# Multi-modal event streams for virtual reality

J von Spiczak[1,2*], E Samset[1,5,6], S DiMaio[1,6], G Reitmayr[3], D Schmalstieg[4], C Burghart[2], R Kikinis[1,6]
[1]Brigham and Women's Hospital, Boston, MA, USA
[2]Technical University of Karlsruhe, Karlsruhe, Germany
[3]Cambridge University, Cambridge, UK
[4]Graz University of Technology, Graz, Austria
[5]University of Oslo, Rikshospitalet, Oslo, Norway
[6]Harvard Medical School, Boston, MA, USA

## ABSTRACT

Applications in the fields of virtual and augmented reality as well as image-guided medical applications make use of a wide variety of hardware devices. Existing frameworks for interconnecting low-level devices and high-level application programs include high-frequency event streaming frameworks as well as dynamically typed message passing systems. Neither of these approaches exploits the full potential for processing events coming from arbitrary sources and neither is easily generalizable. In this paper, we will introduce a new multi-modal event processing methodology that uses dynamically-typed event attributes for event passing between multiple devices and systems. The existing OpenTracker framework was modified to incorporate a highly flexible and extensible event model, which can store data that is dynamically created and arbitrarily typed at runtime. The software architecture that is introduced in this work provides multi-modal event streaming, network and file logging support, and the ability to merge separate event streams. Due to the implementation of a self-descriptive event serialization mechanism, the architecture does not put any constraints on new data types to be used within the library. The main factors impacting the library's latency and throughput were determined experimentally and the overall performance was shown to be sufficient for most typical applications. Several sample applications were developed to take advantage of the new dynamic event model that is provided by the library, thereby demonstrating its flexibility and expressive power.

**Keywords:** Device Interface, Middleware, Tracking Systems, Mobile Augmented Reality, Virtual Reality, Generic Programming

## 1. INTRODUCTION

Tracking and navigation systems, peripheral devices, image acquisition, image visualization, and data logging play an indispensable role in the field of virtual and augmented reality (VR and AR, respectively) as well as image-guided medical applications. A wide variety of hardware devices typically need to be supported, including:

- *Input Devices* such as tracking systems, image and monitoring sources, other signal acquisition devices,
- *Output Devices* like displays, visualization systems, robots,
- *Input/Output Devices* like triggers, selectors, controllers, medical interventional devices, and file servers.

In order to integrate such systems and devices in a flexible and extensible manner, it is necessary to have a software framework that provides a high-level abstraction of hardware devices and their associated low-level software drivers. This framework needs to efficiently handle streams of data in real time, potentially using a pipes-and-filters architecture. In addition, transparent network support is desirable for distributed systems.

Software systems have been developed to give formerly independent devices the ability to cooperate in a straightforward way [2, 3, 6 – 9, 11 – 13, 15, 16]. However, many of these solutions do not exploit the full potential for processing events coming from arbitrary sources and are not easily generalizable. Prior work in this area can be divided into two different classes of architectures.

The first class uses an efficient fixed data structure for the streamed events that can be accessed, copied, and processed with minimum delay. This is therefore suitable for high-frequency, real-time streaming of event data; however, the

* Please send all correspondence to Jochen von Spiczak at jochen.v.s@gmx.net.

predefined, fixed event structure is limited in its applicability (e.g., it may be designed exclusively for tracking, image or video streaming).

A second class of architecture implements processing of multi-modal events, the content of which can be flexibly defined. The event format encapsulates data of different types in a potentially self-descriptive way. Such systems are often used for asynchronous event processing and typically achieve lower performance due to the associated overheads. Therefore, they can often more appropriately be described as message passing, rather than stream processing systems.

In this paper, we introduce the concept of streaming multi-modal events with generic types. Following this approach, events are generated dynamically at runtime—as required by the application—and can contain data of arbitrary types, including composite high-level types defined by the user. The structure of each event can change during the lifetime of the event—information can be added, accessed, modified, and deleted. Event streams containing data from different data sources, which logically belong together, can be merged and handled as a single event stream. The approach is suitable for high-performance VR applications that require both high-bandwidth event streaming and support for multiple heterogeneous devices. This software framework has been released under the BSD Open Source license [1] and is freely available.

## 2. RELATED WORK

The framework presented here for multi-modal event processing can be seen in the context of other projects that have dealt with the trade-off between high-frequency stream processing and more flexible event passing in a number of applications. Device abstraction is a standard requirement for graphical user interfaces and VR systems. Examples include the MR-Toolkit [2] and trackd [16], which is commercially available. VRPN [15] is a well-known library that implements a network-transparent interface between application programs and a limited set of physical devices (especially tracking systems).

Stream processing frameworks exist for different types of real-time data, for example Microsoft DirectShow for video [12] or Chromium [6] for OpenGL streams. These frameworks generally use the pipes-and-filters design pattern [5] for flexible composition of filter nodes in a graph structure. Data is handled at high frequencies, but conforms to a fixed data format (or a small set of formats) that cannot be changed at runtime. OpenTracker 1.1 [7, 8], which is the basis for the work presented in this paper, also falls into this category.

In contrast, message passing systems handle data of various formats in an asynchronous and potentially infrequent manner. All conventional 2D windowing systems fall into the category of such architectures and most current VR frameworks incorporate such a mechanism. For example, VRML [13] uses so-called ROUTES to interconnect multiple components of a scene graph.

