TREC: Platform-Neutral Input for Mobile Augmented Reality Applications

Jason Kurczak and T.C. Nicholas Graham
School of Computing, Queens University
Kingston, Canada K7L 3N6
{kurczak, graham}@cs.queensu.ca

ABSTRACT
Development of Augmented Reality (AR) applications can be time consuming due to the effort required in accessing sensors for location and orientation tracking data. In this paper, we introduce the TREC framework, designed to handle sensor input and make AR development easier. It does this in three ways. First, TREC generates a high-level abstraction of user location and orientation, so that low-level sensor data need not be seen directly. TREC also automatically uses the best available sensors and fusion algorithms so that complex configuration is unnecessary. Finally, TREC enables extensions of the framework to add support for new devices or customized sensor fusion algorithms.

Author Keywords
augmented reality, tracking sensors, input framework, sensor fusion

ACM Classification Keywords
H.5.1 Information Interfaces and Presentation: Artificial, augmented and virtual realities

General Terms
Design

INTRODUCTION
Mobile Augmented Reality (mobile AR) is rapidly making inroads in the consumer market, with applications such as Car Finder [1] and Layar [2] being released for mobile phone platforms. These applications use the phone's camera and screen to superimpose information on a video feed of the real world, and have served to demonstrate the potential of mobile AR to the general public.

Mobile AR applications use sensors such as GPS, compass, and inertial sensors to determine the physical location and orientation of the device. The problem for mobile AR developers is that such sensors may be unreliable and noisy. Applications often must fuse inputs from multiple sensors to determine accurate values. Even when a framework is used to handle the input devices, it can require complex manual configuration and be hard to extend or modify. These problems require the developer to focus on low-level sensor programming rather than focusing on the functionality and usability of the application.

Figure 1. Using the tourist application

In this paper, we present an architecture for input frameworks designed to help address these issues, which is implemented in the TREC (TRacking using Extensible Components) framework. The advantages of this architecture are illustrated by its use in the Noisy Planet application described in the Motivating Example and Applying TREC sections.

The main advantages provided by TREC's architecture include giving programmers open access to a hierarchy of devices, transformative modules, and high-level abstracted interfaces; being able to dynamically select sensors and sensor fusion algorithms thanks to multiple levels of abstraction; and allowing modification and extension at every level.

MOTIVATING EXAMPLE: AN AR TOURIST APPLICATION
To provide context to our description of TREC, we introduce *Noisy Planet*, a mobile tourist guide. This application, shown in figures 1 and 2, has been implemented using TREC. Noisy Planet allows a tourist staying in an unfamiliar city to navigate to proximate destinations on foot while also allowing serendipitous exploration of the area.

EICS'11, June 13–16, 2011, Pisa, Italy.

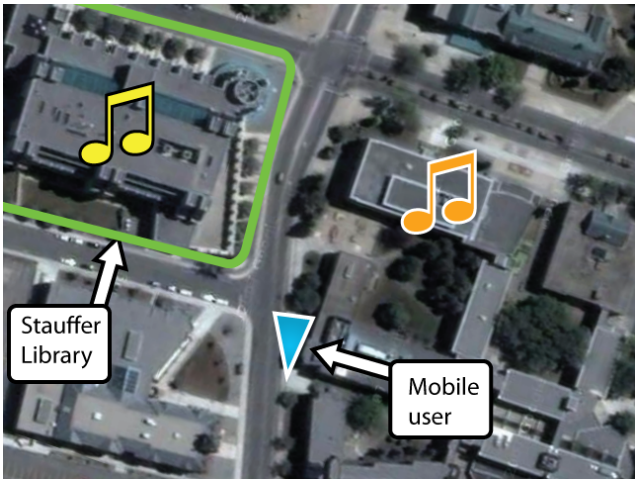

Figure 2. Overhead view

Noisy Planet uses 3D sound to convey to users the position and distance of nearby points of interest. For example, in figure 1, the user "hears" that the Stauffer Library is behind him and to his right. Each point of interest is represented by a subtle repeating tone – for example, the sound of riffling pages represents the library; clinking glasses represent a restaurant, and jingling coins represent a bank. The application overlays an aural landscape over the physical world. The sounds emanate from the correct location, even as the user walks or turns his head. Since the sounds are subtle and repeating, the tourist can easily choose to listen to them or tune them out.

