Technical Report TUM-I0827
A Multitouch Software Architecture

Florian Echtler and *Gudrun Klinker*
Technische Universität München
Institut für Informatik
Boltzmannstr. 3
D-85747 Garching, Germany
{echtler,klinker}@in.tum.de

ABSTRACT
In recent years, a large amount of software for multitouch interfaces with various degrees of similarity has been written. In order to improve interoperability, we aim to identify the common traits of these systems and present a layered software architecture which abstracts these similarities by defining common interfaces between these layers. This provides developers with a unified view of the various types of multitouch hardware. Moreover, the layered architecture allows easy integration of existing software as well as swapping of components. Finally, we present our implementation of this architecture, consisting of hardware abstraction, calibration, event interpretation and widget layers.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Standardization

Keywords: multitouch, framework, architecture, widgets

INTRODUCTION
Research in multitouch interfaces has increased notably in the last years. For the most part, this is due to the emergence of multitouch hardware which can easily be built from off-the-shelf components. Consequently as more researchers have access to such hardware, the amount of software written to support these systems is growing speedily.

So far, however, most of this software is specific to the one single system it was written for. Therefore interoperability is almost non-existent at the moment. In this paper, we present our work to create an architecture which encompasses most of the common traits between these systems. Our goal is to provide two advantages over existing software - first, to enable developers to use a high-level API for the creation of multitouch-enabled software, second, to allow existing software to be used across hardware boundaries with the least change possible.

RELATED WORK
The significant amount of software which supports multitouch interfaces can roughly be divided into two groups, which are low-level input processing tools and high-level interaction software.

In the first group, an example is the system presented by Han [9], which has helped start the current wave of multitouch research. It describes a back-end software which provides image processing for the data from the FTIR (frustrated total internal reflection) screen and transmits the extracted touch spots to end applications.
One other widely used example is touchlib [18]. This library provides a simple configurable image capture and processing system which is geared towards blob extraction. This library is designed to be linked directly into applications. It also supplies a tool to transmit extracted data over the network to other applications.
Another well-known software is reacTIVision [12]. Its main focus is on tracking of fiducial markers, but it also provides support for finger touch points. Data is also transmitted over a network connection.
One example for the host of other systems which use their own internal image processing is the Soundscape Renderer [1]. A Java-based solution extracts touch points from the image and sends them to the application.
The protocol in which all these programs exchange touch data is the OSC-based TUIO [19, 14] format. Although TUIO has been designed with tracking of tangible objects in mind, it has become a de-facto standard for multitouch data. An updated version of the specification can be found in [13].

The second group of software aims to support higher-level interaction. Interestingly, many of these programs are based on the well-known DiamondTouch [3] interactive surface. One example is DiamondSpin [16], a Java-based toolkit which allows continuous rotation of windows and control from multiple touch points. Another variant is DTFlash [7], which focuses on adding multitouch support to the Macromedia Flash authoring suite. A .NET-based toolkit also exists [2]. One common trait of these systems is that they try to extend existing widgets or widget sets with multitouch support.
Other toolkits which support multitouch input through TUIO are vvvv [17], Processing [8] and others based thereon [15].

These focus on a easy, Java-like programming language or, in the case of vvvv, visual programming.
Another approach, which is not directly related to multitouch, but should be mentioned nevertheless, is followed by MPX [11] (Multi-Pointer X). This is an extension of the well-know X server environment to support dynamic generation and control of multiple pointers, which can then be used to control mouse-based applications.

A different aspect of higher-level interaction support is provided by software which tries to recognize gestures in the input stream as opposed to simply reacting to touch/release events. Several approaches based on DiamondTouch have been presented by Wu et al. [21, 20]. A common aspect of these systems is that gesture recognition is performed inside the application itself.
Finally, there are approaches to separate the recognition of gestures from the end-user part of the application [10, 5]. However, these systems are not yet beyond the design stage. In designing such a library of gestures, the work of Epps [6] et al., in which the most intuitive use of hands for a variety of common tasks was evaluated, should also be considered.
Despite this large body of work, there do not seem to be any efforts to combine them into a general architecture yet.

A MULTITOUCH SOFTWARE ARCHITECTURE

When looking at the significant body of related work in terms of multitouch software support, some similarities emerge. Many of these systems are split into an input and an application part, connected by a network link. There are several approaches to add multitouch support to existing toolkits and widget sets. Although approaches to generalize gesture recognition exist, most applications perform this internally in an ad-hoc manner.