VR applications are often distributed over multiple systems and thus require network-transparent transport of events. Examples for distributed scene graphs are DIVE [3] and Avocado [9]. DWARF [11] represents a new generation of VR/AR component systems that can dynamically interconnect components, but still require pre-defined event types to determine compatible components.

## 3. BACKGROUND: OPENTRACKER

The goal of this paper is to explore a new trade-off between stream processing and event passing, combining the typical advantages of both approaches. Since the basis of our work is OpenTracker, we present a short overview of the library's main concept in this section.

OpenTracker is an open software architecture and its previous version 1.1 [7, 8] provides a framework for integrating tracking input devices and algorithms for processing 6-degree-of-freedom (DOF) input data in single or multi user VR and AR applications. The library supports a large variety of tracking systems and allows configuration of stream processing graphs for more complex operations. As is typical for stream processing frameworks, event attributes are restricted by a predefined State class (cf. Fig. 1).

New devices can be added to the framework by implementing new modules that translate the device-specific information into the standardized OpenTracker data structure. OpenTracker acts as an abstraction layer unifying output data from multiple tracking systems, providing data handling and flexible configuration of different hardware setups. The data flow of each setup can be defined by a stream processing graph composed of nodes, through which the tracking data flows.

When multiple tracking systems and software applications are connected, data is passed through a series of steps. Data is contained in an event object, which is created by a source node. Source nodes provide the abstraction between the internal event data model and the data source, namely a peripheral tracking device, a file, a user application, or other type of input. The event can pass several transformation steps (filters, transforms, or data merging) before it reaches an output port in the stream processing graph—a sink. The sink is responsible for piping the data to its designated output (e.g., a virtual object, a virtual camera or light source, a file) or for modifying other functional parameters of the application. The presence of network sinks and sources enables distributed tracker data acquisition and processing.

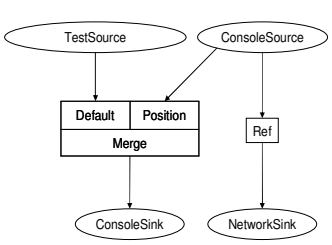
| **State** |
| --- |
| + position: float[3]<br>+ orientation: float[4]<br>+ button: unsigned short<br>+ confidence: float<br>+ time: double<br>+ *null: State* |
| + State(time: double = 0.0, confidence: float = 0.0)<br>+ State(rv: State&)<br>+ operator=(rv: State&): State&<br>+ timestamp(): void |

Fig. 1: UML definition of the OpenTracker 1.1 State class

The definition of the data flow is central to the architecture of OpenTracker. A data flow is set up by connecting different nodes in a graph structure, as defined by a user-written XML configuration file [17]. This file is required to be written according to a predefined document type definition (DTD). Fig. 2 gives an example of a data flow graph that can be built with OpenTracker. Events are created by two data sources and combined by a MergeNode. Multiple data sinks consume different parts of the created and transformed data.

To create more than simple linear graphs, the following concepts allow more complex data flows to be defined:

- *Multiple Input Ports* allow nodes to receive data from more than one input,
- *References* (i.e., Ref nodes) allow for transparent reuse of output data,
- *Different Edge Types* distinguish between event passing, event queuing, and time dependent storage of events, and therefore allow for data pushing and polling.