The Noisy Planet implementation must track the user's location and head orientation so that sounds appear to emanate from the correct direction. It is challenging to accurately determine this information using off-the-shelf sensor equipment, and often multiple sensors are required to accurately estimate location and orientation. We currently use GPS, compass and gyroscope devices. Each device has its own data format, requiring programmers to write low low-level interfacing code. Each device type has limitations that cause it to deliver highly innacurate data under some circumstances. The programmer must therefore and must identify and code heuristics determining which device to use when they deliver conflicting information. As we shall see, our TREC framework helps with these difficulties.

INPUT HANDLING IN MOBILE AR APPLICATIONS

Before presenting the design of TREC, we first review current methods for obtaining input in mobile AR applications. We consider hardware support for location and orientation input, then review the state of the art in sensor fusion, and finally discuss existing input toolkits.

Hardware for Location and Pose Detection

A variety of off-the-shelf devices are available for estimating a user's location and pose in mobile contexts.

GPS devices triangulate their position using signals from orbiting satellites. Consumer devices have an accuracy of about 5-10m, depending on overhead visibility in outdoor environments [14].

Accelerometers can be used to track changes in position by calculating acceleration vectors based on the experienced forces, and integrating this data twice to obtain displacement [8]. Accelerometers provide faster updates and higher resolution updates than a GPS, but lack an absolute frame of reference and are subject to drift, therefore requiring periodic checks with some other absolute measure of position.

Other methods for tracking position include triangulation of signals from known locations, such as cellular tower signals, wifi signals or ultrasonic transmitter systems [6]. These approaches suffer from limited coverage.

Digital compasses (or magnetometers) detect orientation relative to magnetic north, but suffer serious drawbacks in accuracy. With the help of a 3-axis accelerometer to track the direction of earth's gravitational pull, a magnetometer can provide pitch, yaw, and roll data. Errors arise due to the lack of uniformity of the earth's magnetic field and its susceptibility to magnetic materials and artificial magnetic fields. In addition, outside forces (e.g., from walking) disturb the accelerometers' measurement of the gravitational vector and can cause large deviations in measured orientation when walking.

Gyroscopes measure angular displacement relative to some initial orientation, and so cannot indicate absolute direction on their own. However, unlike magnetometers, they are not affected by magnetic anomalies or by outside forces. Errors accumulated over time, however, will cause drift from the true direction. At high speeds of rotation (e.g., due to rapid head movement) some gyroscopes may exceed their upper limit of measurement and return wildly inaccurate results.

Sensor Fusion Techniques

Sensor fusion improves accuracy by combining data from multiple sensors [3]. A simple form of sensor fusion is to average the input of multiple sensors that are measuring the same property, in order to average out the noise from individual sensors (e.g. averaging the measurements of multiple anemometers to ascertain wind speed). More complex techniques take advantage of known properties of different sensor types. For example, a gyroscope, magnetometer, and accelerometer might be used in tandem, where the magnetometer is used to calibrate the gyroscope whenever the sensor is at rest.

Another approach is to use a Kalman filter [13], which takes multiple noisy sensor measurements, estimates the error in these measurements, and then estimates the actual state of the system being measured [4].

Programming fusion algorithms requires iterative tuning based on deep knowledge of the properties of the underlying sensors.

Frameworks

TREC is far from the first framework used to process input from tracking devices.

VRPN provides an interface between input hardware and Virtual Reality (VR) applications. VRPN allows VR hardware peripherals to be shared across many computers on the same network, and simplifies development by providing a standardized interface for peripherals with the same functionality [12]. VRPN standardizes the sensor data being delivered to applications so that their code is not dependent on the sensors being used. It does not, however, dynamically choose which of the attached devices to use; this must be specified by the application developer. In terms of extensibility, VRPN permits the creation of new devices and device types, while also supporting *layered devices* that let the developer program higher-level behaviour based on input from one or more sensors.

OpenTracker supports tracking hardware with a flexible and customizable architecture [10]. It uses dataflow graphs to manage data being passed from sensors to applications. Here, "device drivers" act as source nodes that bring data into the system; "filter nodes" transform, merge, or otherwise modify passed data from source nodes, and "sink nodes" output the data to an application. OpenTracker has a high level of configurability, using an XML schema to define the dataflow graph and supporting custom nodes. Like VRPN, however, OpenTracker does not offer automatic configuration and choice of devices, requiring the developer to provide a configuration file that describes the exact dataflow graph and devices to use.

