Towards a Universal Implementation of 3D User Interaction Techniques

Mathis Csisinko* Hannes Kaufmann†

Institute of Software Technology and Interactive Systems
Vienna University of Technology

Abstract

This paper presents a versatile - write once, use everywhere - approach of standardizing the development of three-dimensional user interaction techniques. In order to achieve a platform and application independent implementation of 3D interaction techniques (ITs), we propose to implement the related techniques directly in the tracking middleware. Therefore a widely used tracking framework was extended by a Python binding to allow straight forward scripting of ITs. We cluster existing 3D ITs, into those which can be fully, partly or not implemented in the tracking middleware. A number of examples demonstrate how various interaction techniques can quickly and efficiently be implemented in the middleware and are therefore fully independent of the underlying application. We hint at how this approach can be used to decouple menu system control from the application with the final goal to help establishing standards for 3D interaction.

Keywords: 3D interaction techniques, 3D user interfaces, tracking middleware, OpenTracker Python binding.

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities; H.5.2 [Information Interfaces and Presentation]: User interfaces—Interaction styles; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques

1 Introduction and Motivation

Providing taxonomies and classifications of user interaction techniques (ITs) [3, 5] is important to establish a set of well known, common ITs within the research community. Whereas a number of standard ITs can be considered common knowledge by now, platform independent or even application independent "standard" implementations of these do not exist. Usually application developers hand-code ITs for their applications themselves. It therefore causes additional programming effort for the developer to experiment with different ITs, which could enhance quality and helps to improve usability of an application though. Decoupling ITs from the application, handling them on a different level can lead to establish application independent, reusable, standard implementations of ITs.

In virtual environments tracking is an indispensable part to acquire data of input devices. Over the past few years, several attempts were made to create a new generation of tracking frameworks, paying more attention to flexibility, modularity and other software engineering aspects. On one hand the aim of these toolkits is to support a high number of various tracking devices. On the other hand they allow extensibility, customization and provide (not so common) features like data filtering and sensor fusion. Combining both aspects in a framework in a highly configurable manner leads to very flexible tracking middleware.

*e-mail:csisinko@ims.tuwien.ac.at
†e-mail:kaufmann@ims.tuwien.ac.at

The main requirements for tracking middleware are that complex tracker configurations should be supported by mixing provided features. In addition, it should still be possible to create and maintain such tracker configurations in a simple and convenient way. Tracking data preprocessing in terms of filtering and merging is ideally supported by the framework to create rather complex tracking behaviour. Easy authoring of the tracking configuration or even dynamic reconfigurability for mobile or ubiquitous computing applications is also desirable.

Currently user interaction (UI) techniques in virtual environments are usually implemented on the application level. Given the possibility to create complex tracking behaviour directly on the tracking middle(ware) level, the idea to implement 3D user interface techniques on that higher level becomes quite obvious. Since most application developers are using tracking middleware in order to get hardware support for various interaction devices (without having to reinvent the wheel), this middleware might serve as a common ground for future UI standards. The advantages of such an approach are versatile:

- 3D UI techniques can be specified in a reusable way and independently of the application. By using a scripting language for authoring ITs, rapid prototyping is supported as well. This proves to be very powerful since reconfiguration, adaptation and exchange of related 3D UI techniques becomes very easy.
- ITs do not have to reside in a proprietary way on the application level and do not have to be reimplemented for each application any more. Even better, a repository containing a collection of ITs can be established. It is to be contacted at runtime in order to download, set up and execute script code without program termination of the tracking system.
- Furthermore, dynamic IT adaptation depending on hardware setup and runtime reconfiguration becomes achievable, since all relevant data is accessible from within the middleware. For example, if a user changes his location in a system with heterogenous tracking setups, ITs can be dynamically adapted according to available hardware at a new location.
- The work load of the application process can be effectively decreased if it does not have to process and interpret huge amounts of raw tracking data.
- Data handling can be more efficient, when tracking middleware is able to set up the data according to the application's needs. Considering distributed MR systems specifically, network traffic can be reduced if there is no need to send large amounts of raw tracking data over the network.
- 3D UI techniques can be handled directly on the tracking server. In distributed environments interpretation of the data is done once for all clients, avoiding inconsistencies in case a client does not receive all raw data and therefore interprets the IT differently (e.g. gesture interpretation).
- Standard or reference implementations of known ITs could be shared by the research community.

After summarizing related work in the area of existing tracking frameworks (section 2), which our work is based on, we describe the design and implementation (in sections 3 and 4) of our approach. In section 5 results are presented, giving a number of examples and sample implementations. To generalize our approach we cluster existing 3D ITs (based on Bowman's taxonomy [5]) into three types of middleware support in section 6. This section goes into detail on ITs that can be fully realized in tracking middleware, ITs where specific helper functions can be implemented in the middleware to support the application, and finally ITs which are extremely application dependent and can only be realized on the application level. In the remaining part (sections 7 and 8) we summarize our work and describe future work, for example, on how tracking middleware can be exploited to standardize menu system interaction and widget control in virtual environments.

2 Related Work

A number of 3D interaction techniques have been developed by various researchers during the past decade. A comprehensive and recent overview is given by Bowman in [5]. We based our clustering of techniques (section 6) on his taxonomy. Tasks are categorized into *selection*, *manipulation*, *navigation*, *system control*, *symbolic input*, and into more complex *tasks* such as modeling.

