will be updated with new devices or installed at different sites. Complex setups require more than one host computer and therefore distribute the processing of tracking data between different software components. Experimental tracking configurations and combinations of hardware require frequent changes to the tracking software.

A focus of multi modal interaction in VR and AR systems is the combination of multiple tracking systems to provide novel interaction techniques or extend the functionality of a single system. Such approaches require appropriate sensor fusion of multiple systems and a flexible integration of these systems into the

application. Current software offloads these tasks to the application instead of encapsulating them within the actual configuration used in a setup.

Some current systems have a modular approach that allows to substitute one type of tracking device for another. Typically, commercial VR products take this approach offering turn-key support for many popular tracking and input devices, but at the cost of a limited amount of extensibility and configuration options. In particular, they make it hard to combine existing features in novel ways.

In contrast, research systems may offer features not found in commercial systems, such as prediction or sensor fusion, but are usually limited to their particular research domain and not intended for the end user. In such systems, replacing a piece of hardware or changing its configuration usually leads to rewriting a significant portion of the tracker software.

In the middle(-ware), there is a lack of tools that allow for a high degree of customization, yet are easy to use and to extend. What is needed is a system that allows mixing and matching of different features, as well as simple creation and maintenance of possibly complex tracker configurations. To address this issue we describe a software design and implementation that applies the pipes-and-filter architectural pattern [5] to provide a customizable and flexible way of dealing with tracking data and configurations.

1.1 Design requirements

We established a number of requirements for an independent software component for processing tracking which were derived from the experience of researching collaborative augmented reality applications for a number of years in the Studierstube project [17].

Device abstraction. Typically, a variety of devices provide the same or similar data, therefore abstraction from an individual device would be beneficial to the reusability and portability of an application.

Support for complex configurations. Processing input data requires more than reading it from input devices. Calibration and registration of tracking data is mandatory in most applications. Tracking data from different devices can be combined to appear as a single more complex device. However, typically the combination of data from different devices is a non-trivial problem due to different measurement modalities, update rates or error properties.

Network transparency. Collaborative and distributed applications require network transport of tracking data. However, applications should be developed independently of the actual configuration used in any given setup. Therefore, a dedicated tracking software layer should provide support for network transparency.

Low overhead and latency. Applications require timely delivery of tracking data to reduce the end-to-end latency perceived by the user. Therefore, any tracking software should only add minimal processing time to the overall process.

Support for iterative development. Configurations should be simple to author and change. Research requires frequent changes to installations and configura-

tions and the tracking software component should separate the application's core from such changes, if possible.

Simple to integrate. To ensure a broad applicability the software component should not dictate a certain software architecture but allow for integration into different styles of application architectures. Use in both simple main-loop driven applications and frameworks employing inversion of control style architectures should be possible.

Allow for extensibility. Tracking hardware itself is a moving target and new devices need to be supported by implementing new device drivers. Multi-modal interaction requires development of new and experimental algorithms to process input data. Both types of functionality should be simple to add to a dedicated software component.

1.2 OpenTracker approach

The contribution of this work is the development of a generic data flow network library called *OpenTracker* to deal specifically with tracking data. It is built on a number of key observations:

- Abstraction of recurring operations on tracking data separates applications from concrete devices and configurations and yields better code reuse.
- Flexible arrangement of such operations simplifies experimentation with and development of VR and AR applications.
- A dedicated configuration language allows simple authoring and configuration of complex setups.
- An extensible software design supports rapid implementation of new functionality and device support.

In a typical VR or AR application tracking data passes through a series of steps. It is generated by tracking hardware, read by device drivers, transformed to fit the requirements of the application and send over network connections to other hosts. Different setups and applications may require different subsets and combinations of the steps described but the individual steps are common among a wide range of applications. Examples of such invariant steps are geometric transformations, Kalman filters and data fusion of two or more data sources.

The main concept behind OpenTracker is to break up the whole data manipulation into these individual steps and build a data flow network of the transformations. Moreover, it abstracts from the details of accessing and manipulating tracking data by forming an architectural layer between the tracking devices and the application (see Figure 1). To describe the details of this concept, we will need some theoretical definitions which are discussed in section 3. Details of an actual implementation are described in section 4. Some example configurations in the area of mobile augmented reality are discussed in section 5 to demonstrate the features of OpenTracker.

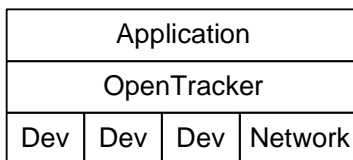


Figure 1: A block diagram detailing the relationship of OpenTracker with other software components. Device drivers and network connectivity are separated from the application sitting on top of OpenTracker.

2 Related work

Device abstraction is a standard requirement for 2D graphical user interfaces (e.g. GKS [10]), and sometimes incorporated into 3D applications [7]. Many interactive systems employ sophisticated event handling schemes. State changes to attributes of scene objects are either propagated by functional dependencies (e.g. routes in VRML [6], engines in Open Inventor [21]), or may be handled by user supplied callback functions (e.g. script nodes in VRML). These approaches inspire the architecture of OpenTracker, although none of them deals specifically with tracker configurations.

An early example of a software toolkit dedicated to developing interactive and immersive graphics applications is the MR Toolkit [19]. It provides device abstraction and network transparency for tracking devices. A similar development is the GIVEN++ toolkit [20] which supports multiple input devices in a distributed framework.

The Virtual Reality Peripheral Network (VRPN) [22] is a C++ library implementing device abstraction for a large number of tracking devices and also networking support based on tracking servers and application clients. It defines a small set of data types that can be reported by a device through individual facets. VRCO trackd is a commercial tracking device software framework [25]. A central server process implements device drivers and provides device abstraction and network transparency to applications that connect to the server process.

All these software libraries provide device abstraction and network transparency, both of which are already important features of a re-useable software design for interactive applications. However, they do not provide for more advanced operations on tracking data and simple configuration thereof.

Two notable software frameworks were developed after a first publication of our work on OpenTracker [15] and draw some inspiration from the data flow approach presented there.

The Virtual and Augmented Reality Input Output (VARIO) framework [18] implements a generic flow scheduling framework with distributed components that can reside on different hosts. A central configuration process manages the connections between components and the configuration of individual components. Configurations are made persistent by saving and loading descriptions of

the connection and configuration parameters to and from XML files.

