GISpL: Gestures Made Easy

Florian Echtler
Munich Univ. of Applied Sciences /
Siemens Corporate Technology
florian.echtler@siemens.com

Andreas Butz
Ludwig-Maximilians-Universität München
Institut für Informatik
butz@ifi.lmu.de

Figure 1: Controlling a widget using alternative input methods

ABSTRACT

We present GISpL, the Gestural Interface Specification Language. GISpL is a formal language which allows both researchers and developers to unambiguously describe the behavior of a wide range of gestural interfaces using a simple JSON-based syntax. GISpL supports a multitude of input modalities, including multi-touch, digital pens, multiple regular mice, tangible interfaces or mid-air gestures.

GISpL introduces a novel view on gestural interfaces from a software-engineering perspective. By using GISpL, developers can avoid tedious tasks such as reimplementing the same gesture recognition algorithms over and over again. Researchers benefit from the ability to quickly reconfigure prototypes of gestural UIs on-the-fly, possibly even in the middle of an expert review.

In this paper, we present a brief overview of GISpL as well as some usage examples of our reference implementation. We demonstrate its capabilities by the example of a multi-channel audio mixer application being used with several different input modalities. Moreover, we present exemplary GISpL descriptions of other gestural interfaces and conclude by discussing its potential applications and future development.

Copyright © 2012 by the Association for Computing Machinery, Inc.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions Dept, ACM Inc., fax +1 (212) 869-0481 or e-mail permissions@acm.org.
TEI 2012, Kingston, Ontario, Canada, February 19 – 22, 2012.
© 2012 ACM 978-1-4503-1174-8/12/0002 $10.00

ACM Classification Keywords

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

General Terms

Design, Standardization, Human Factors

Author Keywords

interaction, user interface, formal language, gestures

INTRODUCTION

Gestural human-computer interfaces have already been presented decades ago, with one of the earliest examples being the use of a ”flick” gesture in SketchPad [21] from 1964. Despite these early beginnings, it is only during the last decade that we have seen a notable increase in the width and depth of gestural interfaces.

In the context of this article, we use ”gesture” in the widest possible sense, meaning any kind of motion-based command directed from the user to the interface. Examples for such ”gestural” interfaces include multitouch surfaces, pen-based UIs, tangible interfaces or those based on free-air hand gestures. What unites these types of UIs from a highly abstract point of view is that they detect a specific subset of the motions executed by one or more users and react to them.

When approaching the development, design and usage of such systems from this generic viewpoint, the primary question is how to describe these motions to the UI system in an unambiguous, machine-readable way. To this end, we present GISpL, the Gestural Interface Specification Language. GISpL is used to describe the expected motions of the user(s) to a generic gesture recognition engine. Designing a gestural UI using GISpL offers advantages to several distinct groups of users:

Developers, for example, can avoid tasks such as the tedious reimplementation of algorithms for custom gesture recognition. In addition, the effort needed to port an application to a different type of input device is reduced considerably. Existing frameworks for the development of gestural interfaces mostly recognize only a fixed set of gestures and are designed for a single type of input device.

User interface researchers can also benefit from using GISpL. In a setting such as an expert review of a prototypical UI, the option to quickly reconfigure an interface based on GISpL can help to immediately integrate the reviewer's suggestions and receive additional feedback on the changes. Additionally, GISpL descriptions are much easier to compare across different UIs and might even offer a way to present gestures in research publications.

Finally, end users can take advantage of a GISpL-based interface. Expert users in particular can customize the interface according to their own preferences. This is not possible with most existing gestural UIs. Even those users who do not wish to apply any customization benefit from the consistent behavior across different devices resulting from the use of a common gesture recognition engine.

RELATED WORK

There exists a wealth of frameworks for the development of user interfaces for novel input devices. Examples include DiamondSpin [20], the Microsoft Surface SDK [16], PyMT [8] and MT4j [15] for multi-touch devices or Papier-Mâché [13] and Phidgets [6] for tangible input. However, all of these software packages share two limitations: a) they have been designed with a single class of input hardware in mind and b) the behaviour of the UI elements is governed by hardcoded rules.

