A Domain Specific Language to Define Gestures for Multi-Touch Applications

Shahedul Huq Khandkar, Frank Maurer
Department of Computer Science
University of Calgary, Canada
{s.h.khandkar, frank.maurer}@ucalgary.ca

ABSTRACT
It is increasingly common for software and hardware systems to support touch-based interaction. While the technology to support this interaction is still evolving, common protocols for providing consistent communication between hardware and software are available. However, this is not true for gesture recognition – the act of translating a series of strokes or touches into a system-recognizable event. Developers often end up writing code for this process from scratch due to the lack of higher-level frameworks for defining new gestures. Gesture recognition can contain a significant amount of work since it often involves complex, platform-specific algorithms. We present a domain-specific language that significantly simplifies the process of defining new gestures and allows them to be used across multiple hardware platforms.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Domain Specific Language, Human Factors, Standardization, Languages.

Keywords
Gesture, Multi-Touch, Multi-User, Touch Interaction, Gesture Definition Language (GDL), TouchToolkit, Domain-Specific Language.

1. INTRODUCTION
Within the domain of human computer interaction, touch has been considered an interaction medium for an extensive period of time. Until recently however, it was limited to recognizing single touch interactions such as selecting options or entering numbers in kiosk systems in banks, stores, etc.: Touch was basically treated as a mouse replacement. Recent innovations in multi-touch devices, have initiated new opportunities for computer interaction that are fundamentally intuitive and natural. As these devices become increasingly affordable, it is essential to create new applications and extend existing ones to support touch-based interaction.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DSM:10, 17-OCT-2010, Reno, USA
Copyright © 2010 ACM 978-1-4503-0549-5/10/10…$10.00.

Comparatively, multi-touch is a newer interaction technique, where different types of touches including multiple fingers, hands or arbitrary tangible objects, can be used to interact with a system. While research to find the most suitable multi-touch hardware technology is ongoing, a number of devices are available that use different approaches to support this form of interaction.

To utilize this newer medium of input in applications, developers will require support from proper frameworks and tools. Currently, application developers predominantly use software development kits (SDK) provided by hardware vendors that are hardware specific. These SDKs provide the necessary infrastructure to communicate between hardware and software as well as to some extent touch enabled user interface widgets. However, they provide limited high level frameworks for programming against the touch interactions. As a result, developers often end up building touch interaction related modules and gestures from scratch.

Depending on the features of a multi-touch device, a command may be triggered by strokes, touches, whole hand interactions, tangible object interactions or even multiple concurrent touches from different people. In this paper, we primarily focus on simple as well as complex touch interactions. Processing the raw touch interaction data provided by the hardware into meaningful application-recognizable events sometimes involves complex algorithms that become cumbersome when fine tuning is required. Subsequently, developers typically select gestures based on implementation complexity instead of usability.

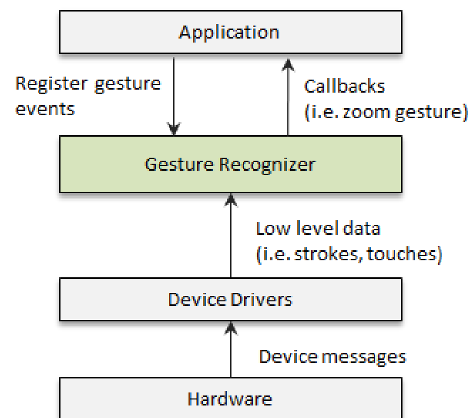

Figure 1: The Gesture Recognition Process

Figure 1 shows the processing pipeline of a gesture recognition system. The messages generated from hardware are translated into low-level data by the device drivers. A gesture recognizer then receives these data and evaluates them to detect gestures that are requested by the application. When a gesture is detected, the

gesture recognizer notifies the application through an asynchronous callback.

Application developers sometimes need to develop gesture recognition modules to support a new gesture that is natural and meaningful for the application. The process can contain a significant amount of work since it often involves complex, platform-specific algorithms that require special background knowledge to develop and fine tune.

To simplify using gestures in applications, we present the gesture definition language (GDL) that hides low level implementation complexities from application developers without compromising the flexibility of gesture definitions. GDL is a domain-specific language designed to streamline the process of defining gestures. It enables application programmers to integrate application specific gestures into their software systems. GDL is part of TouchToolkit[1] - a software development kit we developed to simplify multi-touch application development and testing. Currently, our tool integrates with Windows Presentation Foundation and Silverlight.