Observing these common traits, we conclude that a generic multitouch framework should be able to provide a link between different input hardware on the one hand as well as different graphical toolkits on the other hand. Additionally, it should perform tasks which are independent of these two parts, such as calibration and gesture recognition.

Based on these observations, we shall now present the general layout of our framework. A high-level overview is given in Figure 1.

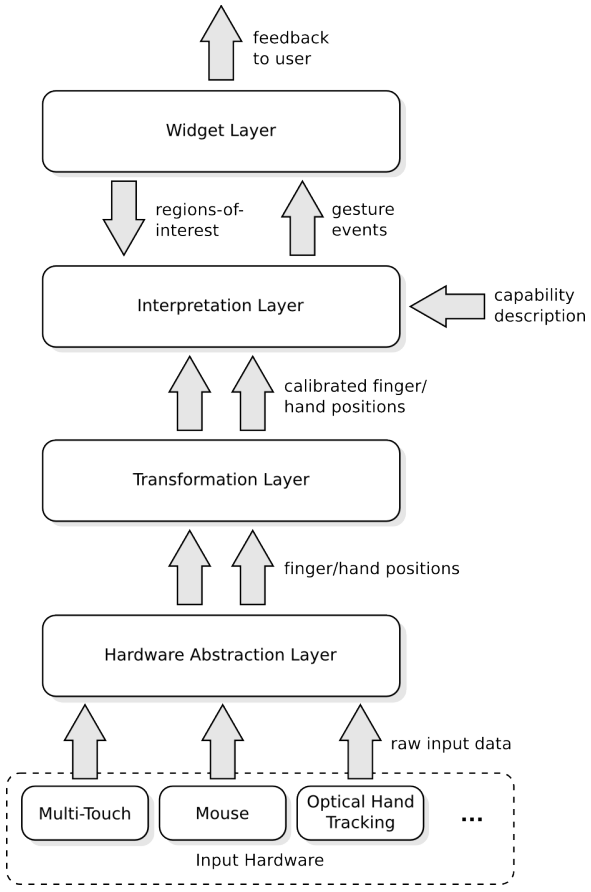

Figure 1: Architecture overview.

The lowest layer is formed by the *input hardware*, which generates raw tracking data in the form of, e.g., a video stream or electrical field measurements. This information is then processed by the *hardware abstraction layer*. Its task is to generate a stream of positions of fingers, hands and/or objects from the raw data, depending on the abilities of the hardware. As the positions are still in device (e.g., image) coordinates at this point, the next layer is the *transformation layer*, which transforms the position data from device to screen coordinates. This is achieved, e.g., with a perspective transformation which is obtained in a calibration procedure. Note that this does not exclude more complex setups, like 3D trackers and curved screens, as long as a suitable mapping into screen coordinates can be found.
At this point, the position data is ready for interpretation. The *interpretation layer* translates the movements of hands and fingers into gestures, thereby assigning a meaning to pure motion. To do so, this layer needs knowledge about regions on the screen. For each region, a list of gestures to match is maintained. When the correct events occur within a region, the corresponding gesture is triggered and passed to the next layer. As the mapping from motion to meaning can be expected to change for different input devices, a capability description has to be supplied which provides this mapping. This final part of our framework is the *widget layer*. Its task is to register and update regions of interest with the interpretation layer and then act on recognized gestures by generating visible output.

Hardware Abstraction Layer

This layer takes raw input data from the underlying hardware. The data is then searched for finger, hand and/or object positions, which are then transmitted to the next layer. To provide compatibility with the various programs mentioned above, this data should be transmitted in the TUIO format. This extends to non-optical input systems like DiamondTouch as well; as long as an adapter is available which provides compatible output. If no multitouch hardware is present, even ordinary computer mice could be used to generate compatible input data.

Transformation Layer

Especially with camera based systems, a transformation has to be performed on the low-level data, which is still in image or other device coordinates. Depending on the type of sensor used, a simple scaling or a perspective transformation may be necessary. In optical sensing systems which use a camera with a wide-angle lens, a radial undistortion step also has to be performed. All these calculations, as well as the prior estimation of the calibration parameters, are the task of the transformation layer. To account for existing tools which already contain a calibration system, this layer should be built as a filter for TUIO data that only needs to be inserted if the hardware abstraction layer does not already provide calibrated data.