Some approaches have been presented which aim to introduce a cleaner separation between the movements executed by the user and the actions taken by the user interface, i.e. between the detection of movement and its interpretation.

In this context, the topic of multi-touch input has been a particular focus of attention. Scholliers et al. [18] present a declarative language for describing multitouch gestures, similar to an UML-based language by Khandkar et al. [12]. Kammer et al. [11] discuss general aspects of the formalization of such gestures. Wobbrock et al. [23] introduce a system where users are able to define gestures for certain actions themselves, thereby allowing customization to a limited degree.

LADDER by Hammond [7] aims to provide a generalized framework for sketching diagrams in specific task domains. Squidy by König et al. [14] provides a configurable hardware abstraction layer which generates data in the common TUIO format [10] from a variety of input modalities. Dragicevic et al. [3] present the ICON toolkit which can be adapted to arbitrary input devices using a graphical editor. Serrano et al. [19] introduce the OpenInterface framework, following a similar approach with a dataflow editor used to connect components for transforming input data. While these two systems already attempt to generalize input across different types of input devices, they do so in an informal way, relying on custom components in the dataflow graph.

An even more generic position is taken by the Human Markup Language [1] which tries to capture all aspects of interpersonal communication, including gestures, in an XML-based formalism.

When reviewing this existing body of work, it becomes apparent that there is a significant interest in a more generic approach towards gestural user interfaces. GISpL aims to provide this generic approach in a formal and fully device-independent way, building on our previous architectural work presented in [4, 5].

EXAMPLE USAGE SCENARIOS

To better motivate the development of GISpL, we will first present two example usage scenarios.

Object Transformations

A common demonstration of the capabilities of multi-touch interfaces is to present a set of objects, usually images, which can be translated, rotated and scaled using two or more fingers.In most cases, an affine transformation matrix is calculated from the touch points and applied to the object. When using GISpL to represent such an interface, a different approach is possible. The transformations executed by the user result in a stream composed of separate move, scale and rotate events. The developer can simply choose to ignore some classes of events to create interface objects which exhibit only a subset of behaviors without having to deconstruct and modify any transformation matrices. Moreover, each transformation class can be mapped to a different type of input. For example, movement of items could be mapped to touch events while rotation is controlled by means of a tangible object without any changes to the application.

Input-Agnostic Audio Mixer

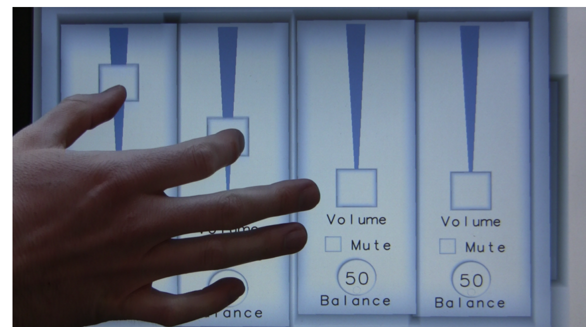

Figure 2. Using a multi-channel audio mixer application with multi-touch input.

In this scenario, we will consider a multi-channel audio mixer application which can be used with a variety of novel interaction devices as well as with a regular desktop interface (see Figures 1 and 2).

Desktop Interface. All elements of the interface can be controlled using drag-and-drop operations or the mouse wheel. On desktops systems such as X11 v7.5 which support the concurrent use of several mice, multiple users can interact with the interface simultaneously.

Multi-Touch Table. On a horizontal multi-touch device, each mixer block can be separately transformed and translated to support several users on various sides of the table. Dials can be rotated using single- and multi-touch gestures.

Tangible Input. Dials and sliders can be moved and rotated using physical handles.