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OpenTracker SYSTEM "opentracker.dtd">
<OpenTracker>
  <configuration>
    <ConsoleConfig headerline=""headerline""/>
  </configuration>
  <ConsoleSink comment=""consoleSink"">
    <Merge>
      <MergeDefault>
        <TestSource noise="5"/>
      </MergeDefault>
      <MergeAttribute attributeName="Position">
        <ConsoleSource number="2" DEF="reference"/>
      </MergeAttribute>
    </Merge>
  </ConsoleSink>
  <NetworkSink name=""networkSink"" ...>
    <Ref USE="reference"/>
  </NetworkSink>
</OpenTracker>
```

Fig. 2: Control flow graph and its XML configuration

By encapsulating any additional functionality in separate modules and node classes, the OpenTracker framework can easily be customized or extended by loading the required modules or by adding new, user-defined modules that support new functionality.

# 4. MULTI-MODAL EVENTS

By refactoring the data model of the OpenTracker library, a general device interface library was developed for arbitrary integration of tracking systems, imaging systems, peripheral devices, device controllers, data loggers, and other systems. It uses an underlying data structure that is flexible and extensible, so that new devices and applications can easily be integrated.

## 4.1 Architecture of multi-modal events

An event structure, consisting of several attributes that can be created dynamically, has been introduced. Each node creating or transforming an event can dynamically add new attributes or access them by name. A large variety of different data types are supported for defining event attributes, and can be further extended to include user-defined types. The architecture puts no constraints on the data types to be used within the library. Data types currently supported are:

- *Basic Data Types* like characters, integers, floats, doubles, and strings,
- *Higher Level Types* like vectors, lists, matrices, speech tokens, and images.

The key feature of the architecture is the dynamic combination of multiple event attributes of different types within the event class, which allows grouping of attributes that belong together (such as simultaneous measurements) in a single event instance. The architecture uses the generic programming paradigm, which allows instantiation of a single class template with multiple types, to realize event attributes of generic types.

A simplified class hierarchy of the core classes is shown in Fig. 3. The Event class was realized as a map that matches names (strings) to corresponding attributes (typed data values). It provides the only public interface to access the information stored in the event's attributes.

All aspects regarding attribute construction, handling, and deconstruction are completely encapsulated behind the Event class's interface. The EventAttribute class was templated on the type it wraps, and contains one member of that type, along with access methods.

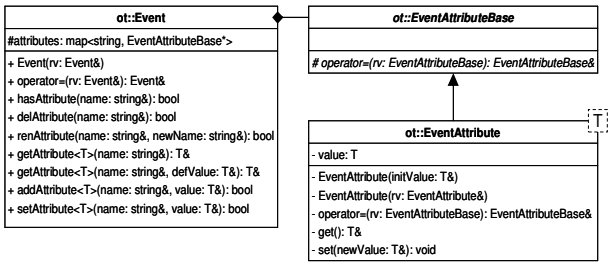



Fig. 3: Simplified class hierarchy of core classes

### 4.2 Event handling

Nodes operate on events by looking for a known attribute name and then checking the type for safety. The semantics of event attributes is established by naming conventions.

The set of attributes in an event is introduced dynamically. When a node attempts to access an attribute by a name not known to the event, this attribute can be created transparently inside the event.

Since the event data structure is not fixed and therefore complex and time-consuming to copy, events are passed through the graph and modified along the way rather than being copied into a member variable of each node and modified there. However, because a node's output can be connected to several inputs, one copy needs to be made for each connection, so that they can be changed independently. This concept was implemented within the Ref nodes (cf. Section 3).

Multiple events can be passed through the stream processing graph along different paths, so that the data flow graph can serve different purposes simultaneously. Thus, many parallel event streams can be processed with possibly different event frequencies.

### 4.3 Extension of modules and nodes

The main functionality of OpenTracker is implemented in a small set of core classes, whereas additional functionality is provided by a set of module and node classes. The introduction of the new multi-modal event model required a redesign of most core classes, as well as modification of several modules and nodes.

#### 4.3.1 Nodes requiring serialization

Event distribution is an essential feature of the OpenTracker library and facilitates:

- using events on multiple host computers for a distributed virtual environment,
- distributed processing,
- spanning multiple operating systems (e.g., if a specific hardware device is only available on one particular host),
- storing event information and replaying it at a later point in time.

Intercommunication of configurations running on multiple hosts within the same network was implemented in the NetworkSink- and –SourceModule. The FileModule can store event information in a file and retrieve such information to push it into the stream processing graph.

The library's network and file logging support imply the need for an automatic, self-descriptive serialization/deserialization mechanism to store, broadcast, and access data in a unified way on multiple systems.

Generic network transmission was implemented to facilitate dynamic events and allow for event processing on distributed computers within the same network. The architecture allows both bi-directional one-to-one IP unicast and multiple senders and receivers to communicate asynchronously through IP multicasting. The event is serialized by the NetworkSinkModule on the sender side using the architecture's event streaming mechanism. On the receiver side, the received data is streamed into an event object by the NetworkSourceModule and then pushed into the local stream processing graph.

Since the architecture's event streaming mechanism uses general type names that are defined during type registration and are therefore compiler independent, the broadcasted data is valid independent of the host operating system.