Various attempts to standardize 3D user interaction can be found in literature from previous years.

InTML [8] is an interesting approach to describe interaction techniques in a standardized way. In its aim to be high-level, reusable and extensible, it is similar to our work. InTML is not supposed to be a standalone toolkit. Instead, it relies on other VR toolkit implementations. In InTML 3D UI techniques are described in an XML based markup language. In order to get executable code, this description has to be translated, interpreted or compiled. Currently, work is spent on an InTML implementation on top of Java3D and VRJuggler.

A uniform approach for specifying mixed reality interfaces, including 3D UI techniques was recently published by Figueroa et al. [7]. In this ambitious attempt to describe components of 3D user interfaces, a formal specification model and a corresponding XML based description language including pseudo code were introduced. This Interface Component Description Language (ICDL) seems not to be suitable for direct implementation. The current focus of development on ICDL appears rather to be on theoretical specification than on practical implementation.

As described in the previous section, tracking middleware toolkits are specialized in handling tracking data of various devices on an abstract level. Combining input of several devices, performing preprocessing and filtering are interesting features in these frameworks, which shall be referred to in the following.

One of the early software toolkits dedicated to developing interactive and immersive graphics applications is MR Toolkit [19]. It provides device abstraction and network transparency for tracking devices. Applications are decoupled from the actual tracking devices and programmers can substitute real devices with virtual ones for debugging and testing purposes. Active development is discontinued but it showed the way forward in this field of research.

VRPN (Virtual-Reality Private Network) [22] is one of the most well known and popular device-independent and network-transparent frameworks for peripheral devices used in MR systems. It supports a wide variety of input devices and different types of data such as 6DOF pose data, button states, analogue values, incremental rotations and more. A device can offer interfaces for several types, and devices can be layered by connecting device outputs to inputs of other devices. VRPN runs on all common platforms and is used in research and industrial projects.

OpenTracker [17] is an open software architecture that provides another framework for the different tasks involving tracking input devices in MR applications. The OpenTracker framework eases the development and maintenance of hardware setups in a more flexible manner than what is typically offered by VR development packages. This goal is achieved by using an object-oriented design based on XML, taking full advantage of this technology by allowing to use standard XML tools for development, configuration and documentation. A multi-threaded execution model takes care of tunable performance. Filters and transformations can be applied to tracking data. XML based configuration files are used to describe tracking configurations that usually consist of multiple input devices. Transparent network access allows easy development of decoupled simulation models.

We use OpenTracker and extended it by a Python binding to allow scripting as it will be elaborated in detail in sections 3 and 4. In the following we compare our approach with existing work.

The previously mentioned dependence of InTML [8] on other toolkits is one of the main differences to our approach. The same deficiency seems to apply to the ICDL [7] language: no direct implementation appears to exist without the necessity to translate the specification. OpenTracker is already widely used and not depending on other third-party toolkits.

Another interesting feature not found in InTML is the possibility to embed interpretable program code in terms of an easy-to-learn scripting language. As a direct effect of the expressiveness of programming languages, this enhancement also results in a massive increase of expressiveness in describing 3D UI techniques.

Our approach is not limited to OpenTracker only since interfacing with VRPN [22] is possible and easy: OpenTracker contains a built-in module to obtain VRPN data from network and is also capable of directly transmitting tracking data in native VRPN format. With the lack of tracking data preprocessing features in VRPN, but with its support for a rich set of devices, combining OpenTracker with VRPN proves to be very powerful. The example in section 5.2 demonstrates the use of VRPN input.

In contrast to related work the contribution of our paper is manifold. We introduce scripting of interaction techniques in tracking middleware. This tight integration of user interaction techniques and tracking provides increased flexibility as described in section 1 and offers new possibilities (see summary in sections 7 and 8). Decoupling ITs from the application is a methodical general approach which helps to reduce application interface code, enhances performance and eases (distributed) MR system development. Scripting ITs once on the middleware allows platform independent usage on multiple systems.

3 Design

Considering the features, OpenTracker is an appropriate framework to integrate 3D UI techniques directly in middleware. We give a brief overview of relevant functionality.

3.1 Data Flow Concept

A main concept of OpenTracker is to build up a data flow network in a modular way, which consists of several steps of data acquisition and manipulation. Breaking up complex behaviour in a number of basic operations results in a data flow graph. Nodes in this directional acyclic graph represent the entities, where tracking data processing occurs, while tracking data is communicated unidirectionally over the interconnecting edges between the nodes. Data is inserted into this graph by special source nodes, processed by intermediate nodes and forwarded to external outputs by sink nodes.

Figures 2, 4 and 6 are example illustrations of data flow graphs: In these representations, data is propagated unidirectionally from top (sources) to bottom (sinks).

From the perspective of a node, data is exchanged by ports, namely by input and output ports. To allow nonlinear graphs, each

node can have multiple input ports and the data generated at its single output port can act as input for several other nodes. This multiple input property is desirable in order to perform more complex computations, when data from different origins is involved. On the other hand, supplying multiple nodes with the same output data of a single node offers a transparent mechanism of data reusability.