The DWARF project [2] aims for a design concept that differs greatly from traditional AR software designs. The basic units of the DWARF framework are distributed services. A *service* is a process running on a stationary or mobile computer that provides a certain piece of functionality such as optical tracking. Services can be connected to use the functionality of other services establishing a data flow network to achieve a more complex function. Dedicated tracker services operate tracking devices and provide raw data to other services which can perform additional transformations or constitute part of an application. Configurations are created in an indirect way by specifying properties on services that describe the conditions for connections to be made.

VARIO and DWARF both support data flows for processing tracking data. Both are based on a component system which models each processing step as an individual process communicating with other processes. Such a solution trades off increased flexibility for additional communication overhead. The individual processing nodes are heavyweight and can lead to increased latency in VR and AR applications.

The earlier related work demonstrates the need for some software design to provide device abstraction or network transparency for developing applications. The use of data flow networks to describe processing steps of tracking data has become a standard approach since the inception of OpenTracker. In contrast to the later implementations OpenTracker presents a lightweight approach that does not force a specific framework on the application. It also provides a direct scripting approach to support simple configuration of the data flow network.

3 Concepts

Each unit of operation in OpenTracker is represented by a node in a data flow graph. Nodes are connected by directed edges to describe the direction of flow. The originating node of a directed edge is called the child whereas the receiving node is called the parent. To allow more than simple linear graphs, we introduce ports, references and edge types as follows.

3.1 Multiple Input Ports and References

Each node has one or more input ports and a single output port. A port is a distinguished connection point for an edge, i.e. the node can distinguish between events passing through different node ports. The output port of one node is connected to any of the input ports of another node. This establishes the flow by defining directed edges in the graph. A node receiving a new data event via one of its inputs computes a new update for itself and sends the new data event out via its output port.

Multiple input ports are desirable because computations typically have more than one parameter. Dynamic transformations, for example, are parameterized by the value of another node and thus use the data value received by a child

to be transformed differently from the data of the parameterizing child. Merge nodes may select part of the data of an event based on the input port the event used. Combinations allow more complex computational structures.

Additionally, an input port can be connected to several output ports to provide fan-in of events. Thereby several children nodes are connected to the same input port of a node. Upon receiving an event, the parent node can only distinguish between the input ports, but not between the actual children. Fan-In of several events is accomplished by serializing the events and the parent operates on each event in turn.

Conversely, an output port can also be connected to several input ports of other nodes to provide fan-out of events by using references within the graph. Again the multiple connections are transparent to the node which cannot selectively send events to only one parent, but all events are distributed equally to all parents.

Figure 2 gives some examples of data flow graphs that can be build with OpenTracker. Part (a) shows a simple linear graph applying a geometrical transformation to a data source, (b) shows a node with several input ports, combining the received data. Part (c) is a graph using a reference node to get a copy of the output of a node and (d) combines these features in a more complicated graph.

3.2 Edge types and time

The basic mechanism behind the data flow concept is event passing. Data events are passed from the children nodes upward to their parents. However, not all computations fit well into this model: Algorithms that operate on a list of tracker measurements or that compute the tracker state at an arbitrary point in time require different types of input or output interfaces. Examples are filtering algorithms that take a history of events into account, or prediction algorithms that compute an expected measurement for a given point in time.

To support different ways of incorporating time into computations, we also distinguish between different edge types. Edges are typed by typing the ports of the nodes they connect. We establish the rule that only two ports of the same type can be connected and this type is the type of the edge. There are three edge types: *event*, *event queue* and *time dependent*.

event edges implement the typical event push pattern as new events are pushed from children to their parent nodes. Each event is time stamped by the individual node that generated it. Thus nodes can react to the temporal aspects of tracking data. For example, a simple prediction node incorporates the time difference between single events to correctly update its output.

event queue and *time dependent* edges provide a polling pattern for data processing. These interfaces are polled by the parent node, because the data returned is parameterized. The *event queue* interface represents a queue of events and supports querying the number of stored events and retrieving them by index. The *time dependent* interface can be queried by specifying a point

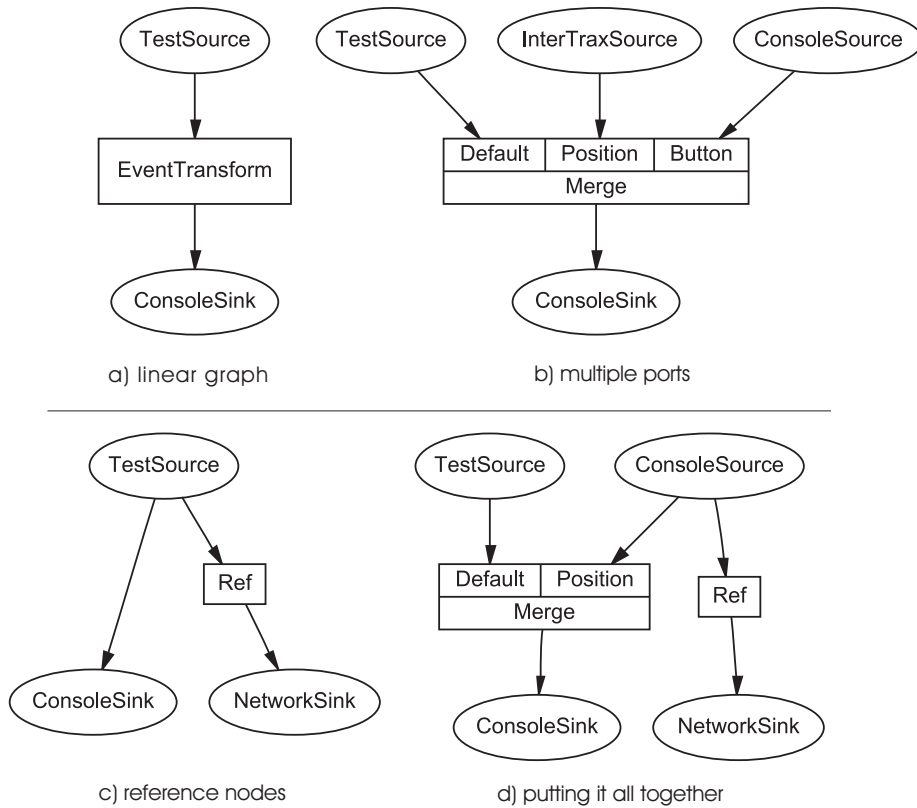


Figure 2: Visualizations of a data flow graphs as used in OpenTracker. (a) A linear flow. (b) A node with different input ports. (c) Fan-Out of output ports. (d) A complex example employing all features.

in time, for which the appropriate data value is returned. How this value is calculated depends on the node's implementation.

