Towards a Unified Gesture Description Language

Florian Echtler  
Munich Univ. of Applied Sciences  
Dept. of Computer Science  
florian.echtler@hm.edu

Gudrun Klinker  
Technische Universität München  
Dept. of Computer Science  
klinker@in.tum.de

Andreas Butz  
Ludwig-Maximilians-Universität  
Media Informatics Dept.  
butz@ifi.lmu.de

ABSTRACT

Proliferation of novel types of gesture-based user interfaces has led to considerable fragmentation, both in terms of program code and in terms of the gestures themselves. Consequently, it is difficult for developers to build on previous work, thereby consuming valuable development time. Moreover, the flexibility of the resulting user interface is limited, particularly in respect to users wishing to customize the interface. To address this problem, we present a generic and extensible formal language to describe gestures. This language is applicable to a wide variety of input devices, such as multi-touch surfaces, pen-based input, tangible objects and even free-hand gestures. It enables the development of a generic gesture recognition engine which can serve as a backend to a wide variety of user interfaces. Moreover, rapid customization of the interface becomes possible by simply swapping gesture definitions - an aspect which has considerable advantages when conducting UI research or porting an existing application to a new type of input device. Developers will be able to benefit from the reduced amount of code, while users will be able to benefit from the increased flexibility through customization afforded by this approach.

Categories and Subject Descriptors

H5.2 [Information interfaces and presentation]: User Interfaces—Graphical user interfaces.

Keywords

gestures, recognition, classification, formal specification

1. INTRODUCTION

In the last few years, gesture-based human-computer interfaces have become quite common. Examples include multi-touch UIs, pen-based input, tangible interfaces, gestures executed in free space and many more. Consequently, the number of applications being written for these systems is increasing steadily. However, this proliferation of novel types of user interfaces has resulted in considerable fragmentation, both in terms of the gestures themselves and in terms of the code used to recognize them.

As a result, most of these applications still have drawbacks from the point of view of a designer or developer. 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. In the end, many applications for novel interactive devices are consequently created from scratch, consuming valuable development time.

Another disadvantage of this monolithic approach concerns users who wish to customize an existing application. Such customization may manifest itself as the modification of some gestures to suit personal preferences, or the attempt to run an application with a different type of input sensor, e.g., using tangible objects instead of multi-touch. Unfortunately, the tight integration of gesture recognition and application is a significant obstacle to such modifications.

Our approach to these problems is the design of a formal, machine-readable language to describe a wide variety of gestures - the Gesture Definition Language (GDL). The benefits of this language are twofold. On the one hand, developers do not have to write yet another piece of gesture recognition code - they can simply describe their gestures in GDL and have them matched by the generic recognition backend. On the other hand, users are able to modify the gesture definitions to suit their own preferences or to adapt the application to a different environment.

From an abstract point of view, the goal of GDL is to separate the semantics of a gesture (the intent of the user) from its syntax (the motions executed by the user). A particular advantage of this approach is evident when considering a usage scenario such as rapid prototyping of a novel user interface. The ability to quickly modify UI elements and related gestures independently from each other provides considerable additional flexibility when compared to other, more monolithic approaches.

2. RELATED WORK

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 [13], the Microsoft Surface SDK or the support for multitouch input in Windows 7 and MacOS X.

In contrast, a number of standalone toolkits dedicated solely to development for novel user interfaces have also been developed in the last decade. Some examples for this kind of framework are PyMT [9], Papier-Mâché [12], Phidgets [7] or libavg [14].

Nevertheless, all these libraries still do not provide any separation between the syntax and the semantics of a gesture. 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. As many...
of these software packages are not available as source code, such changes may even be impossible.

One attempt to formally describe interactions within the context of sketch-based user interfaces has been made with LADDY by Hammond et al. [8]. Other designs which separate the recognition of gestures from the end-user part of the application have been presented in [10, 4]. With respect to gesture recognition, two systems which have already progressed past the design stage include Sparsh-UI [6] and UTouch [1]. Both follow 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 here.