The FileModule provides storage functionality to save event information and to retrieve such information at a later point in time. For the purpose of this discussion, it operates in much the same way as the network modules, in that it requires the ability to serialize/deserialize data when writing/reading event data to/from a file.

### 4.3.2 MergeNode

A MergeNode was designed to process parallel streams of multi-modal data in a synchronized fashion. Events from different branches of the stream processing graph may hold data that logically belongs together, but is processed independently by multiple event streams. Using a MergeNode, such separately acquired event data can be combined into one single event stream. Fig. 4 illustrates the concept of the MergeNode.

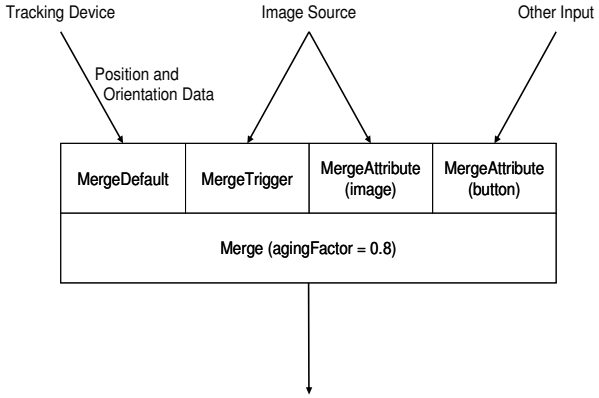



Fig. 4: MergeNode combining multiple input streams

Multiple event streams are connected to the input ports of the node as determined by the configuration file. The MergeNode stores a local event object, which is updated with parts of every incoming event by adding or modifying the corresponding attributes. The node then passes the merged event on to its successors. The following concepts were developed to meet the complex needs of multi-modal event streams:

- Any number of input streams can be merged into one single output stream. Every input port defines which of its event attributes should be merged into the resulting event.
- A MergeDefault port can be defined, which updates all attributes not handled by any other input port.
- Timestamps are set by the MergeTime input. If no stream is connected to this input, the timestamp of the new event always equals the timestamp of the last received event.
- The output stream can be triggered by an input port called MergeTrigger. This concept can be used when there are inputs with different update rates, but only the combination of all updated inputs is interesting for succeeding nodes.
- Assuming that confidence information of incoming events is stored in an attribute conventionally called ‘confidence’, the confidence value of the merged event can be modified to reflect obsolescence of input ports that have not been updated. Whenever a new event arrives, the confidence of all other input ports is multiplied by an aging factor specified in the configuration file. The overall confidence of the resulting event is then calculated as the minimum (default), maximum, or product of all input ports.

## 5. IMPLEMENTATION

The described architecture was implemented as a customizable library in the C++ programming language in a strictly object oriented way. C++ supports the paradigm of generic programming via class and function templates. The C++ Standard Template Library (STL) provides a wide variety of predefined container class templates [14]. Class template capabilities were used for implementing the generic event attribute class, derived from the non-template EventAttributeBase super class. Function templates were used within the Event class to access the event’s attributes of generic type.

The STL Map class realizes an associative array that uses keys for indexing and addressing stored elements. Instantiated with the STL String class as the key type and EventAttributeBase pointers as elements, this container is used to match attribute names to their corresponding attribute values.

The mechanism to automatically serialize event data into self-descriptive data streams was the most complex part of the implementation and is therefore described in more detail.

### 5.1 Self-descriptive serialization

I/O streaming operators were used to implement the serialization/deserialization mechanism for all data types. Streaming operators in C++ are declared in the following format (using namespace std):

- `istream& operator>>(istream &in, ClassName &obj);`
- `ostream& operator<<(ostream &out, const ClassName &obj);`

I/O streaming of primitive data types is handled by the C++ standard library itself. Streaming operators for the most important STL container classes (i.e., vectors and lists) were implemented as a part of OpenTracker. Overloaded streaming operators are expected from user-defined data types. They are to be implemented outside the class, but preferably declared within the same header file, so that they are known wherever the new type is used. Streaming operators for STL container classes were cumulated in an extension of the C++ I/O stream library within the OpenTracker framework.

Serialization of an event and its attributes into an output stream or into a character array is illustrated in Fig. 5. The serialization operator of the Event class writes out the event's timestamp, the attribute count, and a list of all attributes. Attributes are encoded by the generic type name that is member of the attribute class, the attribute's name, and its current value. The value is streamed by the aforementioned output streaming operator.

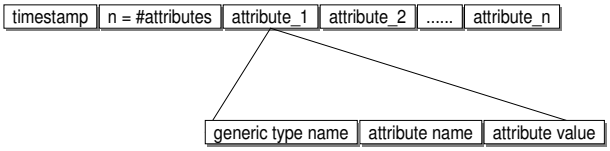



Fig. 5: Serialization of an event and its attribute map

Deserialization of event attributes is handled by an abstract method of the EventAttributeBase class and its implementation in the EventAttribute class template, which streams data provided by the input stream into the encapsulated value. Before data can be streamed into an event attribute, an attribute has to be created according to the generic type name field. Therefore, each Event-Attribute template instantiation provides a static creator method, which creates a new object of the class on the heap and returns a pointer to the object:

- `static EventAttributeBase* create() { return new EventAttribute<T>; };`

A C++ data type called CreateFunction was defined for easier handling of function pointers matching the declaration of such create functions:

- `typedef EventAttributeBase * (*CreateFunction)(void);`