The requirements of GDL were collected in two steps. First, we studied existing research [7] [13] on gestures for multi-touch surfaces to gather a list of useful gestures and compared them with existing frameworks and SDKs[2,3,4]. We then studied requirements coming from several multi-touch applications [1] [2] [11] and interviewed three developers. Upon comparing the requirements and other available frameworks, we realized that it is not practical for a framework to provide every single gesture predefined out of the box. Instead, a domain-specific language to define custom gestures was seen as a more appropriate solution. Following are the four key areas we address in GDL:

- Separation of concerns,
- Flexibility,
- Extensibility, and
- Independence from hardware

We discuss these in detail in section 3.

The remainder of this paper is organized as follows. We start by comparing our work to earlier work. Next, we explain the requirements that we considered during the language development. A discussion follows on how GDL helps solve these practical problems. Next we describe the internal architecture for the language, the compiler, the hardware abstraction layer and the integration of GDL with developer tools. Finally, we discuss the limitations of our work before summarizing this paper in the conclusion.

2. RELATED WORK

Wobbrock [12] proposed a gesture recognition system that recognizes gestures by comparing them with base templates. An advantage of this approach is that it allows the definition of new gestures by adding additional templates. However, this approach does have a number of limitations. For example, it cannot detect gestures in a continuous motion stream and only gestures with explicit start and end points can be processed. Gestures like a lasso (Figure 1) which can be of an arbitrary shape, cannot be detected using this approach. GDL provides the flexibility to use combinations of primitive conditions to define such gestures. It is also possible in GDL to generate gesture definitions from raw touch data.

Kartz [5] proposed another approach that relies solely on simple trigonometric and geometric calculations. His approach requires considerably less training data than some other recognizers. However, it too suffers from limitations such as a smaller gesture vocabulary size and it cannot process gestures with continuous motion. The primitive conditions of GDL include trigonometric and geometric calculations that can be used with other conditions.

Furthermore, both approaches were proposed for single touch systems and are often not suitable for multi-touch scenarios that involve sequences of multiple concurrent touches. In GDL, developers define gestures at a level where they don't need to deal with complexities of multiple touch and users. The gesture validator encapsulates these complexities.

GestureWorks [6] is an application framework for Adobe Flash and Flex. It provides a set of predefined gestures to simplify the development of multi-touch interactions. While it helps save development effort to some extent, it currently works only on Windows 7 supported multi-touch devices. Among many other multi-touch devices, the Microsoft Surface does not have Windows 7 support at the moment. In addition to that, developers are limited to choose from the list of predefined gestures, as it doesn't provide options to define new gestures.

SparshUI [8] is an API for multi-touch applications that provides the extensibility to define new gestures. To create a new gesture in SparshUI, the developer needs to create java classes which involve many low level implementation details including data serialization, implementing interfaces for gestures and events, and so on. Alternatively, GDL is a domain-specific language specially designed for gestures. It simplifies the process greatly by handling features such as method chaining, multi-step gestures, context-oriented primitive types (e.g. distance: increasing) internally. Finally, custom return types that are often complicated and time consuming to implement with standard programming languages can be defined using GDL.

3. REQUIREMENTS

Based on the results of existing research on useful gestures [7] [13], our analyses of different multi-touch applications and comments from developers, we found a set of requirements for a gesture definition language.

3.1 Separation of Concerns

Associating system commands with gestures is an important part of developing multi-touch applications. At present, application developers not only write application specific code but also need to write the gesture recognition modules that recognize a gesture from raw touch data. The gesture recognition process is a complicated process that is often hard to fine tune and requires special background knowledge. As a result, developers either spend a significant amount of time to implement the correct

[1] http://touchtoolkit.codeplex.com

[2] http://www2.smarttech.com

[3] http://msdn.microsoft.com/enus/ library/dd371406(VS.85).aspx

[4] *Microsoft Surface*. 2010. http://www.microsoft.com/surface/

gesture, or select a gesture that is easy to implement. In essence, application developers make compromises on an application's usability.

A domain-specific language (DSL) for definition gestures can hide the low level complexities by encapsulating complex mathematical calculations, pattern recognition algorithms and the like. This can help the developers focus on designing the gesture at a level that is most appropriate to the application context without worrying about the implementation details.

3.2 Flexibility

A specially designed DSL for gestures can help developers focus on application design instead of low-level gesture implementation complexities. However, it should also ensure that it provides the necessary flexibility to define the gesture that is meaningful to the application regardless of its complexity. The language should also allow gestures that may depend on device specific features (i.e. user identification, pressure sensitivity).