Using GISpL enables the developer to decouple the actions possible on each UI element from the motions needed to trigger these actions. Dial widgets, for example, always react to *rotate* events, regardless of how these events have been triggered. For most types of input devices, an intuitive mapping from motions to actions can be found. A collection of presets can be provided by the developer, giving the option of later modification by the user.

CONCEPTUAL MODEL OF GISPL

GISpL offers a means to formally describe a large variety of user motions which may occur in novel user interfaces. To this end, it uses the two central concepts of *gestures* and *features*. Gestures correspond to events triggered by the users' actions and are composed of one or more features which are simple mathematical properties of the raw input data. This input data is generated by one or more classes of *input objects*, i.e. physical entities which can be detected by the respective input hardware. The hardware generates a stream of *input events* which are used to calculate feature values. In most cases, a set of temporally concurrent input events will be delivered as a group in one *input frame*.

As a canonical way of delivering input events, we have selected the widely used TUIO protocol [9] which is supported by the majority of NUI input devices. TUIO 2.0 conveniently defines a range of input object classes which cover a wide range of input devices such as touchscreens, tangible objects, pens or free-hand input. An input object is identified by a unique numeric ID for its lifetime. For a touch point on a multi-touch screen, this may be the duration of the contact between screen and finger, while a tagged tangible object may exhibit the same identifier over and over, even if it is temporarily out of range. GISpL itself does not deal with the generation of TUIO events from raw sensor data, but relies on third-party software such as, e.g., CCV [17] for this task. Consequently, it is also the responsibility of the external software to properly convert sensor data into a suitable TUIO representation on which GISpL then acts.

Input events are spatially filtered by a list of *regions* (i.e., closed polygons) which are provided by the application. Every region contains an independent set of gestures. The need for such spatial filtering becomes apparent when comparing the behavior of the various UI elements in our second scenario. For example, a single touch point moving over a slider widget will result in *move* events representing a linear offset, whereas the same motion tangential to a dial widget should generate *rotate* events representing an angular offset. This spatially dependent interpretation of similar motions is achieved by means of separate regions. The various regions comprising our example application and their spatial ordering are illustrated in Figure 3. Incoming events are checked against an ordered list of regions, starting from the top. When an input event falls within a region, the event is captured by that region and checking is terminated. For more complex multi-modal UIs, specific regions can be selectively made "transparent" to input objects of certain types, allowing those to be captured by lower regions.

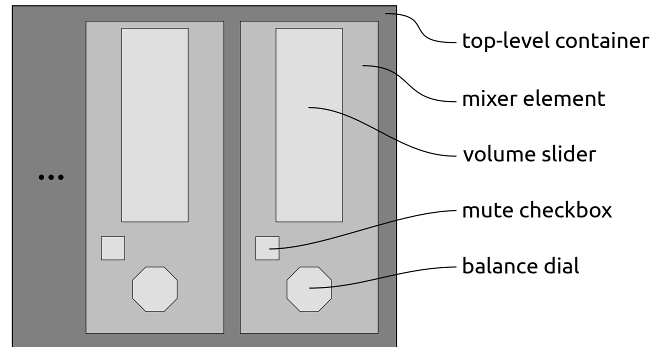

Figure 3. Layered regions in the audio mixer application. Lower regions are darker.

An overview of the entire GISpL processing pipeline is shown in Figure 4.

GISpL Example

In order to create a machine-readable description of the concepts described above, these have to be expressed in a formal language. The design of this language was based on three goals: easy to parse, easily readable by humans and compact. JSON [2] offers a good balance of these goals and consequently forms the basis for GISpL. Its usage is best presented by means of an example, describing a horizontal left-to-right swipe with two fingers (see Figure 5).

The outermost object represents the whole gesture with its name, a number of optional flags and the list of features. Each feature in turn consists of a feature class, a filter bitmask[1] for the types of input objects and a list of constraints which depend on the feature in question. For both features, the filter value of 2046 represents a bitmask (11111111110) matching all object classes corresponding to fingers. A list of GISpL object classes (a superset of those defined by TUIO 2.0) is given in table 1.