To be able to call the create function of a specific EventAttribute<T> class according to a given generic type name (i.e., the generic type name of type T), each generic type name must be concatenated with the according create function. This is realized by a static member variable of the EventAttributeBase class, being a map from type names to create functions (using namespace std):

- `static map<string, CreateFunction> creators;`

All data types that are used as event attributes must be added to this map, together with their create function. The generic type name is arbitrary but must be unique. Data types are added to the map by calling the templated register function provided by the Event class (using namespace std):

- `registerGenericTypeName<T>(const string &genericTypeName);`

All primitive C++ types and predefined high-level data types are automatically added during startup. Generic type names were chosen canonically for all these types (e.g., "vector<float>" is used as the generic type name of the std::vector<float> type). User-defined types need to be added to the map by calling the aforementioned registration method manually.

An input stream encoding an event attribute can now be deserialized by the following mechanism, which was implemented in the deserialization function of the Event class:

1. The create function belonging to the given generic type name is looked up in the creators map. Calling this method creates a new event attribute of the correct type.

2. The content of the attribute value field is streamed into the attribute's value using the input streaming operator for the attribute's data type. If the value field does not contain correctly encoded data of the right type, the failbit of the input stream is set and deserialization is stopped.
3. Upon success, the new attribute is added to the event's attribute map with the given attribute name.

# 6. PERFORMANCE EVALUATION

## 6.1 Evaluation procedure

A number of tests were undertaken to evaluate the effect of multi-modal event processing on overall performance. This evaluation was done using a variety of test configurations designed to isolate the impact of the several different features and mechanisms implemented within the software system.

In all test configurations, events are created by a TestSource, processed by a number of filter nodes, and serialized and written to a file by a FileSink. Configurations varied in the following ways:

- *Event Model:* To determine the performance impact of the multi-modal event model, every setup was run on the former OpenTracker version 1.1 that uses a fixed event data-structure, as well as on the augmented version 1.2 with the fully flexible data structure described in this paper. To be comparable, OpenTracker 1.2 events were created with the same tracking related data fields as used in the former version.
- *Event Processing:* Two different concepts of event processing were used in comparable setups. EventTransformNodes served as an example for filter nodes that use internal copying to compute their output, whereas ThresholdNodes were used as an example of nodes that directly operate on the incoming event and pass it on without copying.
- *Complexity of the Stream Processing Graph:* Configurations with a different number of filter nodes were used to simulate the impact of graph complexity. Configurations with 0, 1, 10, and 30 filter nodes were analyzed for each event model and event processing concept.

Combination of all variations results in 16 possible test setups. Each setup was started and stopped after processing almost 40,000 events on average. By calculating the difference between starting time ($t_{start}$) and stopping time ($t_{stop}$) and dividing the result by the number of processed events (e), the time taken for a single event to pass through the test configuration (referred to as latency l) was determined. The frequency (f) determining the throughput of the system is computed as the reciprocal of the latency.

$$l = (t_{stop} - t_{start}) / e \qquad f = 1/l$$

Performance tests were run on an off-the-shelf computer workstation (Dell Precision™ Workstation 670, 2 Dual-Core Intel® Xeon™ CPU 2.80GHz, 7.72 GB RAM).

## 6.2 Performance measurements

In the experimental setup described in Section 6.1, the time required for a single event to pass through a sample stream processing graph (latency), as well as the frequency of event generation and processing (throughput), was measured. As described above, three main factors were observed in order to isolate their impact on the overall performance, namely the data model (i.e., multi-modal vs. fixed events), event processing method (i.e., event copying vs. event passing) and the number of filter nodes in the graph.

The outcome of the experiments investigating event latency is illustrated in Fig. 6, with numbers provided in Table 1. In Table 1, each row contains single-event latency measurements for 0, 1, 10, and 30 cascaded filter nodes. Each measurement was repeated for the same configuration running on OpenTracker version 1.1 with its fixed data structure and the new OpenTracker version augmented by multi-modal events, as well as for an event passing scheme (ThresholdFilter) and an event copying scheme (EventTransform).

In addition to the time that each event needs to pass through the graph, the frequency of event generation and processing was examined. This figure can be described as the overall throughput of the system. The results of these experiments are presented in Table 2 and illustrated in Fig. 7. The data rows show the frequency for the same setups described for event latency measurements.

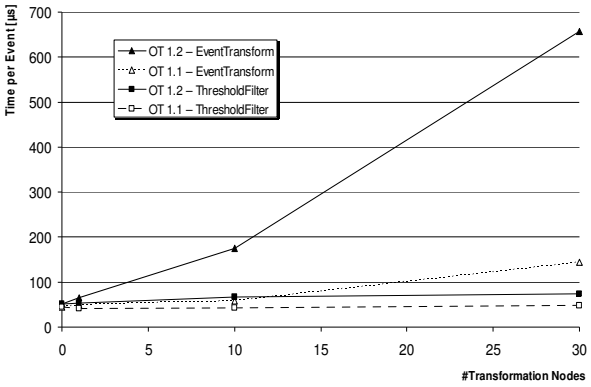



Fig. 6: Latency increase of single event

Table 1: Latency of one single event (in µs)

| | 0 | 1 | 10 | 30 |
|---|---|---|---|---|
| OT12 EventTransform | 50.66 | 65.26 | 175.81 | 657.6 |
| OT11 EventTransform | 44.13 | 49.05 | 58.82 | 144.92 |
| OT12 ThresholdFilter | 50.66 | 53.26 | 66.96 | 73.99 |
| OT11 ThresholdFilter | 44.13 | 40.83 | 42.86 | 48.25 |

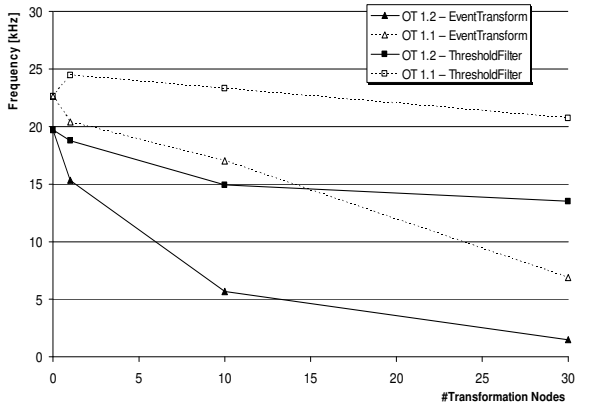



Fig. 7: Graph illustrating frequency decrease

Table 2: Event processing frequency (in kHz)

| | 0 | 1 | 10 | 30 |
|---|---|---|---|---|