Ubitrack, on the other hand, is designed for automatic configuration [9]. It is meant to support large networks of sensors to provide AR tracking using all available resources, with completely dynamic configuration of dataflow networks based on Spatial Relationship Graphs (SRGs) of the sensors in the network. There does not appear to be any way to modify or extend the algorithm used to configure the dataflow networks, or to override this algorithm to use specific devices or configurations.

The *OpenInterface Project* [11] has very interesting parallels in another area of HCI research. It is an open source platform for rapidly prototyping multimodal input interfaces for computer programs, with the benefit of a GUI interface. OpenInterface is designed to transform hardware input into a format suitable for the client application using modular transformation components, support a broad range of input devices, and be easily extensible. OpenInterface does not support the automatic selection of sensors and fusion algorithms, requiring explicit configurations by the developer like VRPN and OpenTracker.

All of these AR frameworks support abstraction of sensor data to standard interfaces. VRPN, OpenTracker, and OpenInterface are extensible, allowing new sensor types to be easily added. Ubitrack provides automatic sensor configuration. None of these toolkits addresses all three of these important goals. The TREC framework has been designed specifically to fill this gap.

THE TREC FRAMEWORK

TREC is a software framework for input handling in mobile AR applications. TREC's goals are to reduce the time required to develop interfaces for hardware sensors, and to reduce the difficulty of adapting AR apps to work with a particular collection of sensors. Specifically, TREC addresses the problems identified in the last section where: 1) writing low-level interfaces to sensors distracts developers from the core application design; 2) changing the set of available sensors may break applications, requiring extensive recoding; 3) combining the input from different sensors is tricky, requiring experimentation and iteration

First, in order to make the application programmer's job easier, TREC abstracts all sensor data into a high level representation of *location* and *orientation*. TREC provides the application programmer with simple interfaces for these two properties in its *abstract input layer*.

Second, TREC automatically configures the sensors by determining at runtime which of the connected sensors can be used to provide the best data to the application. Because the TREC layered architecture hides all differences between sensor hardware, it can provide the application plug-and-play compatibility with different sensors.

Finally, TREC uses sensor fusion algorithms automatically when a supported configuration is available, and the architecture makes it easy to extend the framework to support new algorithms. It uses a three-layer hierarchy (see figure 3) to abstract device details. This allows newly added sensors in the *device layer* to work automatically with existing applications, and fusion algorithms in the *abstract device layer* can take advantage of the abstracted devices automatically. The framework is open and allows access and modification at any level, meaning low-level sensor data can always be accessed directly if necessary.

In summary, the 3-layered architecture of TREC allows it to provide open access to high and low levels of abstraction, makes it easier to support automatic configuration based on a hierarchy of device types, allows sensor fusion algorithms to be dynamically chosen in the same way as devices, and makes new devices and algorithms easily interoperable with existing code.

The TREC Architecture

The framework is structured around a three-layer architecture: the Abstract Input Layer, the Abstract Device Layer, and the Device Layer.

The lowest layer, the device layer, contains objects that expose the data provided directly by device sensors. Devices must implement one one more abstract device interfaces (see below). For example, the OceanServer USB Compass provides both compass and accelerometer functionality. To add

