Features, Regions, Gestures: Components of a Generic Gesture Recognition Engine

Florian Echtler, Gudrun Klinker
Technische Universität München
Institut für Informatik I16
{echtler|klinker}@in.tum.de

Andreas Butz
Ludwig-Maximilians-Universität München
LFE Medieninformatik
butz@ifi.lmu.de

ABSTRACT

In recent years, research in novel types of human-computer interaction, for example multi-touch or tangible interfaces, has increased considerably. Although a large number of innovative applications have already been written based on these new input methods, they often have significant deficiencies from a developer's point of view. Aspects such as configurability, portability and code reuse have been largely overlooked. A prime example for these problems is the topic of gesture recognition. Existing implementations are mostly tied to a certain hardware platform, tightly integrated into user interface libraries and monolithic, with hard coded gesture descriptions. Developers are therefore time and again forced to reimplement crucial application components.

To address these drawbacks, we propose a clean separation between user interface and gesture recognition. In this paper, we present a widely applicable, generic specification of gestures which enables the implementation of a hardware-independent standalone gesture recognition engine for multi-touch and tangible interaction. The goal is to allow the developer to focus on the user interface itself instead of on internal components of the application.

INTRODUCTION & RELATED WORK

More and more researchers and hobbyists have gained access to novel user interfaces in the last few years. Examples include tangible input devices or multitouch surfaces. Consequently, the number of applications being written for these systems is increasing steadily.

However, from a developer's point of view, most of these applications still have drawbacks. Core components such as gesture recognition are usually integrated so tightly with the rest of the application that they are nearly impossible to reuse in a different context. Additionally, most applications were designed to run on a single piece of hardware only. While this approach probably suited the original developers best, it impedes other persons who try to build on this work. In the end, many applications for novel interactive devices are consequently created from scratch, consuming valuable development time.

For example, countless applications contain code which tries to discover widely used multi-finger gestures for scaling and rotation, these being prime candidates for a more general approach. Of course, limiting any kind of generic gesture recognition to these few examples alone would not provide a great advantage, as many more gestures exist which the developer might want to include in an hypothetical interface.

Therefore, the first step in a general approach to gesture recognition is to design a formal, extensible specification of gestures. From an abstract point of view, the goal is to separate the *semantics* of a gesture (the intent of the user) from its *syntax* (the motions executed by the user).

While many graphical toolkits such as Qt, GTK+, Swing, Aqua or the Windows User Interface API exist today, all of them have originally been designed with common input devices such as mouse and keyboard in mind. To some extent, issues such as multi-point input, rotation independence or gesture recognition are being addressed in recent versions or extensions of these toolkits. Examples include DiamondSpin [8], the Microsoft Surface SDK or the support for multitouch input in Windows 7 and MacOS X.

Nevertheless, all these libraries still do not provide any separation between the syntax and semantics of gestures. When attempting to customize an application on a per-user basis or adapt it to a different type of hardware, this still requires significant changes to internal components of the library or application itself.

Some attempts have already been made with respect to recognizing 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. [11, 10]. A common aspect of these systems is that gesture recognition still is performed inside the application itself. Some preliminary approaches to separate the recognition of gestures from the end-user part of the application exist [6, 3]. With the exception of Sparsh-UI [5], these systems are not yet beyond the design stage. Sparsh-UI also follows a layered approach with a separate gesture server that is able to recognize some fixed gestures for rotation, scaling etc. independently from the end-user application. However, while being a step towards a more abstracted view of gestures, the crucial aspect of gesture customization has not yet been addressed.

A FORMAL SPECIFICATION OF GESTURES

Before discussing the details of our approach, some necessary prerequisites need to be described first. We assume that the raw input data which is generated by the input hardware has already been transformed into an abstract representation such as the popular TUIO protocol [7]. We also assume that the lo-

cation data delivered by this abstract protocol has been transformed into a common reference frame, e.g., screen coordinates. These assumptions should serve to hide any hardware-related differences from the gesture recognizer. Below, we will refer to data generated by the hardware as *input events*. Usually, every *input object* (e.g., hand, finger or tangible object) generates one input event for every frame of sensor data in which it is present. More details on the layered software architecture on which this approach is based can also be found in [2].