There exist three different edge types: The most commonly used is the event edge type (implementing the event generator interface), where data is pushed from the origin successively through the graph. In contrast to this push-driven mechanism, event queue and time dependent edge types are pull-driven: A history of previous events is kept and a polling mechanism has to be established. This is usually more suitable for performing processing on data at different instants. With event queueing, events ordered by time can be retrieved by index, while the time dependent interface allows to directly query by instant.

3.2 Multi-modality of Events

Within OpenTracker, the contents of events are not restricted to predefined data records. Instead, each event can consist of multiple, dynamically created attributes of certain data types.

Apart from predefined generic data types, custom type registration is also supported. Exploiting this possibility allows to pass further configuration data (not necessarily tracking data) between nodes, which can be very helpful in the attempt to implement more complex behaviour directly in tracking middleware. Intercommunication between several nodes in order to pass configuration data becomes feasible.

4 IMPLEMENTATION

OpenTracker is an open source, extensible tracking framework that enables developers to add support for new input devices; providing additional methods for filtering or preprocessing by implementing new nodes is possible. Consequently, the same applies when trying to integrate a 3D UI technique directly in OpenTracker.

4.1 Monolithic Approach

At first, one might be tempted to implement a single simple node, which meets the requirements of a particular IT and fits the user's needs. Although this might be suitable for some purposes, the method has its deficiencies since it is rather proprietary. The functionality of the interaction technique is completely hidden in code and not disclosed in the tracking configuration. With a set of special nodes, each of them tailored to implement a particular 3D UI technique, one can not take advantage of similarities of related techniques. Even if reusing some common code parts by inheritance is possible to some extent, this is not reflected in the configuration file.

In addition, this is not very suitable for rapid prototyping: Each time a new 3D UI technique is to be implemented, it is mandatory to rebuild the whole framework. A more universal solution is desired.

4.2 Modular Approach

In a second attempt, some basic and multi-purpose operations have to be identified. In order to implement a particular 3D UI technique, an appropriately chosen selection from this set of basic operations has to be combined properly to achieve a more complex and meaningful behaviour. This building block style construction process allows synthesizing new 3D UI techniques without code altering and makes functionality more obvious, as it is not completely hidden in code. Such a modular attempt can also be seen as a discrete way, opposed to the integrated way described before.

It is positive that quite a lot of these general multi-purpose operation nodes are already pre-existing in OpenTracker and are reusable in this new context. For example, coordinate transformation nodes in many different flavours were implemented in the past and can be reused to perform calculations on point and orientation data. Merging multiple inputs in several ways as well as various gate nodes were used for other purposes before.

As it turns out, one disadvantage of this modular attempt can be seen in the high complexity of the resulting tracker configuration and - compared to programming languages - the reduced expressiveness of its elements. Figure 2, as referred to in section 5 illustrates this complexity issue for a rather simple task.

4.3 Scripted Approach

This leads us to a third attempt, which tries to overcome the deficiencies of both mentioned methods in 4.1 and 4.2. Embedding a scripting language radically increases flexibility, ensures high expressiveness and offers the possibility for rapid prototyping. No framework rebuilding is required and the 3D UI technique program code is closely coupled to its tracking configuration.

We selected Python [18] as a quite expressive scripting language. It supports object-oriented programming and is easy to learn without much effort. Even though it is an interpreted language, code execution is speeded up by using a byte code representation of script code. Utilizing a powerful wrapping tool, namely SWIG (Simplified Wrapper and Interface Generator) [1, 6], the creation of some kind of "glue code" to access OpenTracker objects from within the domain of Python was straightforward. With this automatically generated wrapper code, Python code can deal with OpenTracker objects in a native and similar way as in C++. Even parameter passing and sharing objects at the interface between Python and OpenTracker is possible to achieve strong data coupling between both languages.

Python script code is called in the event processing mechanism of OpenTracker. For each script node, execution is carried out in methods of a Python class, whose name can be freely chosen. Due to the fact that to the script programmer the same set of interfaces as in C++ is offered, it nicely integrates into the concept of OpenTracker. Also, the Python class exactly resembles C++ class implementations and argument passing is done in the same natural way as in C++, but without the necessity to recompile the whole framework. Apart from rapid prototyping (testing the effects of new feature implementation candidates before actually extending the code base), this also offers the possibility to get familiar with OpenTracker in an easy way.

Going more into detail concerning the data processing mechanism in event generation, events are usually inserted into the data flow graph actively by sources, represented by child nodes of the tracking tree. Subsequently, each child node pushes events further down the tracking tree by notifying its parents. Events do not necessarily have to be processed in this strict manner of being inserted by sources and being removed by sinks though. Event generation and filtering in intermediate nodes is possible as well. In order to accomplish that, the event generator interface in the node class offers a method to notify and update all immediate parent nodes of an event. They get notified whether the event was newly generated, processed or manipulated by any child of this particular node. As a consequence of this notification mechanism, a certain method is called, whenever an event is received from any child. In that call the actual event and information about its origin are passed as arguments to the method.