Interpretation Layer

The interpretation layer receives calibrated TUIO packets from the lower layers and uses this motion data to generate gesture events for the next layer.

In this context, three different entities are important: regions, events and features.

- *Regions.* A region is a polygonal area, given in screen coordinates, in which a certain set of events will be matched. Regions whose extent will never change after initial registration can be flagged as *static* for optimization purposes. Regions can partially or totally occlude each other. They are therefore ordered from front to back, with the foremost regions having the highest priority. Regions can be seen as a generalization of the window concept which is used in all common GUIs.
- *Events.* An event is always registered for a region, and if specific conditions within that region are met, the associated event is triggered. An event can be flagged as *sticky*, meaning that it will continue to be active even if the actions which triggered it in the first place move outside the original region. The metaphor used here is that the event "sticks" to the user.
- *Features.* A feature is one single condition, of which several compose an event. In general, a feature is an easily obtainable, atomic property of user input, e.g. the number of touch points inside a region or the average distance between them.

These three entities are registered with the interpretation layer at the start of the application and triggered by the former or updated by the latter. An application first registers a region along with a unique identifier. A region is defined by a closed polygon, given in screen coordinates. Events can then be registered for this region. Using the identifier, the corresponding entity can also be updated or removed. Upon request from the interpretation layer, an application is required to send an update of the current region polygon, in case the corresponding UI element has moved. Along with *sticky events*, this prevents continuous updating of regions that would degrade performance. Instead, the interpretation layer only has to request an update when new input data arrives that is not yet assigned to a sticky event.

When an event is specified, a name from a list of predefined events can be used (e.g., "`move`", '`tap`" etc.). As the mapping from features to events is dependent on the capabilities of the hardware, a description file should be provided for each type of input hardware which describes the predefined events and their corresponding features. Additionally, it is possible to specify the list of features which comprise the event instead, along with a name. If the name is equal to one of the predefined names, the newly provided features will be used instead. Otherwise, a new gesture is registered and made available for the application to use. This allows applications to define new gestures and receive events that are not yet part of the capability description of the default events.

Widget Layer

As mentioned previously, the widget layer has the task of generating visible output for the user. It receives events from and registers regions with the interpretation layer. As both of these actions are of a very basic nature, this behavior should be easy to integrate with existing toolkits or widget sets. As regions are ordered, the widget layer just has to register a series of bounding polygons in the same sequence as the stacking order of the graphical widgets.

IMPLEMENTATION

So far, we only presented considerations of theoretical nature. To evaluate them in a practical context, we built an example implementation of all four layers. In our previous paper [4], we already have presented an example for the hardware abstraction and transformation layers. In addition to the capabilities of similar programs mentioned above, it offers the ability to track fingers, hands and objects simultaneously through use of a secondary light source. However, it should be emphasized that any tracking software which is able to provide TUIO data is to be usable with our framework. We have also built working prototype implementations for the interpretation and widget layers. All parts of our implementation are based on C++ and OpenGL. We have chosen these languages as they provide the best balance between performance, cross-platform availability and rich graphical capabilities.

As a test case, we are currently building several applications based on our framework. By analyzing their requirements, we are refining the key components of the architecture, especially the event specification. As the usability and success of any such framework depend on usage and feedback by developers, we are planning to distribute the code under an open-source license to a wider audience. It will soon be made available on Sourceforge.net (`http://tisch.sf.net/`).

In terms of interoperability with other software, special considerations apply with respect to MPX [11]. MPX, which offers the ability to control several mouse pointers at once, can be integrated into our framework in two ways. As a backend, an adapter which converts X pointer events into TUIO packets can be used to attach the two upper layers of our architecture to MPX. As a front-end, an adapter which works the other way round, converting TUIO data into pointer events, enables MPX to control legacy pointer-based applications with direct-touch hardware. We already developed a software which does the second type of conversion and will also build an adapter of the first type.