3. 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 [11]. We also assume that the location data delivered by this abstract protocol has been transformed into a common reference frame, e.g. screen coordinates in pixels or 3D coordinates in meters relative to an arbitrary origin. These assumptions should serve to hide any purely 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, pen, finger or tangible object) generates one input event for every frame of sensor data in which it is present. Note that there may be various types of input events when the sensor setup is able to detect several fingers or tangible objects. More details about the layered software architecture on which this approach is based can also be found in [3].

3.1 Widgets and Event Handling

Before discussing the specification of gestures and events, we will briefly examine how events are handled in common mouse-based toolkits. In such a toolkit, the user interface is mostly composed of widgets which are small, self-contained UI elements such as buttons, sliders, textboxes etc. In most cases, every widget which is part of the user interface corresponds to a window. While this term is usually 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. Such a situation occurs, e.g., when using 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.

Figure 1: Relationship between regions, gestures and features

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 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.

3.2 Abstract Description of Gestures

As no artificial restrictions should be imposed 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 spatial areas given in reference coordinates. A region usually corresponds to one GUI element. Regions are ordered according to precedence. 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.
3.2.1 Features

The basic building blocks of our formalism are features. Every feature is a single, atomic property of all input events within a certain 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.

\[ \text{feature ::= name filters [constraints] result} \]

The name describes the specific kind of feature, i.e., how the feature calculation is performed (see below). The filters are a bitmask which describes what kind of input object this feature is sensitive to. For every type of input object, one filter bit is present. If this filter is enabled, input events of this type are incorporated into the feature calculation. Note that two features within a single gesture can filter for different types of input objects each.

Depending on the type 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 to a feature match which has a valid result value and is sent back to the application.

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. 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 event and three location events.

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.

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.

---

**Figure 2:** Sample input data for feature descriptions.

### Single-Match Features

#### 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 a lower and upper boundary for each component of 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.

#### 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 1.0, 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 number of constraint triplets which describe the predefined path as $x/y/z$ values in the range of $[0; 1]$. The starting point of the path should be oriented at $0^\circ$ relative to its centroid as described by Wobbrock et al. [15]. 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 template, the result for the example data will be close to zero, as little similarity between the straight paths and the constraint path exists.
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 fin-
gers, 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.

ObjectDelay
This feature represents the number of frames for which
input events have been available within the region and can
again be filtered through a lower and upper boundary value.
This allows the description of gestures such as a brief tap or
press-and-hold for a minimum duration.

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 bound-
ary values can again be specified to constrain the results to a
smaller subset of IDs, e.g., to filter for specific tangible ob-
jects 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 them-
selves. To receive both IDs for all objects, this feature can be
paired with ObjectID in a single gesture. This particular fea-
ture requires the input hardware to detect a parent-child re-
lationship between certain objects, e.g., between finger con-
tacts 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 positions of input objects 1 and 3 in reference
coordinates.

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 ob-
jects can be given as constraints, specifying upper and lower
limits for each component of the shape descriptor. This fil-
ter 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 ob-
jects 1 and 3.

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

3.2.2 Regions

The primary tasks of regions are spatial filtering of input events
and assignment to different gesture sets. As it is the case with any
regular GUI, a gesture-based interface can also be assumed to be
divided into partially overlapping spatial areas. In a mouse-based
UI, these areas are called windows as described above. When mov-
ing 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 inter-
faces, e.g., a round coffee table.

Therefore, a region is defined as an area in reference coordinates
which has a unique identifier and is described by a list of coordinate
triplets.

region ::= id flags [coords ...] [gesture ...]