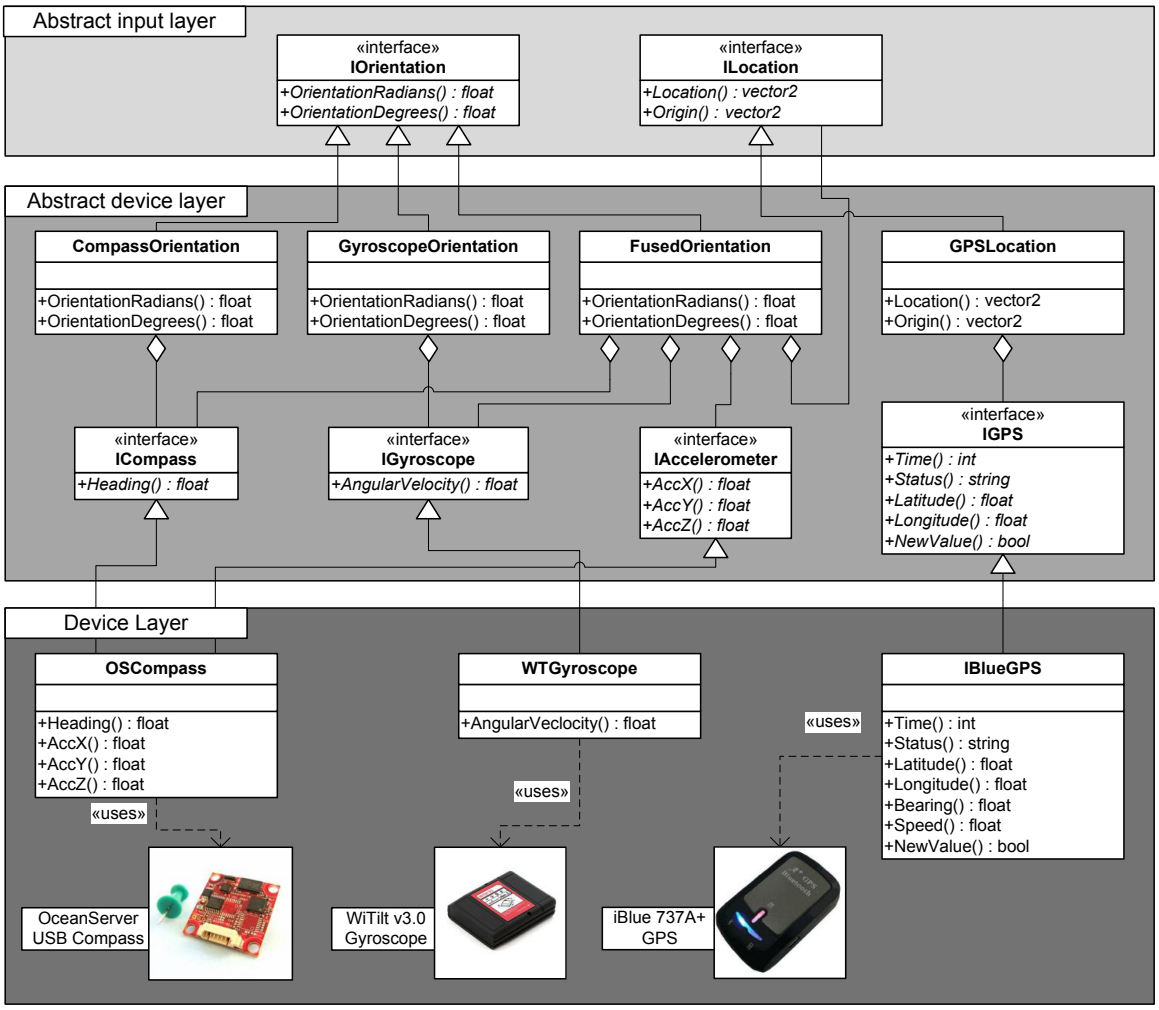

Figure 3. The TREC 3-layered architecture.

a new device to the framework, a developer needs to create a device-layer class for the device.

The abstract device layer groups standard types of sensor devices, and also provides virtual sensors by fusing the data from multiple concrete devices. For example, the *CompassOrientation* class provides orientation data from compass-like devices, such as OSCompass, while the *GyroscopeOrientation* class provides orientation data from gyroscope-like devices. Meanwhile, *FusedOrientation* provides orientation information by fusing data from a number of devices.

Finally, the abstract input layer provides interfaces for specific input data types. In the current version of the framework, two types of input are defined: *IOrientation* provides head orientation data and *ILocation* provides position information. To access input, an application queries the TREC device manager for one of these types of input, and receives an object in return implementing the appropriate abstract input type. In this way, the TREC framework shields applications from the details of individual devices or fusion algorithms, and simplifies the work of the developer by providing a simple and uniform way to access input information.

APPLYING TREC

We now show, by example, how TREC helps both application programmers creating a mobile AR system, and systems programmers adding new functionality to TREC itself.

The Application Programmer's Perspective

To illustrate how TREC helps in developing mobile AR applications, we examine the implementation of the Noisy Planet tourist application. Users of the application carry a mobile device containing a GPS, and wear a gyroscope and compass for head tracking. These sensors are hidden behind TREC's *IOrientation* and *ILocation* abstract input types.

To access the abstracted sensor data, the Noisy Planet code requests an appropriate data source from TREC's Device Manager. E.g., when an orientation data source is requested, the Device Manager provides an object that adheres to the *IOrientation* interface, with data coming from either the *OS-*

Compass or *WTGyroscope* device. The application cannot tell which device is supplying this data, and does not need to know since the incoming data is in a standardized format. This allows TREC to provide the best available sensor, and allows new devices to be added to TREC without impacting application code. If the programmer for some reason preferred a compass, he/she could choose to access the *CompassOrientation* object in the abstract device layer to guarantee the use of a compass, or even directly access the *OSCompass* in the device layer to choose that specific device.

