An Open Software Architecture for Virtual Reality Interaction

Gerhard Reitmayr
Vienna University of Technology
Favoritenstrae 9-11/188/2
A1040 Vienna, Austria
reitmayr@ims.tuwien.ac.at

Dieter Schmalstieg
Vienna University of Technology
Favoritenstrae 9-11/188/2
A1040 Vienna, Austria
schmalstieg@ims.tuwien.ac.at

ABSTRACT

This article describes OpenTracker, an open software architecture that provides a framework for the different tasks involved in tracking input devices and processing multi-modal input data in virtual environments and augmented reality application. The OpenTracker framework eases the development and maintenance of hardware setups in a more flexible manner than what is typically offered by virtual reality development packages. This goal is achieved by using an object-oriented design based on XML, taking full advantage of this new technology by allowing to use standard XML tools for development, configuration and documentation. The OpenTracker engine is based on a data flow concept for multi-modal events. A multi-threaded execution model takes care of tunable performance. Transparent network access allows easy development of decoupled simulation models. Finally, the application developer's interface features both a time-based and an event based model, that can be used simultaneously, to serve a large range of applications. OpenTracker is a first attempt towards a "write once, input anywhere" approach to virtual reality application development. To support these claims, integration into an existing augmented reality system is demonstrated. We also show how a prototype tracking equipment for mobile augmented reality can be assembled from consumer input devices with the aid of OpenTracker. Once development is sufficiently mature, it is planned to make Open-Tracker available to the public under an open source software license.

Keywords

Tracking, Mobile Augmented Reality, Virtual Reality, XML

1. INTRODUCTION

Tracking is an indispensable part of any Virtual Reality (VR) and Augmented Reality (AR) application. While the need for quality of tracking, in particular for high performance and fidelity, have led to a large body of past and current research, little attention is typically paid to software engineering aspects of tracking software. Some current systems have a modular approach that allows to substitute one type of tracking device for another. Typically, this is the approach taken by commercial VR products that offer 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. One notable exception is the MR toolkit [21] of the University of Alberta, which still serves as a starting point for many VR research projects despite its aged architecture and lack of active development. What is needed is a system that allows mixing and matching of different features, as well as simple creation and maintainance of possibly complex tracker configurations.

In this article, we describe a tracking software system called *OpenTracker* with the following characteristics:

- An object-oriented approach to an extensive set of sensor access, filtering, fusion, and state transformation operations
- Behavior specification by constructing graphs of tracking objects (similar in spirit to scene graphs or event cascades) from user defined tracker configuration files
- Distributed simulation by network transfer of events at any point in the graph structure
- Decoupled simulation by transparent multi-threading and networking
- A software engineering approach based on XML [4], which allows to use many generic tools such as [2, 11, 10] for development, documentation, integration and configuration

- An application independent library to be integrated into software projects to ease the burden of dealing with tracking equipment and data

Through its scripting capability (tracker configuration files) as well as easy integration of new tracking features, *OpenTracker* encourages exploratory construction of complex tracking setups. It is equally useful for end users, which can fully exploit their hardware without any custom programming, as well as developers, who can easily build test environments. The modular approach gives instant access to wide range of tracking related functionality for any application. Through the release under the LGPL Open Source license [7], *OpenTracker* is available to a larger audience.

2. RELATED WORK

Ideas implemented in *OpenTracker* were drawn from several areas:

Device abstraction is a standard requirement for 2D graphical user interfaces, (e. g. GKS [12]), and sometimes incorporated into 3D applications [9]. There is a number of libraries such as VRPN [15], MRToolkit [21] implementing device abstraction for input devices typically found in VR and AR systems. Their main goal is to provide a fixed interface to the application for different devices and provide simple services for relaying the data over the network between several hosts. However, these libraries mostly lack any further means to process the data. Device abstraction is also an important goal of *OpenTracker*. However, it goes beyond pure abstraction using a static interface in that the data can be re-combined in novel ways.