The formal specification which we will describe here forms the basis for a communications protocol. This protocol is used by the application to specify screen areas and the gestures which are to be recognized within them. Afterwards, the gesture recognizer will use the same protocol to notify the application when one of the previously specified gestures has been triggered by the users' motions.

Widgets and Event Handling

Before discussing the specification of gestures and events, we will briefly examine how widgets and events are handled in common mouse-based toolkits. Here, every widget which is part of the user interface corresponds to a *window*. While this term is mostly applied only to top-level application windows, every tiny widget is associated with a window ID. In this context, a window is simply a rectangular, axis-aligned area in screen coordinates which is able to receive events and which can be nested within another window. Due to this parent-child relationship between windows, they are usually stored in a tree.

Should a new mouse event occur at a specific location, this tree is traversed starting from the root window which usually spans the entire screen. Every window is checked whether it contains the event's location and whether its filters match the event's type. If both conditions are met, the check is repeated for the children of this window until the most deeply nested window is found which matches this event. The event is then delivered to the event handler of this window. This process is called *event capture*.

However, there are occasions where this window will not handle the event. One such occasion is, e.g., a round button. Events which are located inside the rectangular window, but outside the circular button area itself should have been delivered to the parent instead. In this case, the button's event handler will reject the event, thereby triggering a process called *event bubbling*. The event will now be successively delivered to all parent windows, starting with the direct parent, until one of them accepts and handles the event. Should the event reach the root of the tree without having been accepted by any window, it is discarded.

When we now compare this commonly used method to our approach, one fundamental difference is apparent. Instead of one single class of event, we are dealing with two semantically different kinds of events.

The first class is comprised of *input events* which describe raw

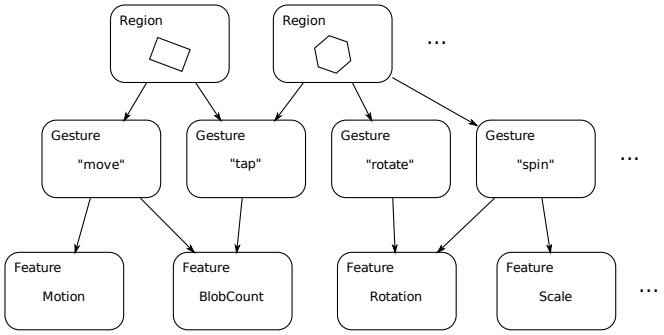


Figure 1. Relationship between regions, gestures and features

location data generated by the sensor hardware. These events are in fact quite similar to common mouse events. However, if we were to deliver these events directly to the widgets, no interpretation of gestures would have happened yet. The widget resp. the application front-end would have to analyze the raw motion data itself, which is exactly what our approach is trying to avoid.

In the gesture recognizer, these input events are therefore transformed into a second event class, the *gesture events* which are then delivered to the widgets. The existence of these two different event classes will influence some parts of the specification which will be discussed in the following section.

Before we arrive at this discussion, the following question should be asked: why are existing concepts such as widgets and events used in this approach instead of a radically new concept? This question is easily answered, as these existing concepts have the significant advantage of being widely used by a vast number of developers. As the primary goal of this approach is to make it easier for developers to build an interface based on novel input devices, building on established and widely known practices is the reasonable choice.

Abstract Description of Gestures

As no artificial restrictions should be imposed on the developer as to which gestures are available, a generic and broadly applicable way of describing them has to be found. To this end, the three abstract concepts of *features*, *regions* and *gestures* shall now be introduced. Their relationships are shown in figure 1.

From an abstract point of view, regions are polygons in screen coordinates. Each region corresponds to one GUI widget. A region can contain an arbitrary number of gestures which are only valid within the context of this region. Gestures can be shared between regions and are then valid in all containing regions. A gesture itself is composed of one or more features. Features are simple, atomic properties of the input objects and their motions which can in turn also be shared between gestures. Each of these features can be specified in more detail through constraints. Should all features of one gesture match their respective constraints, the gesture itself is triggered and delivered to its containing region.

At runtime, an application registers one or more regions with the gesture recognition engine. Every region has an unique

identifier and can contain several gestures. The gesture recognizer receives input events from the hardware and continuously tries to match them against the features in each gesture. When such a match succeeds, a gesture event containing information about the matching input events will be delivered back to the application.