The latter two interfaces are used to implement nodes that perform computations on a set of event values such as windowed filters or that use a point in time as parameter such as a prediction for a given point in time. Only specialized nodes implement these interfaces while the majority of nodes only supports the *event* interface.

4 Implementation

In an actual implementation we distinguish *source nodes*, which are leaves in the graph and receive their data values from external sources, *filter nodes*, which are intermediate nodes and modify the values received from other nodes, and *sink nodes*, which propagate data values to external outputs.

| Component | Description |
|-------------|--|
| position | 3 component vector describing a position in space. |
| orientation | 4 component vector describing a rotation as a quaternion. |
| button | 16 bit integer value describing the state of 16 button. |
| confidence | floating point value in the interval $[0, 1]$ describing the quality of the measurement. |
| time | time stamp giving the time of measurement. |

Table 1: Components of the OpenTracker event data type.

The data type passed between nodes is a complex data structure tailored towards the requirements of AR applications and consists of a fixed set of components (see Table 1). Although this restriction to a fixed data type appears as an limitation, it can easily be extended or generalized because nothing in the supporting system relies on the type of the event data.

4.1 Source Nodes

Most source nodes encapsulate a device driver that directly accesses a particular tracking device, such as a Polhemus or Ascension tracker connected to a serial interface. Other nodes objects form bridges to complex self-contained systems, such as the video tracking library from ARToolkit [11] or implement a DWARF service interface [3]. A third type of source node emulates a tracker via the keyboard, access network data (see section 5.2) or simply responds with constant values (useful for development and debugging).

The default implementation for most source nodes only provides the last data received from either a tracking device or the network. Some source nodes have a multi-threaded execution model to implement an efficient decoupled simulation model [19] (e.g. , when blocking I/O must be used to poll a device).

4.2 Filter Nodes

Filter nodes receive values from one or more child nodes. Upon receiving an update from one or more of their children, they compute their own state based on the collected data. A non-exhaustive list of filters includes:

- Transformation filters perform geometric transformations of their children’s values. These include pre- and post-transformations and may be static or depend on data values received from other children. The latter allows to modify the filtered state relative to another tracker state.
- Button filters perform boolean operations on the button state of different input sources to combine them into a new event value.
- Prediction filters allow to partially compensate for lag in the measuring and processing tracker data.

- Noise and smoothing filters are handy to deal with inherent inaccuracies of trackers.
- Undistortion filter are necessary e.g. to linearize distortions in the magnetic field of a magnetic tracking device.
- Merge filters assemble new data values using different parts of the data values of several children. Use cases include the combination of orientation from an inertial tracker with position information from an acoustic tracker, or adding a button device to a closed tracking solution such as Polhemus Ultratrak.
- Conversion filters are able to translate one data type into another. For example, 2D positions from a desktop pointing device can be translated into 3D positions by adding a constant third value.
- Clamp filter are special nonlinear transformation filters that cut off values at user-specified extrema, for example to deliberately limit interaction to a valid range.
- Confidence filters select data values from different children based on some measure of confidence in the accuracy of the data.

4.3 Sink Nodes

Sink nodes are similar to source nodes but distribute data rather than receive it. They include output to network multicast groups, debugging output to a user interface or thread-safe shared memory output to integrate OpenTracker as a library into other applications. A logging node writes the received data into files to record a stream of data which can be played back a corresponding file source node.

4.4 Time

Time is reflected in several ways in the architecture of OpenTracker. As described in section 3.2 the type system for edges supplies us with different ways of dealing with time, either having an event based approach, with or without queueing of events, or by specifying functions of tracking data as continuous functions of time. OpenTracker does not implement any clock synchronization of different hosts working together in a network. There are well established means to solve this problem such as the NTP protocol [1].

4.5 Software architecture

The intent of OpenTracker is to provide an auxiliary library that is to be integrated into VR or AR applications. Therefore it is kept lightweight and customizable. The library is designed as a class hierarchy of tracker objects,

implemented in C++. It is build around a small set of core classes that implement the basic node interfaces, a parser that builds the runtime structure from a configuration file and the main loop driving the event model. Any other functionality is implemented by a set of module classes that can be easily extended or modified.

The library is extensible through the use of an abstract *NodeFactory* interface to define the class interface for creating new nodes and through the *Module* class that provides an interface for processing within the main loop. Any extension adds new node types by providing an object that implements the *NodeFactory* interface. The object is added to a list of factories known by a *Context* object at startup and can then create nodes of the new type as requested by the parser. Parameters for node creation are passed in by the parser as a generic map of key-value pairs.

To add more complex functionality such as device drivers a subclass of *Module* is created and added to the list of modules known to a *Context* object. The modules are called regularly during processing of the main loop. Within these callbacks they can implement any processing and create new events. These events are then propagated into the data flow network by associated nodes. Events can also be read in from the network by these nodes. A module obtains references to the nodes it is interested in by implementing the *NodeFactory* interface. It acts thereby as both the creator and the active implementation of the nodes.

There are also nodes that perform without an underlying module. Examples are filter nodes that implement geometric transformations on incoming events and pass the transformed events to their parents.

There is no fixed interface to the integrating application in order to maximize flexibility. Application programmers either have to use one of the supplied nodes (such as a generic call back node) or to supply their own module implementing sink nodes as interfaces to their application. Moreover, the use of the library main loop is not mandatory. The processing can be integrated with the application's main loop to avoid additional threads and synchronize the tracking data processing more closely with the application. These design decisions ensure that the library can adapt to the needs of any application.

Figure 3 shows a class diagram of the core classes. The class *Context* implements the main loop and keeps reference of all modules and the data flow data structure. It employs an object of class *ConfigurationParser* to parse the configuration files. Actual node implementations are derived from *Node*, for example the *Transformation* or the *TestSource* class. *WrapperNode* and *RefNode* are special nodes that implement the port and reference functionality. *State* is the default event type.