In the case of the *Count* feature which represents the number of input objects that are currently present, the constraints describe a lower and upper boundary within which the value must fall. The same applies to the *Motion* feature which represents the overall motion vector of the input objects. However, since this is a vector-valued feature, the boundary val-

[1]For better readability, named constants would be preferable, however, JSON does not offer this feature.

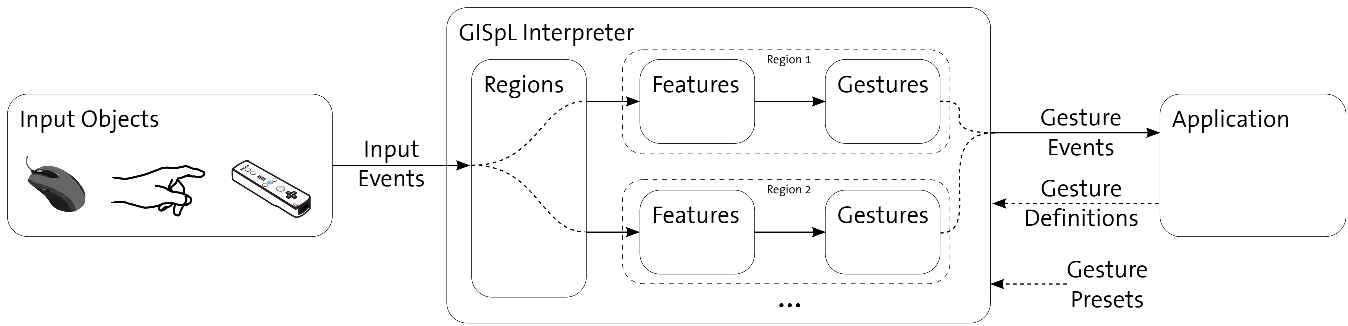

Figure 4. GISpL processing pipeline

Class ID	Description
0	unknown object
1	generic finger (≙ right index finger)
1-5, 6-10	individual fingers on right,left hand
11-17	pointing devices (stylus, laser pointer, mouse, trackball, joystick, Wiimote, eye/gaze tracker)
18-19	tangibles (generic/tagged with ID)
21-23, 24-26	right/left hand (open/pointing/closed)
27-30	right/left foot, head, person

Table 1. GISpL object classes

ues are given as two vectors which are applied componentwise. All values have the same dimensions as given in the TUIO 2.0 specification.

```
{
  "name":"two_finger_swipe",
  "flags":"sticky,oneshot",
  "custom":"userdata",
  "features":[
    {
      "type":"Count",
      "filters":2046,
      "constraints":[2,2],
      "result":[]
    },{
      "type":"Motion",
      "filters":2046,
      "constraints":[
        [0.01,-0.01,-0.01],
        [0.1,0.01,0.01]
      ],
      "result":[]
    }
  ]
}
```

Gesture

Feature 1

Feature 2

Figure 5. Example gesture description in GISpL

This definition is supplied to the GISpL interpreter by the application. When later the motions of the input objects result in feature values which fall within their respective constraint values, then the gesture is triggered and a corresponding gesture event is delivered to the application. This event is in a very similar format, the sole difference being that the constraint values are omitted and a result value is instead supplied for each feature. As this gesture has the *oneshot* flag set, it is sent only once for a set of input IDs. This means, for example, that as long as two fingers remain in contact with the surface and their IDs consequently do not change, only a single event will be triggered, even if the same motion is executed again without lifting them up. After one or more of the touch points have disappeared, the input IDs are reset and the gesture can be re-triggered. As the gesture is also flagged as *sticky*, the participating input objects will ”stick” to the original region, even if their path later crosses other regions. This avoids spurious interactions that might otherwise happen after the primary gesture was triggered.