Finally, one important topic which should not be neglected is

latency. As all layers in the current implementation communicate with each other by means of UDP packets, network latency should not be underestimated. We have therefore measured the latency of the hardware-independent upper layers (interpretation and widget layer) by means of a mouse-driven interaction. A small tool captures timestamped mouse events and converts them into TUIO packets which are then inserted into the interpretation layer. A second timestamp is taken after the recognized event has been delivered to and processed by the widget layer. The difference then provides a rough estimate of the additional latency introduced by the two upper layers. We have taken 100 samples on a standard laptop computer running at 2 GHz, which resulted in an average measured latency of 2.35 ms with a standard deviation of 0.26 ms. This seems to be an acceptable additional latency, especially when taking the latency of the lower layers into account. For example, a camera-based sensing hardware, even if running at 60 Hz, already has an absolute minimum latency of 16.67 ms. Of course, any additional latency should therefore be kept as low as possible

CONCLUSION AND FUTURE WORK

We have presented a software architecture which aims to encompass the major common traits of existing multitouch software. To examine this architecture in a real-world context, we have built a framework based on this architecture and developed several applications as test cases. By distributing the code under an open-source license, we hope to foster use of our framework by other developers to gain valuable feedback. We are also examining which gestures might comprise a standard library that should be available per default.

Moreover, we are working on alternative implementations for the top and bottom layers. These include hardware abstraction layers for other kinds of input hardware, e.g., combined optical and acoustic tracking. We are also looking into the adaption of existing toolkits, like Qt, as a widget layer.

In conclusion, we believe that our architecture offers a comprehensive way to integrate different kinds of multitouch and direct-interaction devices as well as different toolkits into a well-structured framework.

REFERENCES

1. K. Bredies, N. Mann, J. Ahrens, M. Geier, S. Spors, and M. Nischt. The multi-touch soundscape renderer. In *Proc. AVI '08*, pages 466–469, 2008.

2. R.A. Diaz-Marino, E. Tse, and S. Greenberg. Programming for multiple touches and multiple users: A toolkit for the DiamondTouch hardware. In *UIST '03 companion proceedings*, 2003.

3. P. Dietz and D. Leigh. DiamondTouch: a multi-user touch technology. In *Proc. UIST '01*, pages 219–226, 2001.

4. F. Echtler, M. Huber, and G. Klinker. Shadow tracking on multi-touch tables. In *Proc. AVI '08*, pages 388–391, 2008.

5. J. Elias, W. Westerman, and M. Haggerty. Multi-touch gesture dictionary. United States Patent 20070177803, 2007.

6. Julien Epps, Serge Lichman, and Mike Wu. A study of hand shape use in tabletop gesture interaction. In *CHI '06 extended abstracts*, pages 748–753, 2006.

7. A. Esenther and K. Wittenburg. Multi-user multi-touch games on DiamondTouch with the DTFlash toolkit. In *Proc. INTETAIN '05*, 2005.

8. B. Fry and C. Reas. Processing. `http://processing.org/`.

9. J.Y. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *Proc. UIST '05*, pages 115–118, 2005.

10. X. Heng, S. Lao, H. Lee, and A.F. Smeaton. A touch interaction model for tabletops and PDAs. In *Proc. PPD '08*, 2008.

11. P. Hutterer. MPX - The Multi-Pointer X server. `http://wearables.unisa.edu.au/mpx/`.

12. M. Kaltenbrunner and R. Bencina. reacTIVision: a computer-vision framework for table-based tangible interaction. In *Proc. TEI '07*, pages 69–74, 2007.

13. M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. reactable TUIO Protocol Specification. `http://reactable.iua.upf.edu/?tuio`.

14. M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO: A protocol for table-top tangible user interfaces. In *Proceedings of Gesture Workshop 2005*, 2005.

15. H.-H. Lin and T.-W. Chang. A camera-based multi-touch interface builder for designers. In *Human-Computer Interaction. HCI Applications and Services*, 2007.

16. C. Shen, F.D. Vernier, C. Forlines, and M. Ringel. DiamondSpin: an extensible toolkit for around-the-table interaction. In *Proc. CHI '04*, pages 167–174, 2004.

17. vvvv Group. vvvv: a multipurpose toolkit. `http://vvvv.org/`.

18. White Noise Audio. Touchlib. `http://www.nuigroup.com/touchlib/`.

19. M. Wright. The Open Sound Control 1.0 Specification. `http://opensoundcontrol.org/spec-1_0`.

20. M. Wu and R. Balakrishnan. Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays. In *Proc. UIST '03*, pages 193–202, 2003.

21. M. Wu, C. Shen, K. Ryall, C. Forlines, and R. Balakrishnan. Gesture registration, relaxation, and reuse for multi-point direct-touch surfaces. In *Proc. TABLETOP '06*, pages 185–192, 2006.