4.6 Software engineering with XML

XML, the eXtensible Markup Language[4], is a markup definition language that allows to define hierarchical markup languages with so-called document type definitions (DTD). With the appropriate DTD, standard XML tools can be

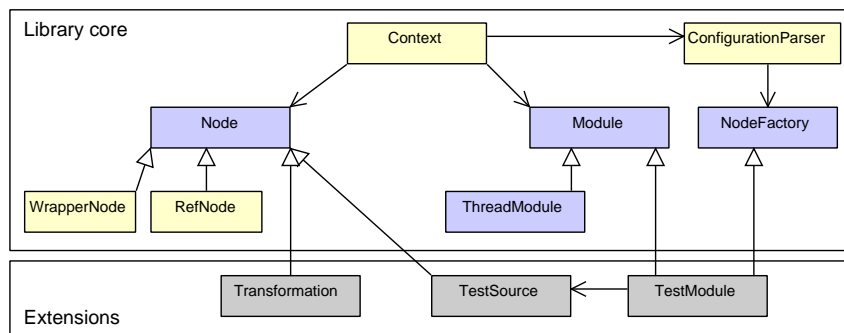


Figure 3: Class diagram of the OpenTracker library. Light gray classes are extended to add new nodes and modules. Dark gray classes are examples of extensions. Arrows with hollow heads describe derivation, other arrows associations.

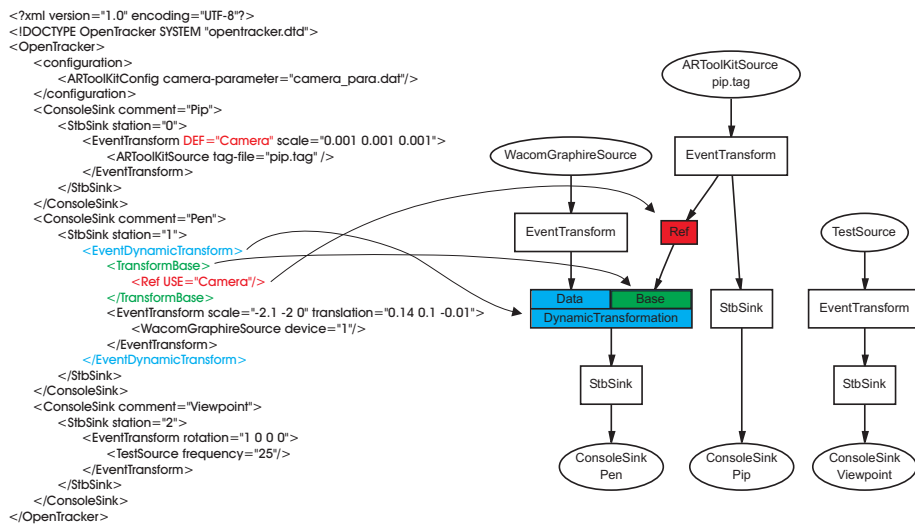


Figure 4: An example configuration file and the corresponding data flow graph. The use of Ref nodes and multiple input ports is highlighted.

used to conveniently edit, type check, parse, and transform any XML file. Thus, providing a simple DTD for describing the data flow graphs of tracker nodes opens access to software libraries and tools that simplify several steps of the development cycle. A visual DTD editor can be used to design and maintain the DTD. An XML parser [23] enforces content format on the tracker configuration file while building the corresponding structure in memory, thus automatically performing many of the consistency checks that otherwise have to be hand-coded. A convenient XML editor such as [9, 8] allows the end user to design the tracker configuration without having to master the syntax while enforcing the correct content format, reducing syntax and semantic errors.

Configuration files for OpenTracker are written in a dedicated XML language defined by the DTD. Elements correspond to nodes in the dataflow and attributes to parameters of such nodes. The parent - child relationship of the data flow graph is directly mapped onto the parent - child relationship of XML elements. The content model of the language enforces interface and semantic constraints on the specified graph. As described in section 3.2 edges and the corresponding node ports are typed and therefore restrict the possible combinations in the construction of the graph. These constraints and others on the number of children are described in the DTD. For example, source nodes typically do not have any children as they rely on data from external sources to compute their own data. In contrast, confidence filters use any number of children to compute their data value.

The reference structure is created by using unique ID attributes on elements and referencing these IDs in reference elements. While children of nodes with

only one input port are directly mapped to children elements in the XML file, multiple input ports need to be addressed differently. Children that are connected to a specific input port are wrapped by an additional XML element which in turn is the direct child of the node of interest. These elements are mapped to special wrapper nodes that can be distinguished by the node implementation. Otherwise they are transparent to the actual data processing.

Figure 4 gives an example of such a configuration file, using all of the features described before. The interesting constructs are highlighted and cross linked with the corresponding nodes in the resulting data flow graph.

4.7 Data flow implementation

The implementation of the node graph and the data flow of events is directly based on the Document Object Model (DOM, [27]) structure provided by the XML parser library Xerces [23]. The library reads in a configuration file and constructs a tree structure in memory representing the elements and attributes and the relations between. OpenTracker reuses the in-memory DOM tree and decorates it with instances of the node types described above. A DOM node provides a facility to store a mapping of names to pointers to user data and OpenTracker stores the pointer to the node instance associated with a certain element in the configuration file in this map (see Figure 5).

Tracking events flow through the network via a push and a pull mechanism. The *event* interface uses a push mechanism, that passes the current event from the source nodes via any intermediary nodes to the sink nodes. Every node calls an update method on its parent which recursively calls its own parent's update method after processing the event (see Figure 6(a)). Thus, the modules associated with the source nodes only have to trigger this event propagation by calling the update method on their source nodes. Only a reference to the object storing the event data is passed. A node that changes the event's data has to provide a new instance to avoid changing an instance that may also be used by other nodes. This instance is typically a member of the node reused to avoid frequent allocation and deallocation of an event object on the stack.

The *event queue* and *time dependent* interfaces use a pull mechanism. If a node is queried via one of these interfaces and it requires event information from any children nodes upstream in the network, it recursively calls the childrens' interfaces with the appropriate parameters (see Figure 6(b)). Pulling data is typically not implemented for network nodes to avoid incurring large delays and therefore is only happening within one process. Again all nodes have to provide their own object instances to avoid side effects by shared event instances.