Another important part of a gesture definition is to prepare the results when the gesture is detected. Some gestures need only the touch position (i.e. tap), whereas others need more detailed information like the boundary of an arbitrary shape drawn by the gesture (i.e. lasso), direction of the finger (i.e. one finger drag), etc. The language should provide options to define new return types as necessary.

3.3 Extensibility

Researchers are actively working on finding the best technology for multi-touch interaction. While existing technologies such as diffuse elimination, frustrated total internal reflection (FTIR) and capacitive touch are widely becoming available to consumers, new technologies continue to emerge in the research arena. For example, "UnMousePad" [9], a flexible and inexpensive multi-touch input device that provides data on touch pressure in addition to touch position.

As new technologies are discovered, the language should provide the infrastructure to add new features without affecting the existing applications.

3.4 Device Independence

Multi-touch interactions [3] were initially developed in the early 1980's. Since then a number of different technologies have been introduced by different industrial and research labs. Hardware vendors provide different multi-touch devices with similar features that are driven by different technologies. Due to a lack of standards in the field, these hardware vendors often end up implementing the device-to-software communication systems differently. As a result, applications sometimes become so dependent on a particular device that the developer needs to rewrite significant portions of their application to make it compatible for another device. For example, an application developed for the Microsoft Surface using their SDK, will not work on other devices like a SMART Table.

The gesture definition language, along with its compiler and related framework, needs to be device independent so that it can work on multiple devices without any change in application code.

4. IMPLEMENTATION

In Section 3, we described how GDL can address some of the challenges of multi-touch application development. In this section we describe the language and how it can be used to solve the issues we discussed earlier.

Figure 2 shows the structure of a simple gesture definition. A gesture definition contains three sections: a *name* that uniquely identifies a definition within the application; one or more *validate* blocks that contain combination of primitive conditions; and finally the *return* block that contains one or more return types.

```
name: <unique identifier>

validate
    /* code to detect a gesture */

return
    /* one or more return types */
```

Figure 2: The Structure of Gesture Definition

The *name* must be unique within the scope of the application. GDL is part of TouchToolkit that provides a set of commonly used gestures including zoom, drag, rotate, lasso, flicks in different directions, geometric shapes, and so on out of the box. If a developer wants to override any of the predefined gestures, they may use the same name. The compiler will override the predefined gesture with their defined gesture. However, if the user mistakenly defines two gestures with the same name, the compiler will throw an exception message.

The *validate* block contains the logic for evaluating raw touch data to detect a gesture. The logics are defined using combination of primitive conditions, the smallest unit to evaluate the raw data.

Table 1: Example of Primitive Conditions

No	*Primitive Condition*
1	Distance between points: increasing
2	Touch limit: 1..4
3	line1 perpendicularTo line2

Primitive conditions can be of different types. Table 1 shows some examples of primitive conditions that can be used to define a behavior pattern of touch points (No.1), the range of touch points allowed in the specifying gesture (No.2), and a geometric condition between two previously recognized partial results of a multi-step gesture (No.3). Details on multi-step gesture are discussed in section 4.2. There are currently 15 primitive conditions available out of the box. A detailed list can be found in the project website at **Error! Reference source not found.**. Developers can also create their own primitive condition which is described in section 4.3.

Multiple primitive conditions are virtually connected like a chain using the logical operators (i.e. and). The validation process follows the lazy evaluation approach where it starts from the first primitive condition in the chain and it only passes the valid data set (or multiple possible sets) to the next condition in the chain. This allows the system to improve performance by realigning the elements of the virtual chain without breaking the logic. When multiple validate blocks are defined, the compiler considers each block as a step in a multi-step gesture and performs the validation in the order it is defined.

The last section in a gesture definition is the *return* block. Users can specify any number of return types. Each of the return type is linked to a return type calculator. The runtime gesture validation engine passes the final set of valid touch data to each of the return type calculators and finally sends the results to the application layer through a callback event. The common return types including touch position, bounding box, direction, unique id (when supported by hardware), rotation and many more are predefined out of the box.

Like primitive conditions, return types are also extensible. We now describe how GDL addresses some of the key issues of multi-touch application development including the steps to create new primitive condition and return type.

4.1 Hiding Low Level Complexities

Let's consider a scenario where a user may use a lasso gesture (Figure 3) to select some objects from a scattered collection of objects.