Many interactive systems employ sophisticated event handling schemes. State changes to attributes of scene objects are either propagated through functional dependencies (e. g. routes in VRML [5], engines in Open Inventor [22]), or may be handled by user supplied callback functions (e. g. script nodes in VRML [5]). These approaches inspire the architecture of *OpenTracker*, although none of them deals specifically with tracker configurations.

Finally, an important requirement for virtual environments is support for distributed simulation, partly to support simultaneous users, partly to better exploit available hardware. Decoupled simulation was first introduced in MR [21], and later used in almost any major VR software system. Decoupled simulation can either be implemented by multithreading and/or symmetric multiprocessing on one host, or by configuring a small set of hosts to work as an ensemble. The latter approach may be inferior performance-wise because of network lag, but it is inexpensive and flexible, and thus favored by many researchers - for example, Rekimoto's "hyperdragging" system [19] uses a distributed architecture very much like our own.

3. DATA FLOW OF TRACKING DATA

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 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. To describe the details of this concept, we will need some theoretical definitions which are discussed in section 3.1. Details of an actual implementation are described in section 3.2.

3.1 Data Flow Concept

Each transformation 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 the following concepts.

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 it's 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. This allows more complex computational structures.

Additionally, an input port can be connected to several output ports. This enables several children nodes connected to the same input port of a node. Upon receiving an event, the parent node can only distinguish between the input ports, not between the actual children.

Conversely, an output port can also be connected to other nodes by using references within the graph. This establishes new edges between a nodes output port and other nodes input ports. However this is transparent to the child node. It cannot selectively send events to only one parent, but all events are distributed equally to all parents.

Edge types

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 vector of tracker measurements or that require or compute the tracker state at an arbitrary point in time require different types of input or output interfaces. Examples are smoothing algorithms that take a history of events into account, or prediction algorithms that compute an expected measurement for a given point in time.

Therefore, 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 then the type of the edge. There are three edge types: *event*, which is implemented by event passing, *event queue* and *time dependent*. The latter two are implemented as interfaces that are polled by the parent node, because the data returned is parameterized. In the case of the *event queue* interface, it is possible to query the number of stored events and retrieve them by index. The *time dependent* interface can be queried by specifying a point in time, for which the appropriate data value is returned.

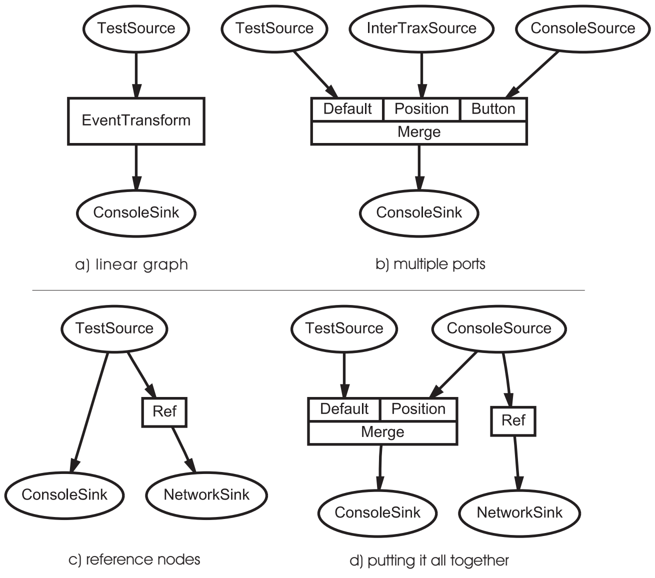

Figure 1: Visualizations of a data flow graphs as used in *OpenTracker*

Figure 1 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 Implementation Speci£c Details

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 their data values received from other nodes to external outputs.

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 [13]. A third type of source node emulates a tracker via the keyboard, access network data (see section 5) or simply responds with constant values (useful for development and debugging).

Some source nodes have a multi-threaded execution model to implement an efficient decoupled simulation model [21] (e. g., when blocking I/O must be used).

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.
- 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.
- Permutation filters are necessary to match data representations from different hardware or software platforms, such as equivalent, but incompatible quaternion representations.
- Merge filters assemble new data values using different parts of the data values of several children. Sample uses 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.
- Store-and-forward filters are useful if transient loss of tracking can be expected, for example if occlusion occurs in optical tracking. The last measured value is simply repeated to provide at least a reasonable and valid state.
- Confidence filters select data values from different children based on some measure of confidence in the accuracy of the data.