Not all nodes implement all interfaces. The *event* interface is the standard case and is implemented by most nodes. The remaining interfaces are only implemented in a small subset of nodes that use it to implement more complex behaviors. For example, an *EventQueue* node stores a queue of the last events pushed through it and exposes the queue through the *event queue* interface. Another node called *Filter* then uses an *EventQueue* node to implement a linear filter over the last events that is triggered by any event pushed through it.

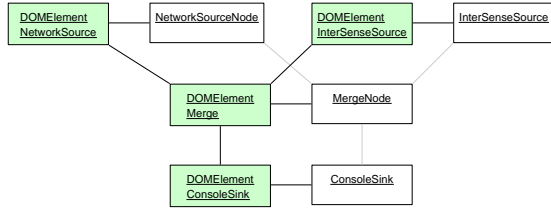


Figure 5: The DOM tree elements in green are decorated with the white OpenTracker nodes. The links between the nodes are only virtual and are inferred from the DOM structure.

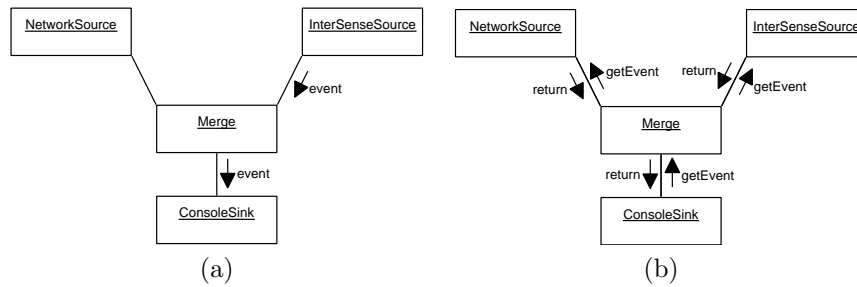


Figure 6: Two types of data flow in OpenTracker. (a) Events are pushed by source nodes through the graph. (b) Events are pulled by sink nodes.

5 Results

We continue with a description of some results showcasing the flexibility of OpenTracker and giving an overview of some applications that were implemented using it.

5.1 Rapid prototyping

Modifying and refining a user interface and its underlying tracking configuration are common tasks in developing research prototypes of VR and AR applications. During the development of a mobile system described in more detail in section 5.3 a number of systems were created that provided different user interfaces. The system described in this work uses a pen and a graphics tablet as main input devices. Another version used a pair of tracked gloves [24] and a touch pad. Here one glove mapped to the pen in the original setup and the touch pad controlled a 2D cursor mapped into 3D to replace the tracked graphics tablet. Therefore all applications working on the original system could also run on the changed one. Only changes to the OpenTracker configuration were necessary.

Using an application in different configurations also requires flexibility within the tracking software framework. Examples are a debug scenario running on a desktop machine versus an installation using a dedicated 6DOF tracker and head-mounted displays. Our group develops a complex augmented reality application for 3D geometry education [12] working in a laboratory setup with six inputs tracked with 6DOF. However development mostly takes place on a desktop setup where the inputs are mapped to virtual devices controlled by keyboard or mouse input. The logging features of OpenTracker also allow to record tracking data of individual sessions which helps in debugging wrong behavior occurring only during certain movements of the users. The tracking data can be replayed on the desktop in real time or at accelerated speeds to provide reliable test cases and speed up development. In all these cases only the tracking configuration is modified, never the application itself.

5.2 Distributed tracking

OpenTracker implements a pair of source and sink nodes that transport tracking events over the network. These nodes allow multiple senders and receivers of tracking data to communicate asynchronously by using IP multicast. It is even possible for a single host to operate as a sender and receiver at the same time, by picking up data, then modifying it and re-sending it to the network on another network channel. Thus, larger networks spanning multiple hosts can be created. There are several reasons why it is desirable to share tracking data over a network.

Multi-processing based on inexpensive PCs becomes possible with little configuration effort. This is useful to achieve some degree of load balancing. In particular, computationally expensive functions such as filtering or undistortion can be assigned to either sender or receiver, depending on the computational

budget. Also, network support makes it easy to span multiple operating systems, in particular if a specific tracking device or service is only available at one particular host.

Examples of such configurations are used in our laboratory. A dedicated host operates the tracking device and runs some filters on the tracking data. Individual development workstations running the actual applications receive the data. Therefore we can update the tracking configuration without interfering with the applications themselves or implement dedicated additional transformations for each individual application in its local OpenTracker configuration.

Transparent substitution of tracking devices enables to switch devices during run-time or to use virtual devices for testing and playback of prerecorded or generated tracking data. For example, an outdoor mobile AR system we built communicates with a dedicated tracking process that operates a GPS receiver and inertial tracker to provide position and orientation information. The configuration of this process can be switched to a different one which provides information from a user controlled simulator without stopping the application itself.

While there is a preferred network protocol for OpenTracker, support for additional formats can be easily implemented. So far, we implemented support for VRPN and DWARF allowing an OpenTracker based process to both receive and send data to and from VRPN or DWARF based processes. Such a process could be used to perform some registration and filtering computations in an otherwise VRPN based setup.

5.3 Mobile augmented reality system

Augmented reality setups often require the integration of various different tracking devices. We built a series of mobile AR systems to investigate mobile collaborative applications [14]. The first iteration consisted of a PC notebook equipped with a NVidia GeForce2Go video chip and a 1GHZ processor and worked under Windows 2000. As an output device, we use an Sony Glasstron see-through stereoscopic color HMD. The display is fixed to a helmet worn by the user. Moreover, an InterSense InterTrax² orientation sensor and a web camera for fiducial tracking of interaction props are mounted on the helmet. The setup is carried by the user in a backpack.

The main user interface is a pen and pad setup using a Wacom graphics tablet and its pen. Both devices are optically tracked by the camera using markers. The 2D position of the pen (provided by the Wacom tablet) is incorporated into the processing to provide more accurate tracking on the pad itself. Figure 7 gives an overview of the setup.

Tracking of the user and the interaction props is achieved by combining data from various sources. The OpenTracker component receives data about the user's head orientation from the InterTrax² sensor to provide a coordinate system with body stabilized position and world stabilized orientation.