In addition to automatically choosing from available sensors, TREC can automatically use sensor fusion algorithms when sufficient sensors are available. Noisy Planet could actually receive a *FusedOrientation* object (from the abstract device layer) after the earlier request for an orientation object. This happens without any changes or configuration on the application side. TREC decides which device or combination of devices to use based on an internal rating mechanism, which is currently a hard-coded list of the known devices.

For comparison, we implemented a version of Noisy Planet without access to TREC. This implementation required approximately 250-350 lines of code per sensor (GPS, compass, gyroscope) to handle serial input from these devices, in addition to another 80 lines to manage these sensors and perform the sensor fusion. The version using TREC requires just two method calls: one to get the current orientation, and another for the current location.

The Toolkit Developer's Perspective

TREC is an open framework, making it straightforward to add support for new devices. We now discuss our experience in adding a new hardware and a new virtual device to TREC.

Adding a New Hardware Device

The custom code that connects to and processes data from a sensor is contained within a class at the device layer. Based on the type of sensor, accessor methods need to be created that will adhere to the proper interface at the abstract device layer (e.g., the *IBlueGPS* in figure 3 must implement the *IGPS* interface in order to automatically work with TREC applications).

Adding a Simple Sensor Fusion Algorithm

TREC's layered architecture allows easy integration of sensor fusion algorithms into applications, and allows the fusion code itself to be shielded from the details of the underlying devices. Sensor fusion algorithms are written as abstract device modules that adhere to a standard interface. Therefore, any program taking advantage of TREC's abstract input modules (*IOrientation* and *ILocation*) can automatically use the new algorithm.

For example, to implement the *FusedOrientation* class of figure 3, a new abstract device class is created in the abstract device layer. This class implements the *IOrientation* interface, and therefore can be used whenever orientation information is requested by the application. Inside *FusedOrientation*, a gyroscope, compass and accelerometer are used (each of which are abstract devices), and some kind of location service is used (an abstract input.) Specifically, the gyroscope is used to determine orientation; the compass is used to calibrate the gyroscope from time to time, but not when the compass is moving (as determined via the accelerometer or from changes in location).

This example shows how the fusion algorithm leverages TREC's layered architecture. When necessary, the fusion algorithm knows the kind of device that it is using (gyroscope, compass, etc.), but does not need to know exactly which device is in use. Furthermore, when it is not necessary to know the kind of device (i.e., for the location service), the appropriate abstract input can be used.

DISCUSSION

We have shown how TREC helps developers by providing a single standardized view of sensor data, making sensor selection and sensor fusion automatic, and letting developers easily extend the framework to support new devices or custom fusion algorithms. We now discuss broader questions related to TREC's design.

Platform Standardization: Smartphone platforms are making sensors suitable for mobile AR applications broadly available. Exciting future work would involve customizing TREC around the sensor packages in common phones. The easy extensibility of TREC is important, as the sensors change between generations of phones, requiring updates to the framework. One limitation to this approach is that current phones do not provide sensors suitable for head-tracking.

Computer vision for tracking: Computer vision is widely used for pose detection in AR applications (e.g., using ARToolkit [7]), but we have not discussed its use in TREC. Vision is less immediately useful in mobile applications, due to the difficulty of placing fiduciary markers in the outside environment. However, there is no inherent reason why vision could not be included as a sensor type within the framework. This would integrate into the TREC hierarchy just as any other sensor does, though requiring a new abstract device interface to be defined for vision systems.

Extending to other kinds of AR: The framework as presented is heavily influenced by the ambient audio Noisy Planet application. In Noisy Planet, head tracking is important to overlay an audio soundscape onto the real world, and this head tracking need only be in a 2D plane. TREC can easily be extended to other forms of AR, however, simply by adding new device types.

For example, one common form of mobile AR involves tracking the position and orientation of a handheld device (e.g., a Smartphone), so that its display can provide visual overlays onto the real world. Here, positioning information is detected for the device, not the head, using the sensors in the device. TREC's *IOrientation* interface would need to be extended to provide 3D positions. This change would require additional programming, but does not represent a fundamental change to TREC's design.