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.

3.3 Time

Time is reflected in several ways in the architecture of *OpenTracker*. The type system for edges supplies us with different ways of dealing with Time, either having an event based approach, with or without queuing of events, or by specifying functions of tracking data as continuous functions of time.

For the event based nodes, each event is time stamped by the individual device driver or node that generated it. Thus nodes can react on the temporal aspects of tracking data. For example, a simple prediction node incorporates the time difference between single events to correctly update its output.

More complex aspects such as a prediction for a changing prediction interval is satisfied by the different edge types. An application that wants to get a calculated value for an arbitrary point in time can query the state at that time from a node supporting *time dependent* output. How this value is calculated depends on the node's implementation.

OpenTracker does not implement any clock synchronization of different hosts working together in a network. There are already well established means to solve this problem such as the NNTP protocol.

4. SOFTWARE ARCHITECTURE OF THE LIBRARY

The intent of *OpenTracker* is to provide an auxiliary library that is to be integrated into VR or AR applications. Therefore it is kept very 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 module classes create and manage the nodes representing the functionality of the module. In the main loop of the library each module is called to provide new events and after an event is processed to handle results of the data flow. For example, the implementation of a network sink node stores any event data that it received during event propagation. Afterwards the network module checks each network sink node for updated data values, constructs a new network packet and sends it to the configured destination. Modules may be implemented multi-threaded to avoid stalling the main thread during longer computations or polling a device with blocking I/O.

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 to maximize flexibility. Application programmers have to either use on of the supplied nodes (such as a generic call back node) or 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 applications main loop to avoid additional threads and synchronize the tracking data processing more closely with the application. These design decisions ensures that the library can adapted to the special needs of every application.

The primary type of event used in the current implementation is tailored toward tracking applications. It encodes a position in space, an orientation, button states, a time stamp and a confidence value to describe the quality of the data. 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.

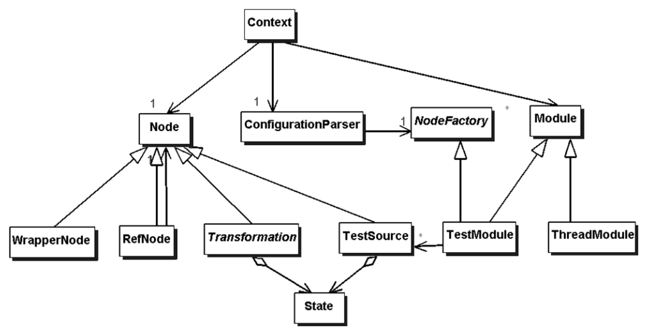

Figure 2: Architecture of the *OpenTracker* library

Figure 2 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.

5. DISTRIBUTED TRACKING

There are several reasons why is is desirable to share tracker data over a network:

- Using the tracker data at multiple host computers for a distributed virtual environment (local or remote): Input in the form of tracker data becomes readily available through transparent network access via *OpenTracker*. The scene database still has be to kept consistent through a proprietary application protocol, but the task is much simplified.

- With the same approach, 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.

- 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 (e.g., an SGI O2 has fast video hardware but a slow CPU, whereas for a PC the opposite may be true).

OpenTracker allows multiple senders and receivers of tracker data to communicate asynchronously through the use of IP multicasting (Figure 3). This approach effectively implements decoupled simulation in a distributed over several hosts, since each of the senders and receivers can operate independently. 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.