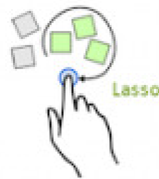

Figure 3: Using the "Lasso" gesture to select multiple objects from a scattered view

Implementing this gesture from scratch means processing the raw touch inputs that mostly contain the position and order of touch points. Thus, the developer needs to write code to check the following conditions at a very low level:

- Is this the last action of current touch stroke? The gesture should be evaluated when the touch stroke ends.
- Does the collection of points in the specific touch stoke represent a closed loop?
- Is the area of the bounding box and the length of the path within a certain limit?
- Is the area of the arbitrary shape created by the enclosed path within a certain limit?
- Only one touch should be involved in this gesture. If multiple active touch points are available then it should consider each point individually.

Some of the validation logic like the calculation of the area of an arbitrary shape could involve complex mathematical equations and requires proper testing.

Figure 4 shows the GDL code to detect the lasso gesture using the above logic. Implementing this from scratch not only requires a lot of development time, but also additional time to test various possible user scenarios.

Also, the order of condition validation can significantly affect the overall performance of the system. For example, it is quite simple to check the state of the touch action compared to calculating the enclosed area of an arbitrary shape. The GDL compiler can internally reorganize the order of condition validation, to improve performance.

```
name: Lasso

validate
    Touch state: TouchUp
    Touch limit: 1
    Closed loop and
    Touch path bounding box: 200x200..1000x1000 and
    Touch path length: 600..100000 and
    Enclosed area:5000..1000000

return
    Touch points
```

Figure 4: Defining the lasso gesture using GDL

4.2 Flexibility

Hiding low level implementation details can give the desired simplicity and improve productivity of developer. However, it should also provide the necessary flexibility to define gestures of various requirements. Let's think about a scenario where a gesture may be composed of touches in multiple steps. For example, in a UML designer tool (i.e. Smart UML [11]) the user would do the following touches to create an "Actor" object.

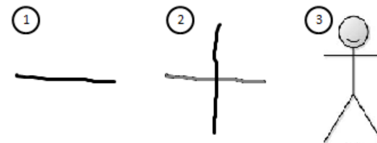

Figure 5: Sequence of touch strokes to create an "Actor"

This means the developer not only needs to detect a gesture of certain characteristics, but also keep track of history to use the results of partial validation for later use. Keep in mind that we are talking about multi-touch scenarios that may involve multiple users and some of these partially validated results could end up representing different gestures.

```
name: Actor
validate as firstLine
    Touch state: TouchUp and
    ...
validate as secondLine
    Touch state: TouchUp and
    ...
validate
    firstLine perpendicularTo secondLine and
    ...
return
    Position, Bounding box
```

Figure 6: Defining Actor gesture for Use Case diagram

To address this issue, GDL provides the syntax to define validation in multiple steps, as well as the storage of partial results for later use. The following code snippet (Figure 6) defines the actor gesture. In code, the intermediate results of the first and second steps are stored in variables defined using the "as" keyword. These variables can also store multiple partial results if necessary.

4.3 Extensibility

Multi-touch devices are evolving at a great speed. Until just recently, devices were mostly providing touch points and user identification for some specific devices [12]. Now some devices can provide touch directions **Error! Reference source not found.** and information about the pressure of a touch [9]. The extensibility framework of GDL allows creating new primitive conditions as well as return types.

The process of adding a new primitive condition can be described in two steps. First, update the language grammar that is used to parse the code. Figure 7 shows a code snippet of the grammar that is responsible for parsing the "TouchStep" primitive condition.

```
/* Primitive condition: Touch Step */
syntax TouchStepRule
    = "Touch step" ":" x:ValidNum "touches" "within"
      y:ValidNum z:TouchStepUnitsForTime
    => TouchStep{TouchCount=>x, TimeLimit=>y, Unit=>z};

token TouchStepUnitsForTime
    = "sec" | "msec";
```

Figure 7: A code snippet of GDL grammar to parse the "Touch Step" primitive condition

Then, create a validator that takes raw touch data as input and does the validation. A class implementing an interface written in any .NET supported language can contain the computation logics. In the same way, we can also add new return types in the language.