Feature Types

GISpL currently provides 11 feature types which are grouped into two classes, single-match and multiple-match features. Single-match features will only generate a single result value irrespective of the number of input objects that are present and will consequently only trigger at most a single gesture event per input frame. In contrast, multiple-match features can generate one result value per input object and will therefore potentially generate as many gesture events as there are input objects. An overview of all available features is given in Table 2.

To provide fine-grained control over the conditions which have to be fulfilled for a gesture to be triggered, every feature exhibits a number of constraint values. Depending on the feature type, these constraints may represent, e.g., a lower and upper boundary value, a template path or a bounding box. A gesture can only be triggered when all features match their respective constraints.

Consequently, a gesture may be regarded as a conjunction of several boolean-valued functions. Nevertheless, it is possible to express arbitrary boolean functions over the set of features. By defining several gestures with the same name but different sets of features, a composite gesture corresponding to a disjunctive normal form can be represented. Negation of a feature can be achieved through inverted constraints.

Table 2. GISpL feature types

Single-Match Features				
Name	*Type*	*Unit*	*Constraints*	*Description*
Count	Integer	dimensionless	lower/upper bound	Number of input objects in region (e.g., number of touch contact points)
Delay	Float	seconds	lower/upper bound	Duration since last change in input object set (e.g., seconds since last touch contact entered region)
Path	Float	dimensionless	template path	Accuracy of match between template path and actual path travelled by input objects (supports shape-based gestures as described in [24])
Motion	Vector	TUIO units[1] / s	lower/upper bound (per component)	Average motion vector of all input objects in region
Rotation[2] - MultiPointRotation - RelativeAxisRotation	Float	rad / s	lower/upper bound	Relative rotation of the input objects with respect to their starting positions
Scale[2] - MultiPointScale - RelativeAxisScale	Float	1 / s	lower/upper bound	Relative size change of input objects with respect to their starting positions

Multiple-Match Features				
Name	*Type*	*Unit*	*Constraints*	*Description*
Position	Vector	TUIO units[1]	bounding box	Positions of all individual input objects (e.g., all tangible objects within region)
ID	Integer	Object ID	lower/upper bound	Numerical IDs of input objects (e.g., for identification of tagged tangible objects)
ParentID	Integer	Object ID	lower/upper bound	Numerical IDs of parent objects for parent-child relation (e.g., for recognition of hand and matching fingers)
Dimensions	Vector triple	TUIO units[1]	lower/upper bound (per component)	Physical dimensions of objects as described by equivalent ellipse (e.g., for matching objects of certain shape)
Group	Integer	TUIO units[1]	minimum/maximum group radius	Group of input objects as specified by minimum & maximum radius of group (e.g., for matching groups of closely spaced touch points)

[1] TUIO units range from 0.0 to 1.0 in all dimensions and cover the entire sensor range in each dimension, e.g. the X axis of a touchscreen.

[2] These features both exist in two variants which react to the relative rotation and scale change of either a single object or of a group of point-like objects.

Should boolean functions still be insufficient to express more complex interactions, it is also possible to daisy-chain several feature blocks as shown below:

```
"features":[
  [ { feature1.1 },{ feature1.2 } ],
  [ { feature2.1 },{ feature2.2 } ]
]
```

In this example, the features 1.1 and 1.2 will be examined first. Only if both match their respective constraints, then processing will proceed to the second block containing 2.1 and 2.2. If, for example, every block contains a *Delay* feature, this can be used to represent sequences of interactions such as a double-tap within a single gesture.

Gesture Flags

As mentioned above, gestures can be differentiated further by specifying certain flags. The *oneshot* flag prevents a gesture from being triggered continuously. Consider, for example, a simple ”move” gesture: as long as the user intends to move an object, small incremental updates of the motion vector should be sent to the application to ensure smooth behavior of the graphical representation. On the other hand, gestures such as ”touch” and ”release” represent singular events which should be signalled only once, even if the input conditions persist. Consequently, these gestures should be flagged as *oneshot*.