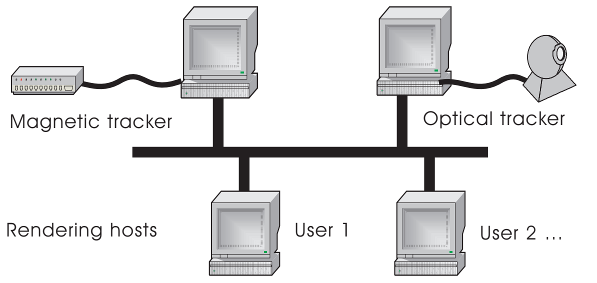

Figure 3: Distributing tracking data send to different rendering hosts

While there is a preferred network protocol for *Open-Tracker*, support for additional formats can be easily implemented. In the following, we give some examples as to how a networked setup can be used:

- A tracker server (typically a cheap PC with lots of serial I/O boards running Linux) samples an Ascension Flock of Birds at highest rate and sends the resulting data stream via multicast to several clients using this data to animate a collaborative virtual environment.

- The Polhemus Ultratrak uses a proprietary network format and IP unicast packages. Unfortunately, its closed architecture does not support input devices with buttons such as a stylus or 3D-mouse. Therefore, we added a tracker object to the client that is able to decode the Ultratrak protocol. A button source reads button values from a standard parallel interface, and a merge filter combines these two sources to emulate a complete VR input device.

- A combination of vision tracking and magnetic tracking – see section 7 for details.

6. SOFTWARE ENGINEERING WITH XML

XML, the eXtensible Markup Language, is the emerging standard primarily aimed at web-based applications and software systems [4]. XML 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 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 [2] enforces content format on the tracker configuration file while building the corresponding structure in memory, thus automatically performing many of the consistency checks that have otherwise to be hand-coded.

- The same parser implements an API to manipulate the data structure at runtime and still keep it consistent with the DTD. Such a runtime structure can easily be written out to a valid configuration file again.

- A convenient XML editor such as [11, 10] with a graphical user interface allows the end user to design the tracker configuration without having to master the syntax. It also enforces the correct content format, reducing syntax and semantic errors made by users.

- Integration with high-level software engineering tools that create code or configuration files from specifications is simplified by the use of XML. Even automatic reverse engineering of complex configurations is easier relying on a defined structure than from pure source code.

- Using the extendible style language (XSL) [1, 6], automatic textual and even graphical documentation can be created from a tracker configuration file, for example by using the free graph drawing utility *dot* [3] (see Figure 1).

Markup languages are generally used to annotate textual documents with structural information. Thus a general XML document consists of text grouped and structured with tags. Markup languages defined in XML consist of elements, essentially expressed as tags, and a structural model (the content model) of the possible ways these elements may be nested. Moreover, elements are annotated by name-value pairs called attributes.

OpenTracker maps elements to nodes and attributes to members of these nodes. We are not using any textual content but purely rely on the content model provided by the DTD. An open source XML parser [2] builds a tree of elements representing the given configuration file. *OpenTracker* walks the tree and creates a new node for each element based on the elements name. The string values of the attributes are parsed according to the objects class and the corresponding members are set. Attributes typically describe such data as the parameters of a transformation. The parent - child relationship of the data flow graph is directly mapped onto the parent - child relationship of XML elements.

The content model enforces interface and semantic constraints on the specified graph. As described in section 3.1 edges and the corresponding node ports are typed and therefore restrict the possible combinations in the construction of the graph. These constraints are expressed in the DTD and are checked by an XML parser or enforced by an XML editor. Also restrictions on the number of children are described in the DTD. *Source nodes* typically do not have any children as they rely on data from external sources to compute their own data. A number of *filter nodes* get the value of a single child node, transform it and pass it on. 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. Again XML enforces the uniqueness of these IDs and the parser library simplifies the search for the referenced elements.

While children of nodes with only one input port are directly mapped to children elements in the XML file, children of different input ports need to be addressed differently. This is handled using wrapper elements. Any group of children that is connected to a specific input port is wrapped by an additional XML element. This element in turn is the direct child of the node of interest. These elements are closely re-