In tracking trees of depth greater than 2 this mentioned notification method is usually called recursively. Assuming that at least one script node is present somewhere in the tracking tree, we illustrate the integrated concept: The internal C++ part of the script node is notified about an event and passes it along with other parameters to the specified Python method. This Python method itself is free to perform any operation on this event (can even ignore the event), but usually pushes it in a modified way further down the tracking tree. This is achieved by calling the same notification mechanism

method as in C++ from within the domain of Python. Due to the wrapping code, the corresponding C++ notification method is executed. In respect to other tracking nodes, there is absolutely no difference, whether an event is created in native C++ code or Python was involved in any stage of its generation.

Even passing events from one Python script node to another, whether directly or indirectly by having intermediate C++ nodes in between, causes no problems: There is only one single Python interpreter present, which operates on the pregenerated byte code of the specified Python code module, which contains class definitions for each Python node. However, it should be noted that the call stack of the notification mechanism in event generation can contain several calls to Python in between of native C++ calls.

5 Examples and Results

In order to demonstrate some characteristics of the implementation techniques (as briefly discussed in section 4) two examples are elaborately presented in the following.

In a first simple example we will compare the modular and the scripted approach by using a mouse wheel to trigger a button event. In a second example the Go-Go interaction technique [16] is fully implemented involving scripted and modular features. In that example input tracking data received from a device via VRPN is used.

5.1 Mouse Wheel to Button Conversion

Figure 1 shows a tracked input device, a wireless pen, that we use in our multi-user optical tracking setup. Optical tracking makes use of the retroreflective marker body attached to the end of the wand. In terms of operating system device type classification it belongs to the group of mouse devices, but some of its mouse features are not directly usable in a 3D environment without conversion. The device appears to the tracking system as a passive prop, except for digital button information, which is transceived wirelessly via Bluetooth. In order to rejoin data coming from different tracking subsystems, native button input of this device is merged with position and orientation data supplied by the optical tracking system.

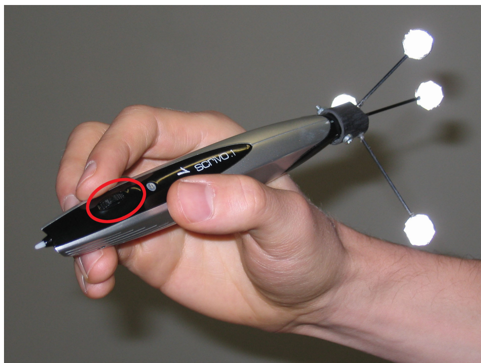

Figure 1: An optically tracked wand/pen. Within the red circle a scroll wheel is visible.

In some applications we use the mouse wheel as a button (by pressing down the wheel). Experience showed though, that some users do not press the wheel but rather tend to scroll the wheel, expecting the application to react to it as if it were a button press. Therefore it may be desirable for several applications to assign button functionality to the mouse wheel. Such a mapping proved to be flexible and very convenient for most users.

At first glance, one might expect that the task of converting data coming from the mouse wheel into button state information is rather simple. The mouse wheel is typically used for scrolling though. Rolling the wheel therefore results in incremental value changes over time which is different to digital button information. The information conversion has to be of differential nature: If the wheel is rolled over a specific time period, a pressed button state should be generated. Whenever the mouse wheel is standing still, a released button state is to be detected. Consequently, the crucial task is to identify the presence or absence of mouse wheel movement. Although this can be done in a monolithic way by creating a special node (as described in section 4.1), a modular implementation (as outlined in section 4.2) shall be depicted in the following.

5.1.1 Modular Approach

We briefly describe the "building blocks" of the data flow graph. Following the right path in figure 2, data input coming from the mouse input device is scaled by $(0,0,1)$ to eliminate all 2D mouse position data in the x and y component. The remaining z component contains mouse wheel information, which is passed to a timestamp generator node. This is due to the fact that the following node depends on getting events on a regular basis. The TimestampGenerator stores the last received event and resubmits it unmodified each time a given timeout is elapsed.

The next node in the chain computes the difference between attributes of two events separated by a given timeframe. The minuend in this operation is the actual received event, the subtrahend a previously recorded past event as described before. The output of this operation is feed to two range filter nodes. These nodes inspect the position attribute and perform event passing or blocking depending on conditions inspecting the length of the position vector. While the RangeFilter on the left blocks all events when a movement is detected, the behaviour of the other is strictly complementary.

The following simple state machine consists of just two nodes resembling each of the complementary button states (pressed and released) and is only present for convenience: An interface method or helper node (ActiveGate) can be used to retrieve the name or index of the currently active state. In this configuration events are passing the state machine at any instant without modification.

The EventUtility nodes in the next step synthesize button data events. As attribute creation is only happening if an event is pushed through one of these nodes, button state generation is according to the actual state.

The parent Merge operation inspects the events of its child nodes and generates its own events with only the button attribute set. With this semantics it is possible to remerge the data flow paths again: No data combination of child node events is needed and the originating paths of these child node events are not relevant any more. In contrast to this implicit path joining mechanism, the final button information is computed arithmetically in the last step. Plain button information coming from the mouse input device is combined with the result of this conversion in a bitwise OR operation.