| OT12 EventTransform | 19.74 | 15.32 | 5.69 | 1.52 |
| OT11 EventTransform | 22.66 | 20.39 | 17.00 | 6.90 |
| OT12 ThresholdFilter | 19.74 | 18.77 | 14.93 | 13.52 |
| OT11 ThresholdFilter | 22.66 | 24.49 | 23.33 | 20.73 |

# 7. APPLICATIONS

Several applications were successfully developed using the approach proposed in this paper, thus proving the usability of the framework. These applications are from different fields, including VR, AR, and medicine. All applications use the introduced, multi-modal event data structure to flexibly encode information of different types that can then be processed and used in multiple ways. Below, the applications will be described in more detail, each followed by an outline of the implementation and the results.

## 7.1 Haptic device interface

*Application description:* A module called PhantomModule was used to interface with haptic devices from SensAble Technologies, Inc. via the OpenTracker data format. Such devices allow touching and manipulation of virtual 3D data models. The PHANTOM® device provides a pen-shaped interface that can be held and manipulated by the user, as shown in Fig. 8.

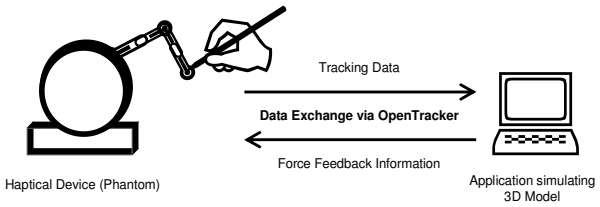



Fig. 8: Haptic device using OpenTracker data exchange

*Implementation:* The PhantomSource gets position and orientation data as well as the button state from the haptic device and translates such information into the OpenTracker data format. In turn, the PhantomSink interprets OpenTracker-encoded force-feedback information—generated using the OpenHaptics™ library—as separate event attributes and sends this back to the haptic interface for feedback to the user's hand. The setup of the system is illustrated in Fig. 8.

*Result:* The ability to create appropriately named attributes (e.g., instead of using the position field for force variables) and thereby maintain the semantics was demonstrated. Maintaining the correct semantics of data fields prevents

unexpected behavior if the stream processing graph is misconfigured, since only a consumer of force information would use the corresponding attributes.

### 7.2 MRI scanner interface

*Application description:* A module called MedScanModule allows interaction with medical Magnetic Resonance Imaging (MRI) scanners via data provided in the OpenTracker data format.

*Implementation:* The MedScanSink allows interpretation of OpenTracker information to control scan plane positioning and orientation. Moreover, additional scanner parameters can be controlled such as slice sickness, field of view (FOV) and start and stop of a scan. The MedScanSource provides images taken by the scanner, which are encoded in the OpenTracker format. The generated image events also contain position information and the orientation of the image.

*Result:* The ability to create non-tracker-related event types such as images and control variables was demonstrated. The application dynamically creates event attributes correctly typed for the current purpose and names them according to the actual semantics. Moreover, integration of high-level user-defined types was proven to be usable.

### 7.3 Voice command interface

*Application description:* An OpenTracker module called SpeechControlModule was developed. The module realizes voice command recognition and modification of 6DOF tracking attributes as well as other attributes according to recognized commands.

*Implementation:* The module allows a set of specified commands to be recognized and translated into the corresponding modification of position and orientation data as well as arbitrary additional parameters, or an adjustment of the program's internal state. The command-set includes different kinds of translations and rotations, as well as predefined poses. Moreover, there are commands to start and stop continuous movements, to save and restore prior states, commands for undo/redo functionality, scaling, state logging and additional control commands. The module also provides feedback for the user, spoken by the system's synthesized voice. Since the module's design encapsulates all recognizable voice commands and their meanings, the set of commands can easily be extended, constrained, or modified to allow for usage within multiple fields of application.

*Example of use:* The SpeechModule was combined with the MedScanModule to implement a voice command interface for real-time interventional MR imaging in the context of medical applications [10]. The system is illustrated in Fig. 9. It allows for interactive control of the scan plane position and orientation from inside the scanner room. The novel interface gives complete voice control of MRI scanning to the user and allows for faster, direct adjustments of the scan plane.

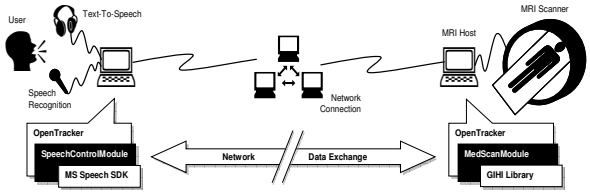



Fig. 9: Voice command interface for interventional MRI

*Result:* The module that was originally developed for OpenTracker version 1.1 was enhanced by making use of the new multi-modal event framework. This demonstrates that applications can greatly extend their functionality and potential field of application by combining the former standard data fields with dynamic and flexible event attributes.

### 7.4 ECG triggered, MRI guided navigation for cardiac interventions

*Application description:* A patient's electrocardiogram (ECG) signal is sampled and encoded in an OpenTracker event by an ECGModule. Navigation software for cardiac intervention interprets the heart rate attribute of the incoming event and synchronizes visualization of a set of pre-operatively acquired MRI images of the complete heart cycle with the patient's current heart rate [4] (cf. Fig. 10).

*Implementation:* The ECGModule samples a patient's ECG signal using a data acquisition interface (National Instruments™ USB-6008) and analyzes the signal to detect

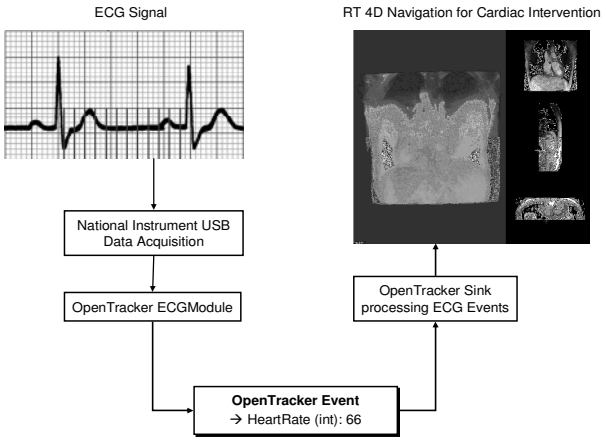



Fig. 10: ECG triggered, MRI guided navigation for cardiac interventions

the heart rate. An event is dispatched for every QRS complex that is detected and includes information about the heart rate.

*Result:* This sample application does not involve tracking or the use of any position or orientation data, thus illustrating the general applicability of the new framework.