Another important feature can be accessed by flagging a gesture as *default*. Such gestures are added to an internal pool of presets. When an application later specifies a gesture without features which could normally never be triggered, the gesture name is used to look up a potential replacement gesture in the preset pool which has the same name. This behavior offers a powerful method to adapt an application to input hardware with varying capabilities. For example, the GUI may use an empty ”rotate” gesture and rely on the preset pool to deliver a result suitable for the present input

device. While this result may be generated through multi-touch on suitable input devices, another system might instead use tangibles which provide a custom definition for the same effect. The GISpL reference implementation already provides a number of presets for common gestures such as move, scale and rotate.

Finally, the *sticky* flag causes a gesture to permanently capture the participating input objects into its containing region, once the gesture has been triggered for the first time. This capture remains valid as long as the input objects are tracked and the gesture is valid, even if the input objects leave the original region. Once one of these conditions is no longer met, the capture is removed and the input objects can again be captured by all regions. This flag thereby enables gestures to continue outside their initial regions.

Matching Algorithm

We will now briefly describe the internal algorithm employed by the GISpL interpreter. This algorithm is executed once for each complete set of input events, e.g. for one fully processed camera image. First, input events are divided according to the defined regions. As regions may overlap, they are arranged in a list. The topmost region to be hit by the input event captures this event. It is appended to an internal queue, thereby creating a history of input events for this region. In the next step, the total set of features used per region is determined and the result value of each feature is calculated once from the input history. Finally, for every gesture, the feature values determined in the previous step are compared with their respective constraints. Should all constraints be satisfied, then the tuple of region and gesture is delivered to the application. A pseudo-code representation of this algorithm is given below.

```
for each input event e:
  for each region r:
    if (e.type in r.filters)
    and (e inside r):
      append e to r.input_queue
      continue with next event

for each region r:

  if gestures in r modified:
    clear r.feature_set
    for each gesture g in r:
      for each feature f in g:
        insert f into r.feature_set

  for each feature f in r.feature_set:
    calculate f.value from r.input_queue

  for each gesture g in r:
    for each feature f in g:
      check f.value against constraints
      if mismatch:
        continue with next gesture
    send tuple(r,g) to application
```

Depending on the type of feature, the calculation of result values may be performance-intensive. However, this is mitigated by the fact that every feature is calculated at most once per region. Moreover, the features in one region can be calculated independently of the others, thereby enabling easy parallelization.

IMPLEMENTATION EXAMPLES

We have already presented two example usage scenarios in which GISpL has been tested. We will now highlight some interesting implementation aspects of these applications.

Input-Agnostic Spatial Transformations

Affine spatial transformations are one of the main interaction modes used with novel user interfaces. These can be supported using GISpL by splitting the transformation into three separate events representing motion, rotation and scaling.[2] An abbreviated sample definition which would be appropriate for a multi-touch screen is given below. Events will be sent after every input frame in which objects matching the filters (i.e., any type of finger) have moved within the containing region. The ”move” event will deliver a result value representing the average motion vector of these objects with respect to the previous frame. The ”rotate” and ”scale” events will contain results which describe the average rotation and scale change of all contact points with respect to their common centroid, relative to the previous frame.

```
[{
  "name":"move",
  "features":[{
    "type":"Motion",
    "filters":2046
}]},{
  "name":"rotate"
  "features":[{
    "type":"MultiPointRotation",
    "filters":2046
}]},{
  "name":"scale"
  "features":[{
    "type":"MultiPointScale",
    "filters":2046
}]}]
```

Should a different input modality be desired later, such as using a tangible object for rotation, then the ”rotate” part of the definition could be changed as follows:

```
{ "name":"rotate"
  "features":[{
    "type":"RelativeAxisRotation",
    "filters":131072
}]}
```

The filter bitmask has been changed to select untagged tangible objects (object type 17), while the feature type has been changed to react to the relative rotation of the major axis of

[2]Support for all affine transformations would include shearing, however, this is not commonly used.

a single tracked object. As the result type stays the same (see also table 2), the change is fully transparent to the application. Any trackable object can be put into the on-screen region, rotated in order to adjust the region's rotation and removed again. Motion and scaling can still be performed using one or more fingers.