5.1.2 Scripted Approach

Corresponding to what we described in section 4.3, the same functionality can also be expressed in a few lines of Python code (see figure 3). This specific Python class which is derived from a base class encapsulates everything needed in an object-oriented way. The instantiation operation method is similar to constructors in C++. It is called on construction of the Python node to allow initialization of any class instance data attributes. Although the attributes are comparable to member variables in C++, they are not declared and can be introduced later too. It can be done in C++ style though to conform to C++ programming patterns. In the case of performing custom initialization, it is essential to call the proper instantiation method of the base class. Otherwise, important initialization tasks for linking this Python class instance to its C++ counterpart will remain uncompleted.

Just like in C++, the method onEventGenerated is called, whenever an event is received from any child. To briefly explain that

```
class ConvertZToButtonNode(PythonNode):
  def __init__(self,pythonNode):
    PythonNode.__init__(self,pythonNode);
    self.eventQueue = [];
  def onEventGenerated(self,event,generator):
    self.eventQueue = filter(lambda prevEvent: prevEvent.time + 250 >= event.time,self.eventQueue) + [Event(event)];
    localEvent = Event();
    if event.getPosition()[2] - self.eventQueue[0].getPosition()[2] == 0:
      localEvent.setButton(event.getButton());
    else:
      localEvent.setButton(event.getButton() | 1);
    self.updateObservers(localEvent);
```

Figure 3: Python code for mouse wheel to button conversion

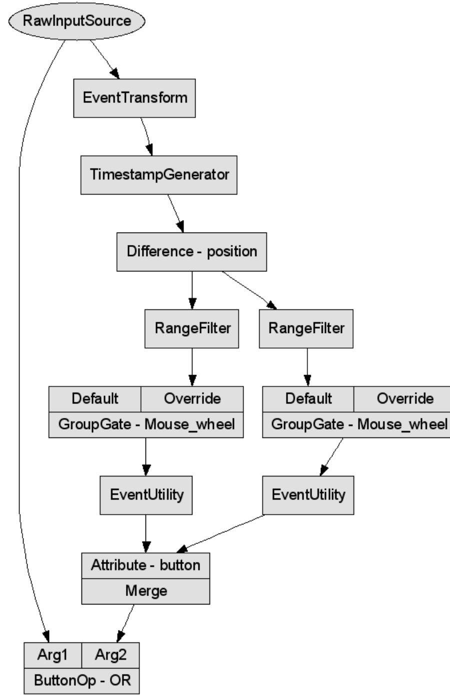

Figure 2: Data flow graph of the modular approach: Mouse wheel to button conversion

method: Data attribute eventQueue contains an ordered list of the last received events. The first line of code in this method deals with keeping the list up to date: The event received at this instant is appended at the end of the list and all past events not used any more in the front are eliminated. This single line statement demonstrates a powerful feature of the Python language, expressing rather complex operations in an elegant and brief way. Afterwards, the local event is generated. Depending on the detection of change in the z component (containing the mouse wheel movement) button state information is determined involving physical button input. Finally, the local event is propagated by calling updateObservers in the same manner as in C++.

It is notable that all operations performed on events involve calling wrapped C++ code. So, throughout the whole method, interfacing with C++ is used extensively.

The tracking configuration as illustrated in figure 4 is rather short this time: Again the TimestampGenerator node has to be present to allow button state fallback, when no movement is detected and device events might not occur until another action is performed with this device. But everything else was substituted by a single Python script node. Even the final bitwise button merging operation was taken care of in the Python node, which is configured by identifying the class to use with its name.

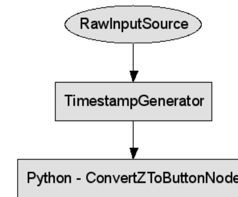

Figure 4: Data flow graph of the scripted approach: Mouse wheel to button conversion

5.2 Go-Go Interaction Technique

The Go-Go interaction technique [16] is an example of how to overcome the limited arm length of a user, who is immersed in virtual reality. The limitation of the user's limbs is directly related to the limited reach experienced by the user, when he is interacting in space but not moving (direct 3D interaction). An attempt to increase the interaction space is to scale the coordinate system depending on the distance between the user's head or torso and the hand or tracked interaction device. While retaining full tracking precision in the immediate surrounding space by leaving the coordinate system unchanged, points farther away are virtually pulled towards the user by scaling down the coordinate system. Alternatively, this transformation can also be seen as virtually extending the user's arm.

The conversion between the real and transformed coordinate system is defined by the following equation expressing the relation between the real (R_r) and virtually transformed (R_v) vector lengths of the user-device distance:

$$R_v = \begin{cases} R_r & R_r < D \\ R_r + k \cdot (R_r - D)^2 & R_r \geq D \end{cases}$$

D determines the radius of the sphere around the user separating near-by objects and those located too far away to be in the user's reach. k is a coefficient in the range $0 < k < 1$ specifying the scaling factor for the non-linear coordinate transformation part.

Supposing, that the origin of the coordinate system is already relative to the user's position, which can be achieved by other standard transformation nodes, the few lines of Python code in figure 5 perform the crucial part of the remaining non-linear transformation characterizing the Go-Go interaction technique.