The id is a unique string which identifies the region. The flags are
similar to the filters described in section 3.2.1, with one additional
flag which determines the interpretation of the following list of co-
ordinates. This list is either interpreted as a closed polygon which
forms a two-dimensional region or as a point cloud which repre-
sents a three-dimensional region by its convex hull. Regions are
managed in an ordered list, with the first region in the list being
the region with the highest priority. This means that regions further
down in the list can be totally or partially obscured by those on top.
Finally, a list of gestures describes the events which this region is
sensitive to.

Figure 3: Overlapping regions 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
regions as shown in a 2D example in figure 3(a). In a standard
toolkit, the input event which was erroneously captured by wid-
get 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 di-
rected 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 arbi-
trary 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 in-
put 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. As with features, 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 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 list. 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. Al-
though the presented method deliberately uses a point-based ap-
proach 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.

3.2.3 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 de-
scribing the gesture is delivered to the containing region and there-
fore to the widget whose outline is described by the region. A ges-
ture is composed of a unique name, a number of flags and one or
more features.

gesture ::= name flags [feature ...]
The name can either be an arbitrary descriptor chosen by the devel-
oper 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 con-
tain 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, three flags can be set to further differentiate the be-
haviour 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 ex-
ample, 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 subse-
quently 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 stan-
dard 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 stan-
dard gestures given below.

Finally, the gesture can be flagged as sticky, meaning that such
a gesture captures all input events which triggered its execution
so that they are unable to trigger other gestures while the original
one continues. This allows gestures to continue, e.g., across region
boundaries instead of stopping abruptly when the input events leave
the original region.

Currently, 6 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. [5]. These gestures and their seman-
tics are as follows:

press - triggered once when a new input object appears within the
region
remove - triggered once when an input object is removed from 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 signifi-
cantly 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 rotat-
ing relative to each other. However, this is irrelevant for the se-
manetics 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.

3.3 Examples

To give a better understanding of how these concepts work, the
decomposition of some gestures into features shall now be discus-
sed. 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. In GDL, this gesture looks
as follows:

release oneshot,default ObjectCount 255 0 0 0

The name of the gesture and the flag values are followed by the
single feature descriptor. The feature also starts with the feature’s
name, followed by a filter bitmask representing all types of input
objects, two constraint values (both 0) and a placeholder for the
result value.

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
rotation 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 ex-
pected final position.

While a large number of interactions can already be modeled
through single features and carefully selected constraints, combin-
sing 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 two fingers horizontally across the screen. This can easily
be described by the combination of an ObjectCount feature with a
lower boundary of two and a Motion feature with a lower boundary
equal to the desired minimum speed.

swipe
ObjectCount 255 2 2 0
Motion 255
100 0 0 // lower boundaries
1000 10 10 // upper boundaries
0 0 0 // result (empty)

This description of the "swipe" gesture has no specific flags and is
composed of the two features mentioned above. Both features
filter for any type of input object. The constraints of the Motion feature specify a lower and upper bound for each component of the resulting motion vector; here, the values are chosen so that only fast movements which are predominantly along the x-axis match the constraints.

Of course, a different kind of user interface might not be well suited for a horizontal swipe. When considering, e.g., a circular tabletop display, it may be quite unclear to the user which direction is meant by horizontal. In this case, the definition of the "swipe" might simply be modified by replacing the Motion feature with one matching, e.g., a circular path, thereby creating an orientation-independent analogy to the horizontal swipe.

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 Multi-BlobRotation 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.

4. 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 spatial regions which usually correspond to widgets or, more generally, sensitive elements 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, either by the developer or by the end-users themselves.

We have implemented a recognition engine based on these concepts in pure C++ and have successfully used it with various user interfaces developed in a variety of languages such as C++, C# and Java. 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 [2].

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 therefore continue to create and evaluate varied user interfaces with our generic gesture recognition engine.

Another future extension which may require modification of the presented approach is support for shape-based interaction. Optical sensors in particular are able to capture very rich information, e.g. about hand postures, which can not easily be subsumed by an approximation such as an equivalent ellipse. To process and react to such data, further investigation into extended formal gesture descriptions may be necessary.

5. REFERENCES