Features

The basic building blocks of our formalism are *features*. Every feature is a single, atomic property of all input events that have been captured by a region. Examples for such properties are the average motion vector or the total number of input objects. A feature can appear in one of two variants: as a *feature template* when it is sent to the gesture recognizer and as a *feature match* when it is later sent back to the application. Both variants never appear as standalone entities, but only as components of a gesture.

By registering a gesture composed of one or more feature templates, the application specifies what properties the input events within the containing region must have in order to trigger this gesture. When these conditions are later met, the actual values of these properties are sent back within the gesture as feature matches.

A feature is described by a name, filters, optional constraint values and a result value. The name describes the specific kind of feature, i.e., which class is responsible for handling the feature calculation. The filters are similar to those already described for regions. For every type of input object, one filter is present. If this filter is set, input events of this type are incorporated into the feature calculation. While the filter settings on a region provide a first high-level selection that determines which input events are captured at all, the filters on each feature provide a more fine-granular control over which input events are actually used for calculating this specific feature. Note that two features within a single gesture can filter for different types of input objects each.

Depending on the class of feature, one or more constraint values can be given in a feature template that limit the value which the feature itself is allowed to take. For example, a feature with a single numerical result can have a lower and an upper boundary value as constraints. Note that the constraints always have the same type as the result value itself. After the value of a feature has been calculated, it is checked against the constraints values if they are present. Should the value of the feature fall within the specified range, the feature template changes into a feature match which has a valid result value.

Features can be divided into two groups: single-match and multi-match. Single-match features have a single result value for the entire region, such as the average motion vector. Multi-match features, on the other hand, can have several result values, usually up to one result per object inside the region. Why is this distinction necessary? As an example, consider a hypothetical user interface which should display a tile that can be moved by the user when touched and dragged. Additionally, every single touch location on the tile should be highlighted to provide additional visual feedback. For the motion information, a gesture that contains a single-match feature providing the average motion vector is sufficient. The individual motion vectors are not needed. However, for displaying the touch locations, the individual coordinates have to be delivered. The respective gesture has to contain a multi-match feature representing the object locations. Should this region be moved with, e.g. three fingers, every movement will trigger one motion gesture and three location gestures.

Conceptually, both types of features are used in exactly the same way; the only difference is that a gesture which is composed of multi-match features can be triggered several times by a single set of input events. Note that while mixing single- and multi-match features within a single gesture is possible, this composition will rarely be used, as only one single result will be produced.

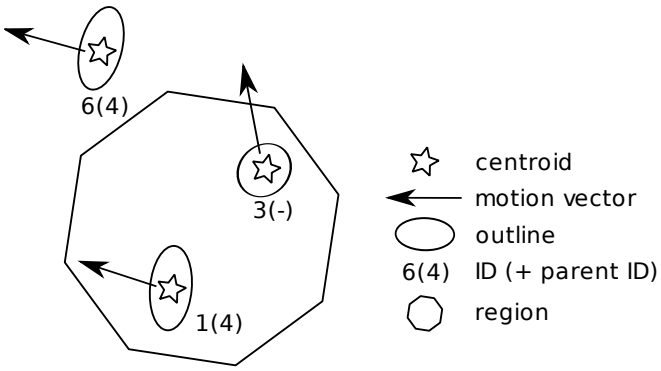


Figure 2. Sample input data for feature descriptions.

We will now briefly describe the currently available features. Their generated result values will be described at the example of figure 2. Note that only the two input objects within the octagonal region can contribute to feature results; the topmost input object moving outside the region will not be captured.

Single-Match Features

ObjectCount This feature counts the number of input events within the current region. E.g., if the appropriate filters for finger objects are set and the user touches the region with four fingers, this feature will have a result value of 4. A lower and upper boundary value can be set. In the example, the result value will be 2.

Motion This feature simply averages all motion data which has passed the filters and gives a relative motion vector as its result. Two constraint vectors can be specified which describe an inner and outer bounding box for the result vector. This can be used, e.g., to select only motions within a certain speed range or with a certain direction. In the example, the resulting relative motion vector will be the average of vectors 1 and 3 and point roughly to the upper left.