This time the Python code operates just on the current event without the need for storing a history of previously handled events. Parameters d and k can be manipulated at runtime.

```
class GoGoInteractionTechniqueNode(PythonNode):
  def __init__(self,pythonNode):
    PythonNode.__init__(self,pythonNode);
    self.d = 0.5;
    self.k = 0.75;
  def onEventGenerated(self,event,generator):
    localEvent = Event(event);
    position = event.getPosition();
    distance = (position[0] ** 2 + position[1] ** 2 + position[2] ** 2) ** 0.5;
    if distance > self.d:
      factor = 1 + self.k / distance * (distance - self.d) ** 2;
    else:
      factor = 1;
    localEvent.setPosition((factor * position[0],factor * position[1],factor * position[2]));
    self.updateObservers(localEvent);
```

Figure 5: Python code for the Go-Go interaction technique

In this particular 3D UI technique, one must take care of the coordinate system origin (head or shoulder of the user) as mentioned before. This can easily be done by using standard OpenTracker nodes, as illustrated in figure 6. Therefore this example demonstrates a combination of existing modular ”building blocks” and a simple Python script.

Two VRPNSource nodes represent tracking information directly coming from VRPN [22]. The top node supplies tracking data for the dynamically changing coordinate system on which the remaining VRPNSource node is depending. Therefore the positional information is extracted by means of the attribute functionality in the Merge node first.

In a next step, EventInvertTransform inverts positional data, which is needed to define the proper coordinate system for the remaining VRPN tracking input connected to the EventDynamicTransform node. Its coordinate system shall be relative to the tracking position of the top VRPNSource node. The output is now enhanced by Go-Go interaction technique features as depicted before.

Afterwards, the initial coordinate system transformation is reverted, as the EventDynamicTransform node is configured complementarily to the former one. The result of this transformation is again tracking data which is expressed in an absolute world coordinate system.

Finally, modified position information is again remerged with all other remaining attributes of the tracking device.

If desired, it is of course possible to move Python code back into the OpenTracker C++ code base. This time such a modified implementation seems to comply more with the modular approach (described in section 4.2), as the original implementation was already mentioned as being a mixture of the modular and scripted approach. A C++ tracking node with Go-Go interaction technique features is similar in its class definition and calling mechanism to the original Python node. So, it is as simple as that to move on from a rapid prototype to a permanent implementation in OpenTracker code.

However, the advantage of keeping scripted implementation is to maintain additional flexibility. Only with this approach a repository of 3D UI techniques can be established in order to truly distribute program code at runtime without a priori knowledge of particular techniques at compile time. It is even possible that no information about a particular technique exists at runtime until the script code is obtained from the repository.

6 CLASSIFICATION

3D user interaction techniques have been classified into sets of categories by various researchers. We have chosen a similar classification system like Bowman et al. [5]. In addition to this taxonomy, we introduce an orthogonal property to describe the level of support.

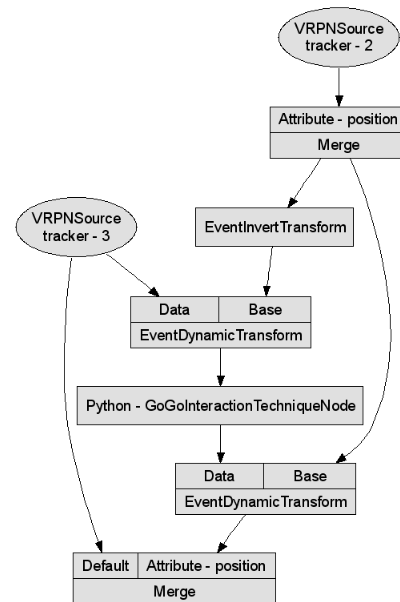

Figure 6: Data flow graph of the Go-Go interaction technique

In terms of supporting 3D interaction techniques in tracking middleware, we observed three different types:

- *Full support*: The 3D UI technique can be directly and fully implemented in tracking middleware.
- *Helper function*: Important tasks of the 3D UI technique have to be carried out on the application level, but some data setup and preprocessing tasks can be accomplished in middleware.
- *Lack of support*: Currently, the 3D UI technique has to be part of the application level. Tracking middleware can not assist in performing the 3D UI technique in a useful way.

In the following, 3D UI techniques (classified in a similar, but simplified manner to Bowman et al. [5]) can be assigned to these three types of support. However, it should be noted that the three levels of support and the implementation techniques as described in section 4 are independent of each other and should not be confused.

6.1 Full support

Most *navigation* and travel techniques can be fully supported in tracking middleware.

Generally speaking, physical locomotion and steering techniques (including semiautomated steering) require transformation or generation of a (relative) coordinate system: Depending on the particular technique, position and orientation data can be determined by simply inspecting input data of travelling-related tracking devices. For example, the tracking middleware can compute a user's position by using data from a steering wheel, combining it with data coming from an acceleration and brake pedal. Of course, this is also possible for any other tracked device used for steering (e.g. gaze-directed steering or steering by pointing).

In addition, some route-planning techniques are well supported: These two-step techniques require a state machine and usually involve input from another digital device (e.g. a button). Considering route-planning as marking points along a travelling path, the implementation is also easily possible. The state machine basically reacts to various events in order to switch between path-planning and actually carrying out the plan.

A practical example would be full support of the ChairIO interface by Beckhaus et al. [2]. All *navigation* interaction techniques as described in the paper (including initialisation) can be fully implemented in tracking middleware. Decoupling these techniques from the application allows flexible use of this innovative interface in various MR applications.