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
    <configuration>
        <ARToolKitConfig camera-parameter="camera_para.dat"/>
    </configuration>
    <ConsoleSink comment="Pip">
        <StbSink station="0">
            <EventTransform DEF="Camera" scale="0.001 0.001 0.001">
                <ARToolKitSource tag-file="pip.tag" />
            </EventTransform>
        </StbSink>
    </ConsoleSink>
    <ConsoleSink comment="Pen">
        <StbSink station="1">
            <EventDynamicTransform>
                <TransformBase>
                    <Ref USE="Camera"/>
                </TransformBase>
                <EventTransform scale="-2.1 -2 0" translation="0.14 0.1 -0.01">
                    <WacomGraphireSource device="1"/>
                </EventTransform>
            </EventDynamicTransform>
        </StbSink>
    </ConsoleSink>
    <ConsoleSink comment="Viewpoint">
        <StbSink station="2">
            <EventTransform rotation="1 0 0 0">
                <TestSource frequency="25"/>
            </EventTransform>
        </StbSink>
    </ConsoleSink>
</OpenTracker>
```

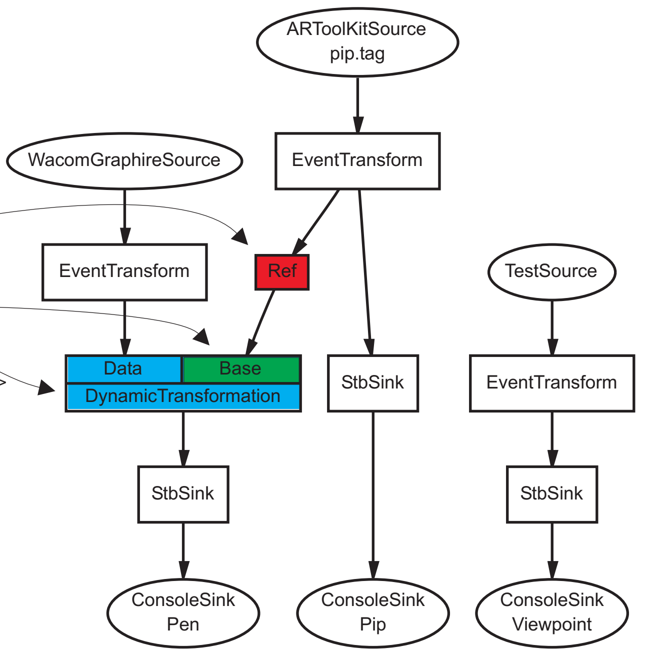

Figure 4: Example configuration file and resulting graph. (This figure is reproduced in color on page 000.)

lated to the node's element and are typically the only possible children elements. They 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.

7. APPLICATIONS

We have successfully used *OpenTracker* in a number of experimental setups either using it as our sole source of tracking data or integrating it with an existing setup.

For example, in an experimental pen-and-pad interface, we combined a vision tracking approach (ARToolkit) for the pad with a magnetic tracker (Ascension Flock of Birds) for the pen. Two separate servers for video and magnetic tracking were sending their measurements over the network to a rendering host, where the combined data was picked up by an *OpenTracker* component (Figure 3). The tracking data from this source was transformed to register with the tracked objects and fed into *Studierstube*, an augmented reality environment recently described in [20]. The rendering was then overlayed onto the video input from the camera. Figure 5 shows resulting image.

Another setup combines simple consumer devices to give a similar interface. Here a Wacom graphic tabled is tracked with ARToolkit and used as the pad, whereas the pen's position is measured only by the tablet. Thus the pen can only be used on the pad. This is enough to manipulate 2D user interfaces displayed on the pad (Figure 6). Note that the pen's position in space is derived by combining the pen's location on the tablet and the tablet's position and orientation in space. Again this behavior was scripted with a simple configuration file (shown as an example in Figure 4). Also the library was integrated directly with *Studierstube*, instantly allowing access to the used devices.

Finally, *OpenTracker* was used extensively to integrate different input devices in a mobile AR setup [18]. The sys-

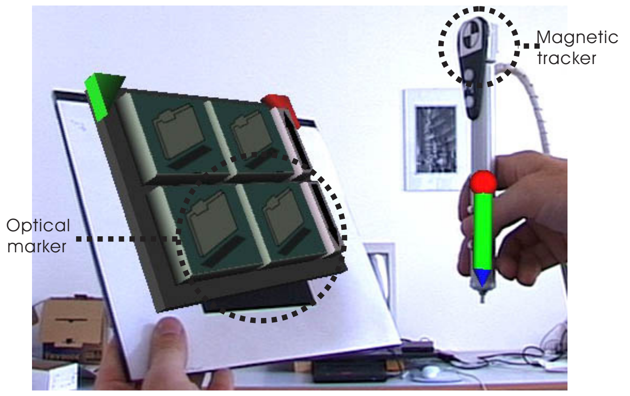