Within this coordinate system the pen and pad are tracked using the video camera mounted on the helmet and ARToolKit [11] to process the video infor-

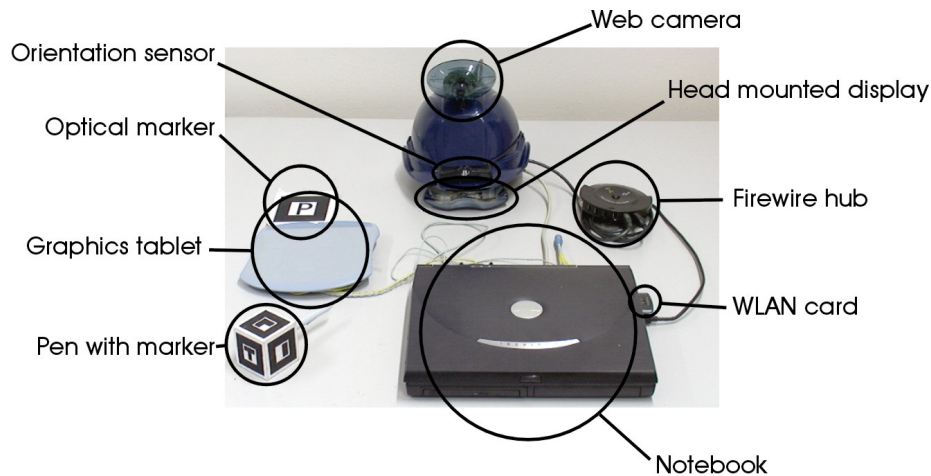


Figure 7: Hardware components of the mobile AR setup.

mation. Because the video camera and the HMD are fixed to the helmet the transformation between the cameras and the users coordinate system is fixed and determined in a calibration step.

The pad is equipped with one marker. This is enough for standard operation, where the user holds it within her field of view to interact with 2D user interface elements displayed on the pad. The pen, however, is equipped with a cube featuring a marker on the five sides which are not occluded. This allows to track the pen in almost any position and orientation. Moreover whenever the user touches the pad with the pen the more accurate information provided by the graphics tablet is used to set the position of the pen with respect to the tablet.

The data flow graph describing the necessary data transformations is shown in Figure 8. Round nodes at the top are source nodes that encapsulate device drivers. The round nodes at the bottom are sinks that copy the resulting data to the AR software. Intermediate nodes receive events containing tracking data, transform it and pass it on, downwards. An important type of transformation is the relative transformation that takes input from two different devices and interprets the location of one device relative to the location of the other (called the *base*).

Different colors denote paths through the graph that describe how the tracking data for different devices are processed. Relative transformations are marked by cross stripes in the color of the two paths connecting. For example, the *optical pen* path describes the five markers that are each transformed to relate the pen point location. The results are merged, then further transformed. After another merge with data from the graphics tablet, it is once more transformed to the reference system established by the orientation sensor.

Similarly, the *optical pad* path describes the computation to obtain the lo-

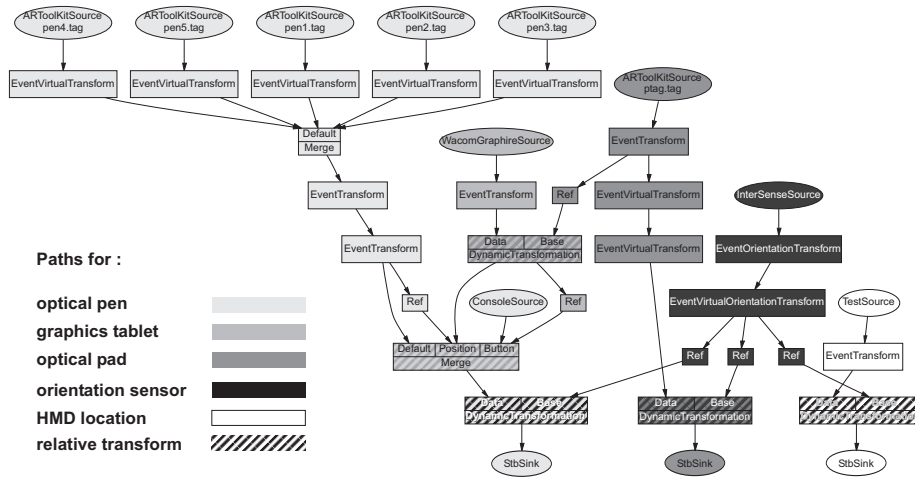


Figure 8: The data flow graph of the tracking configuration for the mobile AR setup. Individual flows are indicated per source. The diagram was automatically generated from the XML configuration description.

cation of the pad. As a side effect, the *optical pad* information is used at one step to transform the 2D information from the *graphics tablet* path to the actual pen position which is subsequently merged with the pure optical information. Finally the white *HMD location* path is used to provide information about the head location. The *TestSource* node’s task is to provide a constant value which is then transformed by the orientation sensors.

We would like to note that using a visual XML editor, this complex configuration was created without writing a single line of code.

5.4 Indoor wide area tracking

To build an environment where we could test drive our mobile AR kit, we implemented an indoor tracking solution to cover a floor of our building. As we did not have access to a proprietary building-wide positioning infrastructure (such as AT&T Cambridge’s BAT system used by Newman et al. [13]), we choose to rely on a hybrid optical/inertial tracking solution. This approach proved very flexible in terms of development of positioning infrastructure, but also pushes the limits of what the used optical tracking library ARToolkit can provide. A more detailed account can be found in [16].

To implement a wide area indoor tracking solution we resolved to use a set of well-known markers that were distributed in the environment. Together with a geometric model of the building that includes the location of the well-known markers (see Figure 9) we can compute the user’s location as soon as a marker is tracked by the optical tracking system. These models and the location of the markers were obtained by manual measurements with a survey instrument.

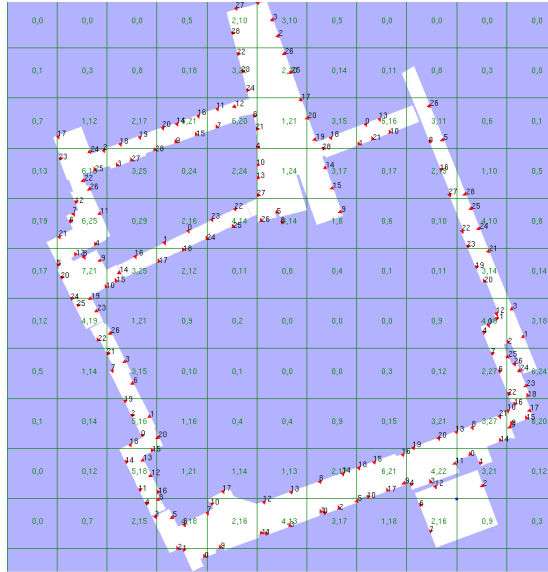


Figure 9: The diagram shows the geometric model of the floor. The dark dots denote the locations of measured markers used to track the user within the environment. The overlaid grid marks the cells defining unique marker sets.