Selection and *manipulation* techniques are not that easy to implement, but core features of the Go-Go [16] and the World-in-Miniature technique [20] are fully supported. Both techniques rely on coordinate space transformation and can be done in tracking middleware. The Go-Go technique is characterized by its non-linear coordinate space transform: At a certain configurable minimum distance around the user, the coordinate system is scaled as described in section 5.2. Scaling is also the key for the World-in-Miniature (WIM) technique, where the user faces an additional down-scaled representation of the virtual environment. By performing these scaling operations directly in middleware the IT is completely transparent to the application and is replaceable.

Various *selection* techniques (e.g. lasso) make use of gestural interaction. Interpretation of gestures such as deletion, selection and other forms of manipulation can be handled directly by maintaining a history of previously recorded events as in section 5.1. Gestural commands can also be used for tasks in *system control*. In addition, *system control* and *symbolic input* techniques like voice input can be processed perfectly in middleware. The application is only informed of the recognized command to react properly.

6.2 Helper Function

The main cause for not achieving full support for a certain 3D UI technique is the object based nature of the particular technique. Usually, almost all *selection* techniques are object based by definition. For such an object based technique, data about all selectable objects would have to be accessible from within tracking middleware. So, a mapping between these objects and corresponding representations in middleware could be set up. In practical applications with a huge number of selectable objects this might not be suitable, since there is a lot of communication effort to keep the object mappings up to date. However, helper support can be achieved by preprocessing input data in middleware, before actually performing object selection in the application.

For example, a two-handed pointing technique can be handled by the application just like a simple ray-casting technique [4], when the middleware performs corresponding data setup. The tracking framework carries out all data conversion making the IT transparent to the application, the application implements only simple ray-casting. This data conversion and transparency attempt is one possibility to perform helper functions in middleware.

Also, precalculation can be done in the Flashlight and Aperture [9] techniques: Even if the selection itself can not be performed in middleware, the selection cone dimensions can be generated by the OpenTracker framework and expressed by custom attributes. So, additional data generation for further use in the application is another option for this helper function category.

A combined IT like HOMER [4] can be done partly in tracking middleware. Coordinate transformation in the manipulation mode can take place in the tracking framework, even if switching between selection and manipulation mode is controlled by application. This requires bidirectional tracking data transfer: Custom events identifying mode changes can be communicated back from the application to the tracking toolkit to indicate the current mode. The scaled-world grab technique [12] is similar in its approach by applying scaling in the manipulation step. These combined techniques demonstrate the possibility for bidirectional communication between tracking middleware and application.

This two-way communication can also be used in *system control* techniques. Meta information about graphical menus can be sent from the application to tracking middleware. Graphical representation of visual feedback has to reside on the application level, but command selection takes place in middleware. In return, the application is informed of user interface manipulation results, executes commands and provides feedback. This allows exchangeable and extensible *system control* techniques. Handling the menu interface this way, speech input can easily be managed too.

Another practical use case of bidirectionality is the definition of constraints within the application. In *modeling tasks*, constraining 3D (6DOF) input is important for precise input. Mine [11] and other authors (e.g. [3]) suggest in various studies that for direct input in 3D space six degrees of freedom are not expedient most of the time. Therefore it is very reasonable to restrict the user's input to two dimensions, for instance by using supporting planes. Positional and orientation data can be restricted independently. The defining parameters of a plane can be transferred to the tracking framework to establish 3D input coordinate restriction (e.g. snapping to that plane).

6.3 Lack of Support

Unfortunately, some 3D UI techniques have to remain being completely implemented on the application level and tracking middleware can not help with certain subtasks. ITs exhibiting strong coupling with objects or their virtual representation are more likely to fall in this category.

One example of this would be: The Voodoo dolls technique [15] requires access to objects, which are defined in the application. Therefore any processing of these objects can only be handled on the application level.

Image plane ITs [14] might be too difficult to implement in middleware as well because of the same reasons. An application-only implementation seems to be more practical, as even gesture processing IT subtasks are strongly related to virtual objects.

The same applies to other techniques like the Magic Mirror [10] or the Through The Lens technique [21]: Viewport rendering for graphics mapped onto interaction devices has to reside on the application level. Coordinate space transformation calculations are rather complex and not practical to implement in middleware.

7 Conclusion

We demonstrate three approaches and implementation techniques to extend the functionality of tracking middleware. By extending OpenTracker to allow Python scripting, a wide range of possible applications emerge: We are currently establishing a central repository of common ITs. With such a repository dynamic IT adaptation can be easily achieved by utilizing network communication.

Rapid prototyping of 3D interaction techniques is important for testing, studying and evaluating user performance and usability of MR applications. Our scripting approach enables testing of new

experimental techniques without changing software components of the application or the VR framework.

Strong interoperability between OpenTracker and VRPN takes advantages of both tracking frameworks: Numerous tracking devices are supported by both and ready for use. Many different processing features such as data filtering and merging are provided. Transformations and more complex tasks (e.g. some ITs) can be applied to input data (as described in section 5).

ITs written once can be reused by others in their own applications as long as the application itself or the related MR framework uses or supports VRPN or OpenTracker as tracking middleware. In this context the idea of an IT repository appears to be very interesting again in order to build up a library of various 3D UI techniques.