### 7.5 Tracked ultrasound imaging

*Application description:* Combination of ultrasound (US) imaging, tracking, and button activation serves as a realistic example for making use of the multi-modal functionality provided by OpenTracker. A tracking device can be rigidly attached to an ultrasound probe and the information streams coming from both devices can then be merged with additional button switch information from a third device. The combined event stream can then be fed to a visualization application for display. The setup of the system is illustrated in Fig. 11.

*Implementation:* A new component called TerasonModule drives Terason Ultrasound (US) probes, which plug directly into the computer via the FireWire port. The TerasonSource provides events containing US images (stored in a user-defined image class) and related information, e.g., image resolution. The probe can also be controlled by using a TerasonSink, so that US frequency and other parameters can be set by the OpenTracker configuration. A transformation node registers the ultrasound images to the coordinate frame of the attached tracking. A MergeNode then combines the event stream containing ultrasound images with the tracking stream. As illustrated in Fig. 4, information that comes from different branches of the stream processing graph but logically belongs together can be combined into one single output stream. Binary button information coming from a third device (e.g., a foot pedal) can be merged within the same step. Button activations can be interpreted in different ways, for instance they can freeze the image at the momentary position.

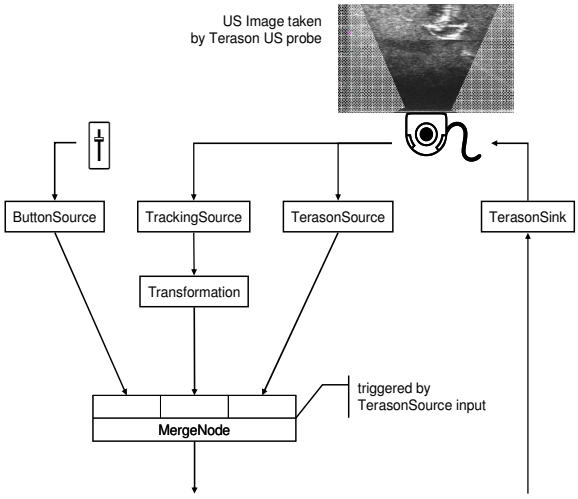


Fig. 11: Tracked US imaging and button switch

*Result:* The described setup successfully combines use of standard tracking attributes, incorporation of user-defined data types, and merging of multiple event streams. Distribution of system components over several computers optimizes performance and offers some robustness, since failure of one device will not bring down the rest of the configuration.

### 7.6 Real-time playback of event information

*Application description:* In this application, any information encoded in the OpenTracker event format can be replayed in real-time. VR interactions can be saved and displayed at a later point in time for monitoring, analysis, review, or documentation.

*Implementation:* Using a FileSink, information gathered during a human's interaction with virtual items within a virtual reality setup can be saved. Such data contains camera positions, position and orientation information of tracked objects, acquired images, switches, and other information about items within the virtual setup. The so-called EventPlayerSource retrieves information from the saved file and pushes out events in the correct order and with the right waiting period between successive events.

*Result:* This application demonstrated the usefulness of the event serialization/deserialization mechanism (in writing and reading events from file) and constitutes a general tool for reviewing interactions.

## 8. DISCUSSION

The software architecture described in this work combines simplicity and expressive power. By introducing events that are dynamically typed at runtime and that have a set of event attributes that can change over the lifetime of the event, any kind of event content can be realized according to the needs of nodes operating on the event, while avoiding the combinatorial explosion of types that would result from strictly typing all conceivable combinations. The design of event attributes following the paradigm of generic programming reduces the implementation effort and any possible

redundancy to a minimum, since only one class needs to be implemented and can be instantiated with any type. Even though there is not one special attribute class for each kind of data type, this approach offers type safety, since attributes are accessed via template get methods provided by the event class.

A general serialization/deserialization mechanism was found to be crucial for an efficient implementation of the architecture. In order not to affect the library's support for every arbitrary data type, this serialization mechanism does not make any assumptions regarding the data types that are stored in an event, but is fully generic and flexible. The only assumption made for high-level data types is the existence of custom I/O operators. Since such operators can easily be provided by the developer of a new class [14], this assumption does not put any constraints on the types to be used within the library.

Overall performance of the library is sufficient for most applications, but further improvements are possible. No special effort has been made to optimize performance. Time consuming operations include: event creation by a source, access, processing, and copying of events within a number of filter nodes, and serialization of events for file storage or network transmission. Three factors were individually examined to analyze their impact on these aspects and the overall performance:

1. *Multi-Modal vs. Fixed Events:* The more complex data structure of a multi-modal event model results in higher latency and a lower throughput of the overall system. However, results do not indicate that the complexity of multi-modal events is a significant performance issue, since the deterioration factor exceeds a value of 1.5 only for a large number (30) filter nodes, which use internal event copying.
2. *Event Copying vs. Event Passing:* Results indicate that internal event copying is the most important factor contributing to the decrease in overall performance. In a setup with 30 intermediate nodes, multi-modal event copying slows the system down by a factor of 8.9 compared to event passing. Future improvement will utilize event passing to a larger extent.
3. *Number of Filter Nodes:* Interestingly, the number of cascading filter nodes does not have a major performance impact if events are modified directly and passed on without internal copying. According to the observations above, a large number of copying nodes decreases performance significantly.