Rotation In this feature, the relative rotation of the input events with respect to their starting position is calculated. This feature itself is a superclass of two different kinds of subfeatures. The first subfeature, *MultiObjectRotation*, can only generate meaningful results with two or more input

objects and extracts the average relative rotation with respect to the centroid of all event locations. The second subfeature, *RelativeAxisRotation*, requires only one input object, but needs a sensor which is able to capture at least the axes of the equivalent ellipse of the object. The average relative rotation of the major axes of all input objects is extracted. In both cases, the result value is a relative rotation in rad which can again be constrained by two boundary values that form lower and upper limit. In the example, both variants will yield a result value close to zero, as neither object rotation nor relative rotation are occurring.

Scale Similar to *Rotation*, this feature calculates the relative change in size of the bounding box and has the corresponding scaling factor as a result. This feature also has two optional constraint values which serve as lower and upper bound. In the example, the result value will be larger than one, as the two input points within the region are moving apart.

Path With this feature, a complex path such as the outline of a letter can be recognized. The result is a value between 0 and 1 describing how well the predefined path matches the actual motion. This feature handles constraints slightly different than other features: it has an even number of constraint values which describe the predefined path as pairs of x/y values in the range of $[0; 1]$. The starting point of the path should be oriented at $0°$ relative to its centroid as described by Wobbrock et al. [9]. This feature can be used to implement shape-based gestures which cannot be reliably recognized by the more basic properties of the input events. Assuming a circular path, the result for the example data will be again close to zero, as little similarity between the straight paths and the constraint path exists.

Multi-Match Features

ObjectID The results of this feature are the IDs of all input objects within the region that have passed the filters. Two boundary values can again be specified to constrain the results to a smaller subset of IDs, e.g., to filter for specific tangible objects with previously known IDs. In the example, the two generated results will be "1" and "3", respectively.

ObjectParent This feature is similar to *ObjectID*, but returns the *parent ID* of each input object instead of the object IDs themselves. To receive both IDs for all objects, this feature can be paired with *ObjectID* in a single gesture. This particular feature requires the input hardware to detect a parent-child relationship between certain objects, e.g. between finger contacts and the whole hand. In the example, only a single result ("4") will be generated from object 1, as object 3 does not have a parent ID set.

ObjectPos The results of this feature are the positions vectors of all input objects. This feature currently does not have any additional constraints. In the example, the two results are simply the screen positions of input objects 1 and 3.

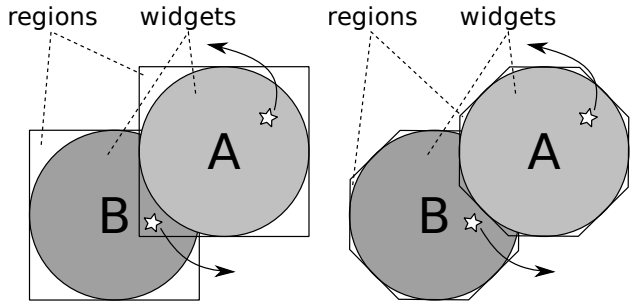
ObjectDim This feature has a special result type called *dimensions*. This is similar to the shape descriptor used in TUIO and gives an approximation for the outline and orientation of an object through its equivalent ellipse. Two optional *dimension* objects can be given as constraints, specifying upper and lower limits for each component of the shape descriptor. This filter allows to select, e.g., only blobs of a certain size and height/width ration. In the example, the two result values will describe the approximate shape and orientation of objects 1 and 3.

ObjectGroup This feature generates a match for each subset of input objects which can be grouped together in a circle of a specified radius. The result is a vector containing the centroid of one group. Two constraint values can be given, with the first component describing the minimum number of objects and the second component determining the radius of the circle. If the radius has been chosen large enough, the example will yield a single result which represents the average position of objects 1 and 3.

Regions

The primary task of regions is spatial filtering of input events. As it is the case with any regular GUI, a gesture-based interface can also be assumed to be divided into nested areas. In a mouse-based UI, these areas are called *windows* as described above. When moving to the presented, more general approach to user interfaces, this concept needs to be extended. For example, the fixed orientation and axis alignment is insufficient when considering table-top interfaces, e.g., a round coffee table.

Therefore, a region is defined as an area in screen coordinates which has a unique identifier and is described by a closed, non-intersecting polygon. Regions are managed in an ordered list, with the first region in the list being the topmost region on screen. This means that lower regions can be totally or partially obscured by those on top.