The implemented tracking approach requires a large set of markers. It is necessary to place a marker about every two meters and to cover each wall of a single room with at least one marker. Deploying it in our floor covering about 20 rooms and long hallways would require over two hundred different markers. However, marking up a large indoor space with unique ARToolkit markers is not feasible. The more markers, the higher the degree of similarity of any pair of markers will be leading to false recognitions. A large set of markers also enlarges the search space that ARToolkit has to traverse, leading to significant decrease in performance. Consequently, the use of ARToolkit is not feasible for large marker assemblies.

To overcome this restriction, we developed a space partitioning-scheme that allows reusing sets of markers within the given environment. The idea behind this approach is that, if the tracking system knows the user's location, it can rule out large parts of the building because they will not be visible to the camera. Therefore, for two areas, which are not visible to each other, it becomes possible to use the same set of markers.

To compute a possible placement of markers, the space of the model is partitioned into a 3-dimensional cubic grid of a fixed length (see Figure 9). Then, marker patterns are assigned to the measured positions such that the patterns are unique within each cubic cell and its 26 direct neighbors. Basically, every cell defines a partial mapping from marker patterns to marker positions in the environment.

The tracking computes the user’s location from a known cell and a marker pattern observed by the camera mounted on the helmet. Because the pattern is unique within the cell and its neighbors, the associated marker position can be established and the user’s location is computed by concatenating the marker position and the relative measurement computed by the ARToolKit library. These computations are executed within OpenTracker by configuring appropriate transform nodes.

The representation of the mapping from marker patterns to marker positions within the cells is implemented with the help of a special node within OpenTracker called *GroupGate*. A GroupGate defines a gate that passes incoming events on, if enabled and stops them otherwise. A set of GroupGate nodes is configured into an directed graph describing a neighborhood relationship. The relationship is typically symmetric but need not be. At any point in time only one GroupGate node is denoted as active. If a GroupGate node is active, it is enabled and all its neighboring GroupGates are enabled as well. All other nodes are disabled and will not pass events. A GroupGate becomes active, if it is enabled and an event passes through it. Additionally, a GroupGate also defines a second input port named *override*. If an event passes through the *override* port, the node is also activated.

A single cell is modelled as a GroupGate and its 26 neighboring cells are configured as neighbors. The measured data from a single marker pattern is passed through all possible transformations for the different maker positions it is used at. Then, each transformed data event is passed through the GroupGate of the cell the marker position is associated with. The active GroupGate corresponds to the cell the user is currently in. Because events can only pass through the active GroupGate and its neighbors, data from a marker will be used only once. Moreover the data passes only through the GroupGate associated with the marker position of the last seen marker activating it if necessary. Thus the activation will always shift with the user’s movement through the set of GroupGates. Figure 10 gives an example of the use of the GroupGate node.

The tracking needs an initial position at startup to set the first active cell and GroupGate. A set of unique markers are used to define starting positions on individual floors and the user can select her current position at any time to correct any errors. Both operations simply activate the correct GroupGate to set the current cell.

6 Conclusions and future work

OpenTracker is the first software framework to thoroughly apply the pipes-and-filters architecture to the problem of manipulating tracking data. The resulting advantages are twofold. The high-level language introduced to configure the processing of tracking data simplifies experimental and exploratory programming of data manipulations. Describing the configuration in a dedicated language renders it also more accessible to automated methods such as generating a certain configuration.

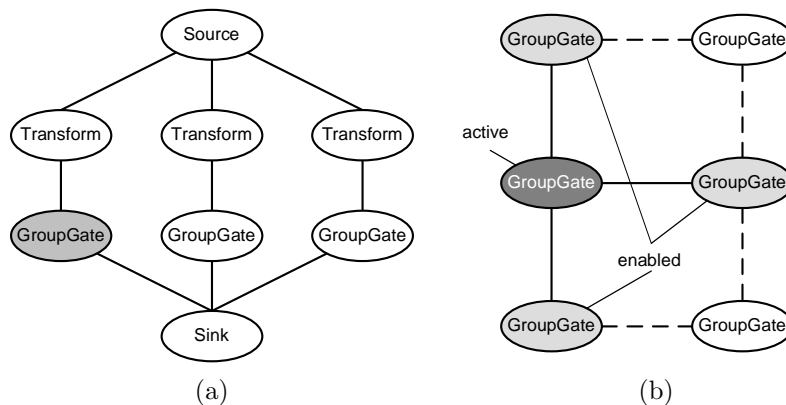


Figure 10: The function of the GroupGate node. (a) A set of GroupGates allows to select a single path from a set of paths through the graph. Only the active GroupGate propagates events. (b) The GroupGates are in a neighborhood relation. The active GroupGate’s neighbors are enabled to propagate events as well and become active, if an event passes through them.

The layered architecture that OpenTracker enforces on the overall application provides a clear cut interface between the application logic and the functions required to deal with tracking devices. A number of issues appearing in relation with tracking devices such as calibration and registration, network transparency or fusion of input data can be dealt with in a way that is transparent to the application. Decoupling the application specific functionality from the device layer also furthers reuse of the application in the context of different tracking systems. While the existing nodes in OpenTracker will not cover the requirements of every application, the extensibility guarantees that new functions can be implemented rather easily. By extending OpenTracker rather than hard-coding the application, new functionality can be reused and is not locked into a single application.

The current OpenTracker implementation has a set of shortcomings. The data type processed is a fixed structure tailored towards a specific application. An extension to different data structures will enable multi-modal processing of input data and expand application area of the OpenTracker concept. Runtime reconfiguration of the tracking graph would allow a number of interesting applications. A dedicated tracking configuration tool can build up a configuration based on user input and simplify setting up an AR system. Auto-calibration of a running system becomes possible and would improve the registration errors of the computer generated images transparently and without intervention by a human operator. These issues are addressed in the next generation of OpenTracker which is under active development.