4.4 Device Independence

In section 3.3, we discussed different technologies that are currently being used by different multi-touch device vendors. Due to the nature of the software development kits (SDK) provided by the hardware vendors to build multi-touch applications, the developed applications often become tightly coupled with the SDK. The result is that significant portions of the application need to be rewritten, to simply run it on another device with similar features. To overcome this, GDL is designed to be independent of these SDKs and applications developed using it, can easily be ported onto different devices without changing any application source code. In addition, GDL provides a domain-specific language to define custom gestures, whereas in most SDKs supplied by the hardware vendors, a limited set of predefined gestures (i.e. Microsoft Surface SDK) and low-level touch data is provided. Figure 8, shows how we can change a device with just one line of code. This can also be handled through configuration settings or automatic hardware detection. More detail on the internal architecture will be discussed in Section 5.

```
/* Select hardware */
var provider = new Windows7TouchInputProvider();
//var provider = new SurfaceTouchInputProvider()

/* Initialize Gesture Framework */
GestureFramework.Initialize(provider, ...);
```

Figure 8: Changing device/input source of the application

The system also allows for the changing of devices while the application is running. This is useful for scenarios where additional external devices like the AnotoPen can be connected at run time or to connect virtual devices that can simulate certain activities for debugging, testing or demonstration.

5. ARCHITECTURE

The architecture of GDL can be divided into two sections: the gesture recognition framework and the hardware abstraction layer that provides the device independence.

5.1 Gesture Recognition Framework

The framework provides an interface to subscribe events for specific gestures in a method similar to how applications receive messages for mouse or keyboard events. Figure 9 shows the code snippet to subscribe a gesture named "zoom". It also allows defining the scope of the gesture which is an image object in this case.

```
G.EventManager.AddEvent(image1, "zoom", ZoomCallback);
void ZoomCallback(UIElement sender, GestureEventArgs e){

    var dis = e.Values.Get<DistanceChanged>();
    if (dis != null)
        Resize(sender as Image, dis.Delta);
}
```

Figure 9: The code snippet to subscribe "Zoom" gesture

The framework passes the source of the gesture and return types specified in the gesture definition through the arguments of the callback method.

Figure 10 shows the internal architecture of the framework that runs the gesture recognition engine. The gesture definitions and primitive conditions live outside the core framework and are loaded on demand. Therefore, the framework only loads the gesture definitions that are registered by the application at run time. This helps utilize the memory efficiently and also improves performance.

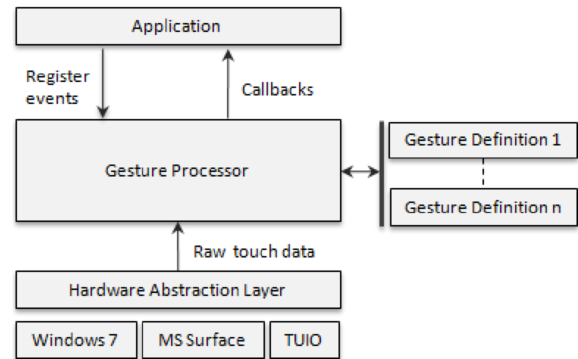

Figure 10: The architecture of gesture recognition engine

Gesture processor is responsible for efficiently evaluating the primitive conditions. For example, if a primitive condition is used in multiple gestures and needs to be evaluated under the same context then the gesture processor will take necessary steps to perform the validation once and reuse the output later.

Figure 11 describes the workflow of the gesture recognition process. When touch data is received from the hardware layer, the toolkit evaluates the primitive conditions defined in validate blocks of the registered gestures. The framework internally handles the multi-user scenarios during result storage and evaluation of primitive conditions in each block. This is because

gestures may appear in parallel when multiple users interact simultaneously. Once a gesture is recognized, the gesture processor calculates the requested return values and notifies the application through the event controller.

5.2 Hardware Abstraction Layer

We followed a similar approach as Echtler [10] to decouple the actual hardware from the application layer. This allows the gesture definition to be device independent. In this module, there is a hardware agnostic interface for capturing multi-touch inputs. This interface can be implemented for wide range of hardware platforms. We currently have implementations for Microsoft Surface, SMART Tabletop, Windows 7, Anoto Pen[5] and TUIO protocol. The framework has an implementation for a virtual hardware that can be used to simulate multi-touch inputs. It can also playback the recorded interactions and run automated tests.