Using GISpL in Practice: Code Example

In order to use the GISpL matching engine in a real-world application, this application needs to supply spatial data about sensitive regions and gesture definitions. This can be done in only a few lines of code, as shown in the following C++ code snippet for a static triangular widget:

```
extern Matcher* g_matcher;

class MyWidget: public GestureSink {

  // ...

  // define spatial extents of region
  Region reg;

  reg.push_back(Vector(  0,  0,0));
  reg.push_back(Vector(100,100,0));
  reg.push_back(Vector(100,  0,0));

  // GISpL definition - see above
  // can also be taken from user config
  std::string definition = "...";
  Gesture rotate(definition);

  // attach gesture to region
  reg.gestures.push_back(rotate);

  // send region to matching engine
  g_matcher->update( this, &reg );
}
```

The global object `g_matcher` represents the matching engine. The widget registers one or more regions containing gesture definitions with the engine, along with a reference to the widget itself. When later an event corresponding to one of the registered gestures is triggered, this reference is used by the matching engine to pass the event object to a callback method in the widget (defined in class `GestureSink`, not shown). The event object will deliver a gesture result in binary form, obviating any need for the developer to parse GISpL data himself. Should the event cause any modification of the widget's spatial extents, another call to `update` can be used to send a current region definition to the matcher.

DISCUSSION

Comparing GISpL to the body of related work already listed above, it is apparent that there hasn't yet been any attempt to unify gesture recognition in the widest sense across the different paradigms for novel user interfaces that are in use today. Of course, this leads to the question whether such an approach is sensible - after all, there is always the possibility that existing and/or future user interfaces may have sensing capabilities that cannot be fully captured using such a common denominator.

However, we believe that by introducing a first attempt at such a common denominator, we can stimulate discussion on the common points of novel user interfaces themselves and their development.

There are some aspects of GISpL which still offer room for improvement. For example, the matching of input objects to regions is done based on the input object's centroid and can therefore not capture objects which are intended for interaction with more than one region at once. A possible way to deal with this limitation would be to check for collisions of an input object's hull with regions similar to [22], thereby enabling matches with more than one region simultaneously.

The expressive power of GISpL is dependent on the available features and their constraints. Future types of UIs may require the introduction of additional feature classes. However, as most of the 11 current feature classes contain less than 50 lines of C++ code, this is not an insurmountable task.

Also, even though the boundary conditions for gestures can be thought of as arbitrary boolean functions over their features, there may be applications where this type of expression is not powerful enough or too cumbersome. One potential extension to GISpL would be to allow arbitrary JavaScript functions in a gesture which calculate a boolean result from the feature values. This result would then determine whether a gesture is triggered or not. This approach would perhaps also mitigate the problem that GISpL descriptions, even for simple gestures, can appear quite complex at first.

Currently beyond the scope of GISpL is textual input by speech, keyboard or other means. One potential way to extend GISpL in this direction could be provided by adding a feature comparing sets of given strings with a textual input stream. The validity of this idea has to be a subject of further study.

CONCLUSION & OUTLOOK

We have presented GISpL, a formal language that enables the unambiguous description of gestural user interfaces across a wide range of input devices. Developers, researchers and users alike can benefit from the logical separation between motions and actions. GISpL allows easier portability of applications across different input devices and enables quick modifications of gestural interfaces which previously required changes to the source code. A full specification of GISpL as well as an open-source reference implementation is available at `http://www.gispl.org/`.