8 Future Work

A survey in 1992 [13] showed that about 50% of application development code (and time) was used for the applications' user interfaces. From our experience with developing complex MR environments, we believe that this may hold true for MR applications as well. Decoupling user interface code from the application could be a first step towards standardizing building blocks (widgets) which should be independent of the application or the higher level VR framework. Since most application developers are using tracking middleware in order to get hardware support for interaction devices (without having to implement it themselves), tracking middleware might serve as a common ground for future user interaction standards.

We think that by using an approach as described in this paper, most code regarding menu system interaction can be decoupled from the application and handled by the tracking middleware. A standardized XML file, specifying the menu layout and widgets could be passed by the application onto the tracking middleware which then sends higher level commands back to the application in case of user interaction. Widget behaviour itself could all be handled by the tracking framework if desired. Another advantage of this approach is obviously consistent application behaviour in distributed MR scenarios with minimal network load. We plan to continue our work in this area.

Furthermore, we believe that building IT repositories is a good way to move on towards the goal of a flexible and adaptive tracking system. In a pervasive MR scenario ITs must be accommodated to local hardware setups. The scripted approach is most appropriate for choosing a suitable IT for each setup and location in a dynamic adaptation process.

Acknowledgements

We thank all developers of OpenTracker, especially Gerhard Reitmayr, for their excellent work throughout the years. This work was funded in part by the Austrian Science Fund FWF project P19265.

References

[1] D. M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *4th Annual Tcl/Tk Workshop '96, July 10–13, 1996. Monterey, CA*, pages 129–139, Berkeley, CA, USA, 1996. USENIX.

[2] S. Beckhaus, K. J. Blom, and M. Haringer. Intuitive, hands-free travel interfaces for virtual environments. In *VR2005: Proceedings of the Workshop "New directions in 3D User Interfaces"*, pages 57–60. Shaker Verlag, 2005.

[3] D. A. Bowman. *Interaction techniques for common tasks in immersive virtual environments: design, evaluation, and application*. PhD thesis, 1999. Adviser-Larry F. Hodges.

[4] D. A. Bowman and L. F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–ff., New York, NY, USA, 1997. ACM Press.

[5] D. A. Bowman, E. Kruijff, J. J. LaViola, and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[6] T. L. Cottom. Using SWIG to bind C++ to Python. *Computing in Science and Engineering*, 05(2):88–96, c3, 2003.

[7] P. Figueroa, R. Dachselt, and I. Lindt. A conceptual model and specification language for mixed reality interface components. In *VR '06: Proceedings of the Workshop "Specification of Mixed Reality User Interfaces: Approaches, Languages, Standardization"*, pages 4–11, Washington, DC, USA, 2006. IEEE Computer Society.

[8] P. Figueroa, M. Green, and H. J. Hoover. InTml: a description language for VR applications. In *Web3D '02: Proceeding of the seventh international conference on 3D Web technology*, pages 53–58, New York, NY, USA, 2002. ACM Press.

[9] A. Forsberg, K. Herndon, and R. Zeleznik. Aperture based selection for immersive virtual environments. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 95–96, New York, NY, USA, 1996. ACM Press.

[10] J. Grosjean and S. Coquillart. The magic mirror: A metaphor for assisting the exploration of virtual worlds. In *Proceedings of 15th Spring Conference on Computer Graphics*, pages 125–129, 1999.

[11] M. R. Mine. Working in a virtual world: Interaction techniques used in the chapel hill immersive modeling program. Technical Report TR96-029, UNC Chapel Hill, Chapel Hill, NC, USA, 1996.

[12] M. R. Mine, J. Frederick P. Brooks, and C. H. Sequin. Moving objects in space: exploiting proprioception in virtual-environment interaction. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 19–26, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[13] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, NY, USA, 1992. ACM Press.

[14] J. S. Pierce, A. S. Forsberg, M. J. Conway, S. Hong, R. C. Zeleznik, and M. R. Mine. Image plane interaction techniques in 3D immersive environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 39–ff., New York, NY, USA, 1997. ACM Press.

[15] J. S. Pierce, B. C. Stearns, and R. Pausch. Voodoo dolls: seamless interaction at multiple scales in virtual environments. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 141–145, New York, NY, USA, 1999. ACM Press.

[16] I. Poupyrev, M. Billinghurst, S. Weghorst, and T. Ichikawa. The go-go interaction technique: non-linear mapping for direct manipulation in VR. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 79–80, New York, NY, USA, 1996. ACM Press.

[17] G. Reitmayr and D. Schmalstieg. An open software architecture for virtual reality interaction. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 47–54, New York, NY, USA, 2001. ACM Press.

[18] G. V. Rossum. *Python reference manual*. Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1995.

[19] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Trans. Inf. Syst.*, 11(3):287–317, 1993.

[20] R. Stoakley, M. J. Conway, and R. Pausch. Virtual reality on a WIM: interactive worlds in miniature. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 265–272, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[21] S. L. Stoev, D. Schmalstieg, and W. Straßer. Two-handed through-the-lens-techniques for navigation in virtual environments. In *Proceedings of the Eurographics Workshop on Virtual Environments*, pages 51–60, 16-18 May 2001.

[22] R. M. Taylor, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61, New York, NY, USA, 2001. ACM Press.