The reason for such major impact of internal copying is the time consuming copying mechanism for multi-modal events. To ensure correct initialization of event attributes, each attribute must be created, copied, and added to the resulting event individually, instead of copying the block of memory holding the event information at once. Memory allocation is one of the most expensive program operations and therefore influences overall performance significantly. Event handling can be optimized by minimizing or eliminating internal copying within filter and transformation nodes. Nevertheless, for most typical applications, the current performance of the architecture is sufficient. The lowest throughput of the system was measured for a cascade of 30 event-copying filter nodes (far more than typically required in practice) resulting in a 1.52 kHz event processing frequency, which is satisfactory even in applications with haptic interaction.

Usability of the software architecture has been demonstrated in a variety of VR, AR, and medical applications that could make good use of the flexible event data model. The following observations were made:

- Usability of event attributes that are dynamically created by nodes operating on the events was proven by all presented applications.
- Correct semantics can be maintained if event attributes are named according to their meaning by the node that adds the new information. Only nodes that expect attributes with a certain name and the according type process such attributes, which prevents unexpected behavior in a possibly malformed stream processing graph.
- High-frequency, real-time stream processing can be mixed with more infrequent handling of other events, which would normally be processed by a different platform realizing event passing.
- Integration of high-level, user-defined types (e.g., image classes) was demonstrated by several sample applications.
- Serialization/deserialization mechanisms were successfully used for file logging, real-time event playback, and distributed event processing.

- Some applications benefited from the MergeNode. Multiple event streams containing different information were combined at a certain point in the graph, so that data logically belonging together could also be physically merged into one single event instance. Thus, parallel streams of data could be processed in a highly synchronized fashion.

## 9. CONCLUSION

This work describes a new multi-modal event processing methodology that uses dynamically-typed event attributes for highly flexible and extensible event passing between multiple devices and systems. Dynamic introduction of any number of arbitrarily typed event attributes with late type-binding during runtime provides significant functional advantages over hard coded, fixed event data-structures, and is more flexible than simpler generic approaches that determine event data types at compile time. This flexibility comes at the expense of performance; therefore, further optimizations—including the elimination of any internal copying of events—will be made in future work.

OpenTracker is freely available under BSD license from `http://www.studierstube.org/opentracker`.

## ACKNOWLEDGMENTS

This study was supported in part by the NIH U41-RR019703 and NSF EEC-9731748 grants. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NIH or the NSF. We also acknowledge Arne Hans for his technical assistance.

## REFERENCES

1. *Berkeley Software Distribution (BSD) License,* http://www.opensource.org/licenses/bsd-license.php, visited May 2006.
2. C. Shaw, M. Green, J. Liang, Y. Sun, *Decoupled simulation in virtual reality with the MR toolkit.* ACM Transactions on Information Systems, 11(3):287–317, July 1993.
3. E. Frécon, M. Stenius, *DIVE: A Scaleable network architecture for distributed virtual environments.* Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments), 5(3), 91-100, Sept. 1998.
4. E. Samset, D. F. Kacher, P. Aksit, G. H. Reynolds, L. M. Epstein, F. A. Jolesz, *ECG Triggered MRI-Guided Navigation for Cardiac Interventions.* In Proc. ISMRM 2006, Seattle, Washington, USA.
5. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns, volume 1.* Wiley, Great Britain, 1996.
6. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, J. T. Klosowski, *Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters.* In Proc. of the 29th annual conference on Computer graphics and interactive techniques 2002, ACM, San Antonio, Texas, USA, pp. 693 – 702.
7. G. Reitmayr, D. Schmalstieg, *An open software architecture for virtual reality interaction.* In Proc. VRST 2001, ACM, Baniff, Alberta, Canada, pp. 47 – 54.
8. G. Reitmayr, D. Schmalstieg, *OpenTracker - A Flexible Software Design for Three-Dimensional Interaction.* In Virtual Reality, Vol. 9, No. 1, December 2005, Springer.
9. H. Tramberend. *Avocado: A Distributed Virtual Reality Framework.* IEEE Virtual Reality, 1999.
10. J. von Spiczak, E. Samset, D. F. Kacher, C. R. Burghart, F. A. Jolesz, S. P. DiMaio, *A Voice Command Interface for Real-Time Interventional MR Imaging.* In Proc. ISMRM 2006, Seattle, Washington, USA.
11. M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reichner, S. Riss, C. Sandor, M. Wagner, *Design of a component-based augmented reality framework.* In Proc. ISAR 2001, pp. 45–54, 2001.
12. M. Pesce, *Programming Microsoft DirectShow for Digital Video and Television*. Microsoft Press, 2003.
13. R. Carey and G. Bell. *The Annotated VRML 2.0 Reference Manual.* Addison-Wesley, 1997.
14. R. Lischner, *C++ in a Nutshell – A Language & Library Reference*. O'Reilly, 2003.
15. R. M. Taylor II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, A. T. Helser, *VRPN: A Device-Independent, Network-Transparent VR Peripheral System.* In Proc. VRST 2001, ACM, Baniff, Alberta, Canada, pp. 55-61.
16. VRCO. *Trackd.* http://www.vrco.com/trackd/Overviewtrackd.html, visited May 2006.
17. *W3C – World Wide Web Consortium,* http://www.w3.org/XML, visited May 2006.