GISpL is based on fundamental concepts developed in several projects over the course of three years. General feedback was very positive, with other research groups adopting it for their applications. Continuous feedback was used for refinement of the GISpL specification and for making the recognition engine robust, efficient and available on all major platforms.

REFERENCES

1. R. Brooks. Human Markup Language Primary Base Specification 1.0. http://www.oasis-open.org/committees/download.php/60/HM.Primary-Base-Spec-1.0.html. accessed 2011/03/29.
2. D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt. accessed 2011/03/29.
3. P. Dragicevic and J.-D. Fekete. Support for input adaptability in the icon toolkit. In *Proceedings of the 6th international conference on Multimodal interfaces*, ICMI '04, pages 212–219, New York, NY, USA, 2004. ACM.
4. F. Echtler and G. Klinker. A multitouch software architecture. In *Proceedings of NordiCHI 2008*, pages 463–466, Oct. 2008.
5. F. Echtler, G. Klinker, and A. Butz. Towards a unified gesture description language. In *Proceedings of the 13th International Conference on Humans and Computers*, HC '10, pages 177–182, Fukushima-ken, Japan, 2010. University of Aizu Press.
6. S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218, 2001.
7. T. Hammond and R. Davis. Ladder, a sketching language for user interface developers. *Computers & Graphics*, 29(4):518 – 532, 2005.
8. T. E. Hansen, J. P. Hourcade, M. Virbel, S. Patali, and T. Serra. PyMT: a post-WIMP multi-touch user interface toolkit. In *ITS '09: Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, pages 17–24, New York, NY, USA, 2009. ACM.
9. M. Kaltenbrunner. TUIO 2.0 Specification. http://www.tuio.org/?tuio20. accessed 2011/03/29.
10. 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.
11. D. Kammer, J. Wojdziak, M. Keck, R. Groh, and S. Taranko. Towards a formalization of multi-touch gestures. In *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '10, pages 49–58, New York, NY, USA, 2010. ACM.
12. S. H. Khandkar and F. Maurer. A language to define multi-touch interactions. In *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '10, pages 269–270, New York, NY, USA, 2010. ACM.
13. S. R. Klemmer, J. Li, and J. Lin. Papier-mâché: Toolkit support for tangible input. pages 399–406. ACM Press, 2004.
14. W. König, R. Rädle, and H. Reiterer. Squidy: a zoomable design environment for natural user interfaces. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, CHI EA '09, pages 4561–4566, New York, NY, USA, 2009. ACM.
15. U. Laufs, C. Ruff, and J. Zibuschka. MT4j - A Cross-Platform Multi-Touch Development Framework. June 2010.
16. Microsoft Corporation. Surface SDK. http://www.microsoft.com/surface/Pages/Product/Platform.aspx, 2009 (accessed 2009-31-12).
17. NUI Group Community. Community Core Vision. http://ccv.nuigroup.com/, accessed 2009-07-06.
18. C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. Midas: a declarative multi-touch interaction framework. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, TEI '11, pages 49–56, New York, NY, USA, 2011. ACM.
19. 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 '08 extended abstracts on Human factors in computing systems*, CHI EA '08, pages 3501–3506, New York, NY, USA, 2008. ACM.
20. C. Shen, F. Vernier, C. Forlines, and M. Ringel. DiamondSpin: an extensible toolkit for around-the-table interaction. In *CHI '04: Proceedings of the Conference on Human Factors in Computing Systems*, pages 167–174, 2004.
21. I. E. Sutherland. Sketchpad - a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, DAC '64, pages 6.329–6.346, New York, NY, USA, 1964. ACM.
22. A. Wilson, S. Izadi, O. Hilliges, A. Garcia-Mendoza, and D. Kirk. Bringing physics to the surface. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 67–76, New York, NY, USA, 2008. ACM.
23. J. O. Wobbrock, M. R. Morris, and A. D. Wilson. User-defined gestures for surface computing. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1083–1092, New York, NY, USA, 2009. ACM.
24. J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 159–168, 2007.