Figure 5: Different tracking devices integrated transparently. (This figure is reproduced in color on page 000.)

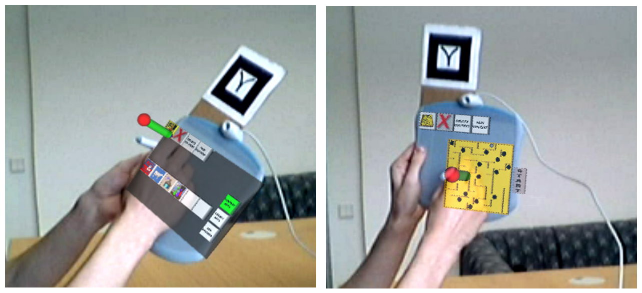

Figure 6: A graphics tablet and a pen yield a simple pen-and-pad interface. (This figure is reproduced in color on page 000.)

tem uses an InterSense InterTrax^2 orientation sensor and a web camera for fiducial tracking of interaction props mounted on a helmet worn by the user. 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. Similar to the last setup, 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. *OpenTracker* deals with all the complex transformations between the coordinate systems of the input devices and passes data in a world stabilized reference system to the AR application for rendering. Figure 7 shows a user overlayed with the AR environment he is working in.

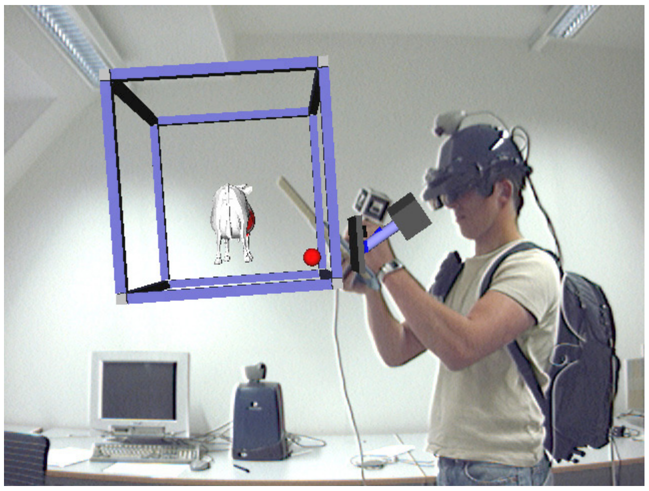

Figure 7: A user interacting with an AR application using a mobile setup. (This figure is reproduced in color on page 000.)

Besides accommodating a wide range of hardware configurations, 3D interaction techniques can be implemented in *OpenTracker* and be used transparently to the application. For example, Head-directed navigation [8] can be added by implementing a node that computes position changes from incoming orientation data. Similarly interaction techniques such as amplifying rotations [16, 17] can be achieved. Once such nodes are realized, these interaction techniques are immediately available to existing setups.

Integration with software engineering tools and approaches such as described in [23], [14] should be possible. Employing XML to describe the configurations allows integration with other tools because the standard software needed to process XML files is available. Simple generating and parsing of XML allows the generation of configurations from specifications or reverse-engineering of existing configurations into these tools. This can results in a full round trip approach to engineering virtual environments.

8. CONCLUSIONS AND FUTURE WORK

None of the important properties of *OpenTracker* – such as filtering, decoupled simulation, or configuration languages – are genuinely new. Yet we were surprised in being unable to find a publicly available solution truly suited for the needs of a virtual reality developer – a lack which led to the conception of *OpenTracker*. While to capabilities of *OpenTracker* are utterly unspectacular, we found them much needed.

The described applications show the versatility of our approach. Not only device abstraction is achieved, but complex dependencies and interactions between the devices can be described and configured. This happens transparently to the application and simplifies the developers task.

Much remains to be done. Although the current version is stable and integrated in our standard applications, it is far from complete. The set of supported device drivers should be exhaustive and up to date. There are also several other interesting extensions, such as generic event type to include other types of data or reconfiguration of the tracker graph at runtime. We thus invite contributors all over the world to help improving this open source project.