(a) with rectangular regions (b) with arbitrary regions

Figure 3. Overlapping widgets capturing input events

But why do regions need arbitrary shapes? Wouldn't a simple rectangle still be sufficient? The answer to these questions is more complicated that it seems at first glance. Consider two overlapping widgets as shown in figure 3(a). In a standard toolkit, the input event which was erroneously captured by widget A could simply be "bubbled" back to widget B. However, in the presented architecture, the input events are converted to gesture events before being delivered to the

widgets. The two input events would merge into one gesture event which cannot be split back into the original input events. Where should this single event now be directed to? The solution is therefore to ensure that input events are always assigned to the correct widget in the first place. The most straightforward way to achieve this goal is to allow regions of arbitrary shape which can closely match the shape of the corresponding widget as shown in 3(b).

Besides their arbitrary shape, regions can also further select input events based on their object type. The available object types depend on the sensor hardware and can comprise classes such as *finger*, *hand*, *tangible* and others. This behaviour is realized through a number of filters, one for each object type. When one of these filters is active, the region is sensitive to to input events from this object type. If the filter is disabled, the region is transparent to this type of input event. Several filters can be active at the same time.

At runtime, the input events described in the previous section are checked against all regions, starting from the top of the stack. When the object's centroid falls inside the region and the filter for the corresponding object type is active, this input event is captured by the region and stored for subsequent conversion into gesture events. Otherwise, regions further down are checked until a match is found. When no match occurs, the input event is finally discarded. Although the presented method deliberately uses a point-based approach to allow for a larger variety of input devices, an extension towards matching against outlines or shapes which are generated by optical sensors can be envisioned.

Gestures

The final and most central element of our formalism are *gestures*. An arbitrary number of gestures can be attached to every region. These gestures can either be created from scratch or taken from a list of predefined default gestures.

At runtime, these gestures can then be triggered by the input events which have been captured by the containing region. Should the conditions for one or more specific gestures match, an event describing the gesture is delivered to the containing region and therefore to the widget whose outline is described by the region. A gesture is composed of a unique name, a number of flags and one or more *features*. The name can either be an arbitrary descriptor chosen by the developer for custom gestures, or one of a list of predefined "common" gesture names. In the latter case, no features need to be specified, as these are part of the existing definition. Should the gesture contain one or more *feature templates*, it acts as a *gesture template* which describes an event to be triggered under certain conditions. Once these conditions have been met, a *gesture match* containing the corresponding *feature matches* is created based on the template.

Additionally, two flags can be set to further differentiate the behaviour of the gesture. When the gesture is marked as *one-shot*, it will only be sent once for a specific set of input objects. For example, consider a "press" gesture which is to be triggered when the user touches a region. The corresponding event should only be delivered once after the first input event has occurred, not subsequently while the user continues to touch the region. In this case, setting the one-shot flag will ensure the desired behaviour.

The gesture can also be marked as *default*. Should such a gesture be received, its name and features will be added to the list of standard gestures which can be accessed using only their name. This allows applications to register their own custom gestures for reuse among several widgets or to overwrite the definitions of the standard gestures given below.

Currently, 5 predefined standard gestures are available which have been selected based on the most common usage scenarios for interactive surfaces. A similar set of gestures has already been used in 1995 by Fitzmaurice et al. [4]. These gestures and their semantics are as follows:

press - triggered once when a new input object appears within the region

release - triggered once when all input objects have left the region

move - sent continuously while the user moves the region

rotate - sent continuously while the user rotates the region

scale - sent continuously while the user scales the region

Note that the actual features which comprise these gestures are not given here. The reason is that these features may differ significantly depending on the sensor. For example, on a camera-based touchscreen, rotation can be achieved by turning a single finger, whereas a capacitive sensor will require at least two fingers rotating relative to each other. However, this is irrelevant for the semantics of the resulting gesture - the intention of the user stays the same. Therefore, the composition of these default gestures can be redefined dynamically depending on the hardware used.

Examples

To give a better understanding of how these concepts work, the decomposition of some gestures into features shall now be discussed. The five standard gestures mentioned earlier can easily be mapped to a single feature each, e.g., the "release" gesture consists of an *ObjectCount* feature with both lower and upper constraint set to zero. As the *one-shot* property of the gesture is also set, this results in a single event as soon as the object count (e.g., finger contacts inside the region) reaches zero.