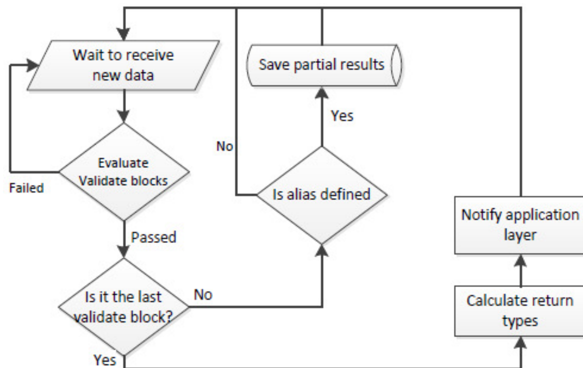

Figure 11: The workflow of gesture recognition

6. ONGOING AND FUTURE WORK

Current research also focuses on generating gesture definitions from sample dataset of touch interactions and a visual representation of the gesture definition. This will allow users to define a gesture from sample touch data and use the visual DSL to fine tune the logical conditions. We are also working on adding additional logical operators in the language.

7. LIMITATION

The language and related frameworks are developed using Microsoft .NET and well integrated with Visual Studio IDE. This makes it easy to use for any application that runs on the same platform. It does not however, support application development with non-Microsoft languages.

GDL is intended to be used for multi-user, multi-touch based applications. However, some other tabletop input techniques such as Smart Tags or physical object based interactions are not yet supported by this tool.

8. CONCLUSION

We introduced a domain-specific language to define gestures for multi-user, multi-touch scenarios. It helps the developers to focus on designing gestures that are the most natural and meaningful to application's context without worrying about low level implementation details. The tool supports including syntax highlighting, on-the-fly error tracking also help reduce the learning curve. The gesture definition language is part of TouchToolkit - a software development kit to simplify the development and testing complexities of multi-touch applications. Both the language and the toolkit provide an extensibility framework to add new devices and interaction features to support new hardware. We believe GDL can help to produce real-world multi-touch applications with good quality within an affordable timeframe.

[5] http://www.anoto.com/

9. ACKNOWLEDGMENTS

We would like to thank Teddy Seyed, Andy Phan, Darren Andreychuk, Theodore Hellmann and Ali Hosseini Khayat for their contribution on evaluating the language.

10. REFERENCES

[1] *eGrid: 2010* http://egrid.codeplex.com/

[2] Wang, X., Ghanam, Y., Park, S. and Maurer, F. Using Digital Tabletops to Support Distributed Agile Planning Meetings, *In Proc. of 10th International Conference on Agile Processes and eXtreme Programming (XP 2009)*, Demo Abstract, Pula, Italy, 2009

[3] Lee, SK., Buxton, W., and Smith K.C. 1985. A Multi-Touch Three Diemensional Touch-Sensitive Table. *In Proceedings of the SIGCHI conference on Human factors in computing systems.* CHI '85. ACM, San Francisco, California.

[4] J. O Wobbrock, A. D Wilson, and Y. Li, 2007. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *In Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST, Newport, RI.

[5] Kratz, S., and Rohs, M. 2010. A $3 gesture recognizer: simple gesture recognition for devices equipped with 3D acceleration sensors. *In Proc. of the 14th international conference on Intelligent user interfaces*, IUI'10.

[6] *GestureWorks, a multitouch application framework for Adobe Flash and Flex*. 2010. http://gestureworks.com/

[7] Wobbrock, J.O., Morris, M.R., and Wilson, A.D. 2009. User-defined gestures for surface computing, *In Proc. of the 27th int. conference on Human factors in computing systems*.

[8] *Sparsh-ui*. 2010. http://code.google.com/p/sparsh-ui/w/list

[9] Ilya Rosenberg and Ken Perlin, "The UnMousePad - An Interpolating Multi-Touch Force-Sensing Input Pad" *ACM Transactions on Graphics* 28, no. 3 (7, 2009): 1.

[10] Echtler, F., and Klinker, G. 2008. A multitouch software architecture. *In Proc. of the 5th Nordic Conference on Human-Computer interaction: Building Bridges.* NordiCHI'08. ACM, New York, NY

[11] *SmartUML*. 2005 http://smartuml.sourceforge.net

[12] Dietz, P. and D. Leigh, 2001. DiamondTouch: a multi-user touch technology. *In Proceedings of the 14th annual ACM symposium on User interface software and technology*. UIST'01, Orlando, Florida, USA.

[13] North, C., Dwyer, T., Lee, B., Fisher,D., Isenberg, P. Robertson, G., Inkpen, K., and Quinn, K.I. 2009. *Understanding Multi-touch Manipulation for Surface Computing*. Interact'09, Uppsala, Sweden.