For more information, check out the project home page: http://www.studierstube.org/opentracker/

Acknowledgments

This project was sponsored by the Austrian Science Fund *FWF* under contract no. P-14470-INF. Special thanks to Anton Fuhrmann for his perpetual will to comment, to Ivan Viola and Matej Mlejnek for their contribution to the implementation and to Mark Billinghurst and Hirokazu Kato for ARToolkit.

APPENDIX

A. REFERENCES

[1] S. Adler et al. Extensible stylesheet language (XSL) 1.0. http://www.w3.org/TR/xsl/.

[2] Apache. Xerces XML parser. http://xml.apache.org/xerces-c/index.html.

[3] AT&T. Graphviz. http://www.research.att.com/sw/tools/graphviz/.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, et al. Extensible markup language (XML) 1.0. http://www.w3.org/TR/REC-xml/.

[5] R. Carey and G. Bell. *The Annotated VRML 2.0 Reference Manual*. Addison-Wesley, 1997.

[6] J. Clark. XSL transformations (XSLT) version 1.0. http://www.w3.org/TR/xslt, 1999.

[7] F. S. Foundation. Lesser GNU Public License. http://www.gnu.org/copyleft/lesser.html, February 1999.

[8] A. Fuhrmann, D. Schmalstieg, and M. Gervautz. Strolling through cyberspace with your hands in your pockets: Head directed navigation in virtual environments. In *Proc. of the 4th EUROGRAPHICS Workshop on Virtual Environments*, pages 216–227. Springer-Verlag, June 1998.

[9] T. He and A. Kaufman. Virtual input devices for 3D systems. In *Proc. IEEE Visualization'93*, pages 142–148. IEEE, 1993.

[10] IBM. Xeena XML editor. http://www.alphaworks.ibm.com/tech/xeena.

[11] Icon Information Systems GmbH. XMLSpy. http://www.xmlspy.com.

[12] ISO. Graphical kernel system (GKS). IS 7942, 1985.

[13] H. Kato and M. Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *Proc. (IWAR'99)*, San Francisco, CA, USA, October 1999. IEEE.

[14] G. J. Kim, K. C. Kang, H. Kim, and J. Lee. Software engineering of virtual worlds. In *Proc. VRST'99*, 1999.

[15] U. of North Carolina at Chapel Hill. VRPN - virtual reality peripheral network. http://www.cs.unc.edu/Research/vrpn/.

[16] I. Poupyrev, T. Otsuka, S. Weghorst, and T. Ichikawa. Amplifying rotations in 3D interfaces. In *Proc. ACM CHI'99*, pages 256–257, 1999.

[17] I. Poupyrev, S. Weghorst, and S. Fels. Non-isomorphic 3D rotational techniques. In *Proc. ACM CHI'2000*, pages 546–547, 2000.

[18] G. Reitmayr and D. Schmalstieg. Mobile collaborative augmented reality. In *Proc. ISAR 2001*, New York, USA, October 29–30 2001.

[19] J. Rekimoto and M. Saitoh. Augmented surfaces: A spatially continuous workspace for hybrid computing. In *Proc. CHI'99*. ACM, 1999.

[20] D. Schmalstieg, A. Fuhrmann, and G. Hesina. Bridging multiple user interface dimensions with augmented reality. In *Proc. ISAR 2000*, pages 20–29, Munich, Germany, October 5–6 2000. IEEE and ACM.

[21] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.

[22] P. Strauss and R. Carey. An object oriented 3D graphics toolkit. In *Proc, ACM SIGGRAPH'92*. ACM, 1992.

[23] J. S. Willans and M. D. Harrison. A 'plug and play' approach to testing virtual environment interaction techniques. In *Proc. EGVE 2000*, June 2000.