Sensor management: Future work with TREC includes improving the sophistication of its sensor management capabilities. Currently, TREC's Device Manager determines which among the available set of sensors to use by traversing a ranked list of known devices. An improved device manager would automatically determine the best set of a sensors to use, and would dynamically reconfigure the sensor set (e.g., in response to failure of one of the sensors.) This issue of sensor management has been more thoroughly covered in the Ubitrack system [9] and in the hybrid tracking AR system of Hallaway et al. [5].

Sensor fusion: The layered architecture of TREC permits AR applications to use sensor fusion automatically. An example was shown in the last section of how a simple sensor fusion algorithm could be implemented as an abstract device. More investigation is required to assess the difficulty of implementing more general fusion algorithms, than the one presented in the example. The algorithm used there does not depend on specific devices and will work with any sensors of the same type. A Kalman filter, however, may require unique knowledge about each sensor to create a good model of the system. In future work, we hope to investigate whether TREC can support these advanced algorithms without restricting their use to predetermined hardware.

CONCLUSION

In this paper we have presented TREC, a new framework for handling sensor input in mobile augmented reality applications. TREC has been designed to provide high-level abstraction of the sensor data, automatic configuration, and ease of extension and manual configuration when desired. While there are many other frameworks addressing these issues comprehensively, none provide all of these features together in a cohesive package.

The key to TREC's success is its three-layer architecture. An abstract input layer provides high-level input types that completely abstract the underlying hardware. An abstract device layer collects classes of sensors (e.g., compass, accelerometer), as well as virtual devices implementing sensor fusion. Finally, a device layer provides interfaces to concrete devices. This architecture allows the abstraction of different hardware sensors into a uniform representation of location and orientation, and simplifies the automatic use of available sensors at runtime (including the automatic fusion of multiple devices). This hierarchy also makes the process of adding new devices straightforward, while providing compatibility with any applications already using TREC.

The next steps for the TREC framework include adding broader support for hardware sensors, investigating the implementation of complex sensor fusion algorithms using the abstract devices in TREC, and implementing a robust sensor-management algorithm to allow better dynamic configuration of the system.

ACKNOWLEDGEMENTS

This work was funded by NSERC Strategic Project #365040-08 and by the GRAND Network of Centres of Excellence. We would like to thank Claire Joly for her feedback on using TREC, and Jonathan Segel for his contributions to the design of Noisy Planet.

REFERENCES

1. Intridea Car Finder. http://carfinderapp.com/.
2. Layar Augmented Reality Browser. http://www.layar.com/.
3. B. Dasarathy. Sensor fusion potential exploitation-innovative architectures and illustrative applications. *Proceedings of the IEEE*, 85(1):24 –38, 1997.
4. E. Foxlin. Inertial head-tracker sensor fusion by a complementary separate-bias Kalman filter. In *IEEE VRAIS*, pages 185–194, 267, 1996.
5. D. Hallaway, S. Feiner, and T. Höllerer. Bridging the gaps: Hybrid tracking for adaptive mobile augmented reality. *Applied Artificial Intelligence*, 25:477–500, 2004.
6. J. Hightower and G. Borriello. Location systems for ubiquitous computing. *Computer*, 34(8):57–66, 2001.
7. H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana. Virtual object manipulation on a table-top AR environment. In *ISAR*, pages 111–119, 2000.
8. P. Lang, A. Kusej, A. Pinz, and G. Brasseur. Inertial tracking for mobile augmented reality. In *IMTC*, volume 2, pages 1583–1587, 2002.
9. D. Pustka, M. Huber, C. Waechter, F. Echtler, P. Keitler, J. Newman, D. Schmalstieg, and G. Klinker. Ubitrack: Automatic configuration of pervasive sensor networks for augmented reality. *IEEE Pervasive Computing*, preprint, June 2010.
10. G. Reitmayr and D. Schmalstieg. Opentracker: A flexible software design for three-dimensional interaction. *Virtual Reality*, 9:79–92, 2005.
11. M. Serrano, L. Nigay, J.-Y. L. Lawson, A. Ramsay, R. Murray-Smith, and S. Denef. The OpenInterface framework: a tool for multimodal interaction. In *CHI extended abstracts on human factors in computing systems*, pages 3501–3506, 2008.
12. R. M. Taylor, II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *VRST*, pages 55–61, 2001.
13. G. Welch and G. Bishop. An introduction to the Kalman filter. Technical Report TR 95-041, Department of Computer Science, University of North Carolina at Chapel Hill, 1995.
14. M. G. Wing. Consumer-grade global positioning system (GPS) accuracy and reliability. *Journal of Forestry*, 103:169–173, 2005.