A more ambitious research direction is Ubiquitous Tracking [26] which aims to provide an ubiquitous infrastructure service to AR applications. An application can register with a UbiTrack service and request tracking information on

objects it is interested in. The service would then automatically compute a configuration based on the available tracking devices and send it to the application. The next version of OpenTracker will then use the configuration to provide the required tracking data to the application. The actual tracking devices and required configuration would be transparent to the application and could change at runtime as required.

acknowledgement

The presented work was sponsored by the Austrian Science Foundation *FWF* under contracts no. P14470 and Y193, and Vienna University of Technology by Forschungsinfrastrukturvorhaben TUWP16/2002. We would like to thank Michael Knapp and the students of the Virtual Reality lab course for their dedicated work on the building model.

OpenTracker is available as an Open Source software project from the web address <http://www.studierstube.org/opentracker/>

References

- [1] NTP: The network time protocol. <http://www.ntp.org/>, February 16 2004.
- [2] Martin Bauer, Bernd Bruegge, Gudrun Klinker, Asa MacWilliams, Thomas Reichner, Stefan Riss, Christian Sandor, and Martin Wagner. Design of a component-based augmented reality framework. In *Proc. ISAR 2001*, pages 45–54, New York, New York, USA, October 29–30 2001. IEEE and ACM.
- [3] Martin Bauer, Otmar Hilliges, Asa MacWilliams, Christian Sandor, Martin Wagner, Gudrun Klinker, Joe Newman, Gerhard Reitmayr, Tamer Fahmy, Thomas Pintaric, and Dieter Schmalstieg. Integrating Studierstube and DWARF. In *Proc. STARS 2003*, pages 1–5, Tokyo, Japan, October 7 2003.
- [4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, et al. Extensible markup language (XML) 1.0. <http://www.w3.org/TR/REC-xml/>, October 6 2000.
- [5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns*, volume 1. Wiley, Great Britain, 1996.
- [6] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, 1997.
- [7] T. He and A. Kaufman. Virtual input devices for 3D systems. In *Proc. IEEE Visualization'93*, pages 142–148. IEEE, 1993.
- [8] IBM. Xena XML editor. <http://www.alphaworks.ibm.com/tech/xena>, visited February 20 2004.

- [9] Icon Information Systems GmbH. XMLSpy. <http://www.xmlspy.com>, visited February 20 2004.
- [10] ISO. Graphical kernel system (GKS). IS 7942, 1985.
- [11] Hirokazu Kato and Mark Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *Proc. IWAR 99*, San Francisco, USA, October 1999.
- [12] Hannes Kaufmann and Dieter Schmalstieg. Mathematics and geometry education with collaborative augmented reality. *Computer & Graphics*, 27(3):339–345, June 2003.
- [13] Joseph Newman, David Ingram, and Andy Hopper. Augmented reality in a wide area sentient environment. In *Proc. ISAR 2001*, pages 77–86, New York, New York, USA, October 29–30 2001. IEEE and ACM.
- [14] Gerhard Reitmayr and Dieter Schmalstieg. Mobile collaborative augmented reality. In *Proc. ISAR 2001*, pages 114–123, New York, New York, USA, October 29–30 2001. IEEE.
- [15] Gerhard Reitmayr and Dieter Schmalstieg. An open software architecture for virtual reality interaction. In *Proc. VRST 2001*, pages 47–54, Banff, Alberta, Canada, November 15–17 2001. ACM.
- [16] Gerhard Reitmayr and Dieter Schmalstieg. Location based applications for mobile augmented reality. In Robert Biddle and Bruce Thomas, editors, *Proc. AUIC 2003*, volume 25 (3) of *Australian Computer Science Communications*, pages 65 – 73, Adelaide, Australia, February 4 – 7 2003. ACS.
- [17] Dieter Schmalstieg, Anton Fuhrmann, Gerd Hesina, Zsolt Szalavari, L. Miguel Encarnacao, Michael Gervautz, and Werner Purgathofer. The Studierstube augmented reality project. *PRESENCE - Teleoperators and Virtual Environments*, 11(1), 2002.
- [18] Ralph Schönfelder, Gerhard Wolf, Michael Reeßing, Ronny Krüger, and Beat Brüderlin. A pragmatic approach to a VR/AR component integration framework for rapid system setup. In *Proceedings of the 1. Paderborner Workshop "Augmented und Virtual Reality in der Produktentstehung"*, pages 67–79, Paderborn, Germany, June 11–12 2002. Heinz Nixdorf Institut.
- [19] Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.
- [20] M Sokolewicz, H Wirth, K Böhm, and W John. Using the GIVEN++ toolkit for system development in MuSE. In *1st Eurographics Workshop on Virtual Environments*, Barcelona, Spain, September 1993.

- [21] P. Strauss and R. Carey. An object oriented 3D graphics toolkit. In *Proc. ACM SIGGRAPH'92*. ACM, 1992.
- [22] Russell M. Taylor II, Thomas C. Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T. Helser. VRPN: A device-independent, network-transparent VR peripheral system. In *Proc. VRST 2001*, pages 55–61, Banff, Alberta, Canada, November 15–17 2001. ACM.
- [23] The Apache Software Foundation. Xerces XML parser. <http://xml.apache.org/xerces-c/index.html>, visited February 20 2004.
- [24] Stephan Veigl, Andreas Kaltenbach, Florian Ledermann, Gerhard Reitmayr, and Dieter Schmalstieg. Two-handed direct interaction with ar-toolkit. In *Proc. ART'02*, Darmstadt, Germany, September 30 2002.
- [25] VRCO. Trackd. <http://www.vrco.com/products/trackd/trackd.html>, visited March 2nd 2004.
- [26] Martin Wagner, Asa MacWilliams, Martin Bauer, Gudrun Klinker, Joseph Newman nd Thomas Pintaric, and Dieter Schmalstieg. Fundamentals of ubiquitous tracking. In *Proc. PERVASIVE 2004*, Vienna, Austria, April 18–23 2004.
- [27] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Gavin Nicol, Jonathan Robie, Robert Sutor, and Chris Wilson. Document object model (DOM) level 1 specification (second edition). <http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>, January 19th 2004.