Another important mapping is that of the "move", "rotate" and "scale" gestures which contain a single *Motion*, *Rotation* and *Scale* feature, respectively. Note that a freely movable widget which uses all three gestures will behave exactly as expected, even though the raw motion data is split into three different entities. Consider, for example, rotating such a widget by keeping one finger fixed at one corner and moving the opposing corner with a second finger. In this case, the widget rotates around the fixed finger, thereby seemingly contradicting the definition of the *Rotation* feature which delivers rota-

tion data relative to the centroid of the input events. However, as the centroid of the input events itself also moves, the resulting motion events will modify the widget's location to arrive at the expected final position.

While a large number of interactions can already be modeled through single features and carefully selected constraints, combining several different features significantly extends the coverage of the "gesture space". For example, a user interface might provide a special gesture which is only triggered when the users quickly swipes five fingers across the screen. This can easily be described by the combination of an *ObjectCount* feature with a lower boundary of five and a *Motion* feature with a lower boundary equal to the desired minimum speed.

A different example would be to create gestures which can only be triggered by one specific tangible object. Of course, the input hardware has to be able to identify objects based on, e.g., fiducial markers. Should this be the case, any of the previously described gestures can be extended by adding one *ObjectID* feature which filters for the particular object ID.

Another powerful application for the conceptual split between gestures and features becomes apparent when considering input devices with different sensing capabilities. As mentioned earlier, optical touchscreens are usually able to detect the rotation of a single physical object on the surface. Therefore, the *RelativeAxisRotation* feature can be used in the "rotate" gesture to deliver relative rotation events. In contrast, a capacitive touchscreen will only be able to deliver simple location points without orientation, thereby requiring the use of at least two objects (usually fingers) to trigger rotation. In this case, the "rotate" gesture can now contain a *MultiBlobRotation* feature which will extract relative rotation data from two or more moving input points.

As both these features are derived from the common ancestor *Rotation*, this switch can be done completely transparent to the application, even at runtime.

SUMMARY & OUTLOOK

In this paper, we have presented a highly generic formalism for describing gestures to a recognition engine which analyzes raw motion data. The term "gesture" is used very loosely to describe any motion(s) by the user which are executed with a certain intent. Gestures can be attached to arbitrary screen regions which usually correspond to widgets in the user interface.

As each gesture is itself composed of one or more feature descriptors, the actual motions which trigger a certain event can be finely tuned. Particularly, predefined events can be adapted to a particular piece of input hardware without any changes to the application, as the semantics of the gesture remain unchanged - just the feature objects representing the syntax have to be modified.

We have implemented a recognition engine based on these concepts in pure C++ and have successfully used it with various user interfaces based on Java/Swing or C++/OpenGL. Due to its portable implementation, this gesture recognizer can be coupled with a wide variety of end-user applications, regardless of the environment they are written in. The recognition engine and the surrounding framework have been released as open source [1].

Of course, the value of this system is directly dependent on the number and flexibility of the features based on which gestures can be recognized. Should applications emerge where the existing features are insufficient, the current implementation will have to be extended. We will continue to create and evaluate various kinds of user interfaces with this approach.

REFERENCES

1. F. Echtler. libTISCH: Library for Tangible Interactive Surfaces for Collaboration between Humans. `http://tisch.sourceforge.net/`, accessed 2010-01-04.
2. F. Echtler and G. Klinker. A multitouch software architecture. In *Proc. NordiCHI '08*, pages 463–466, Oct. 2008.
3. J. Elias, W. Westerman, and M. Haggerty. Multi-touch gesture dictionary. United States Patent 20070177803, 2007.
4. G. Fitzmaurice, H. Ishii, and W. Buxton. Bricks: laying the foundations for graspable user interfaces. In *Proc. CHI '95*, pages 442–449. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1995.
5. S. Gilbert et al. SparshUI Toolkit. `http://code.google.com/p/sparsh-ui/`, accessed 2009-07-06.
6. X. Heng, S. Lao, H. Lee, and A. Smeaton. A touch interaction model for tabletops and PDAs. In *Proc. PPD '08*, 2008.
7. M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO: A protocol for table-top tangible user interfaces. In *Proc. Gesture Workshop '05*, 2005.
8. C. Shen, F. Vernier, C. Forlines, and M. Ringel. DiamondSpin: an extensible toolkit for around-the-table interaction. In *Proc. CHI '04*, pages 167–174, 2004.
9. J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proc. UIST '07*, pages 159–168, 2007.
10. 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.
11. 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.