UNIVERSITY OF CALGARY

A Domain-Specific Language for Multi-Touch Gestures

by

Shahedul Huq Khandkar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December 2010

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled " A Domain-Specific Language for Multi-Touch Gestures" submitted by Shahedul Huq Khandkar in partial fulfilment of the requirements of the degree of Master of Science.

Supervisor, Dr. Frank Oliver Maurer, Department of Computer Science

---

Dr. Saul Greenberg, Department of Computer Science

Dr. Richard Levy, Faculty of Environmental Design

Date

Abstract

Touch has become a common interface for human computer interaction. Portable hand held devices like smart phones to tabletops, large displays and even devices that project on arbitrary surfaces support touch interface. However, at the end, it is the applications that bring meaning for these technologies to people. Incorporating a touch interface in applications requires translating meaningful touches into system recognizable events. This notion of meaningful touch(s) to interact with the system is called gesture. The process of gesture recognition often involves complex implementations that are sometimes hard to fine tune. Due to the lack of higher-level frameworks, developers often end up writing code from scratch to implement touch interactions in their application. Furthermore testing is essential to ensure quality of the application. The lack of automated test frameworks forces developers to rely on manual testing which is time consuming and open to human errors. To address these issues, we present a domain-specific language that defines multi-touch interactions, thus hiding the complexities of low-level implementation from application developers, along with an automated testing framework for touch based interactions. The language allows a developer to focus on designing touch interactions that are natural and meaningful to the context of their application without worrying about implementation complexities; and the test framework helps to detect errors earlier by running the test frequently in an automated fashion.

Acknowledgements

I would like to take this opportunity to thank everyone that helped and supported me to complete the research during the two years.

First of all, many thanks to my supervisor, Dr. Maurer, for limitless support, advice and help in pursuing this thesis. The freedom in choosing a research area and continuous guidance in every stage of the research can hardly be imagined. I would also like to thank Dr. Sillito and Dr. Greenberg for their support and guidance on projects that helped this research in many ways.

Special thanks to Teddy Seyed and Andy Phan for their contribution in development and testing of GestureToolkit. Also, let me express my gratitude towards all my friends in Agile Software Engineering Lab and elsewhere, Ali Hosseini Khayat, Shafqat Ahmed, Darren Andreychuk, Theodore Hellmann, Keynan Pratt, Mehrdad Nurolahzade and Seyed Mehdi Nasehi. Thank you for your time and feedback in so many discussions.

Publications from this Thesis

Portions of the materials and ideas presented in this thesis may have appeared previously in the following peer reviewed publications:

- *A Domain Specific Language to Define Gestures for Multi-Touch Applications.* Shahedul Huq Khandkar and Frank Maurer. In Proceedings of the 10th SPLASH Workshop on Domain-Specific Modeling, Reno/Tahoe, Nevada, USA, 2010.
- *Tool Support for Testing Complex Multi-Touch Gestures.* Shahedul Huq Khandkar, SM Sohan, Jonathon Sillito, Frank Maurer. In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, Saarbrücken, Germany, 2010.
- *A Language to Define Multi-Touch Interactions.* Shahedul Huq Khandkar and Frank Maurer. In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, Saarbrucken, Germany, 2010.
- *FitClipse: A Tool for Executable Acceptance Test Driven Development*. Shahedul Huq Khandkar, Yaser Ghanam, Shelly Park, Frank Maurer. In Proceedings of 10th International Conference on Agile Processes and eXtreme Programming (XP 2009), Pula, Italy, 2009.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
GDL	Gesture Definition Language
API	Application Programming Interface
DSL	Domain-Specific Language
IDE	Integrated Development Environment

Chapter One: **Introduction**

Within the domain of human computer interaction, touch has been considered an interaction approach 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. However, 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.

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.

Figure 1: Different types of multi-touch devices

1.1 Multi-Touch Application Development

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 lack in many areas that are necessary to build a reliable and robust application within an acceptable time frame that are discussed in the following sub-sections.

Figure 2 shows the steps in the life cycle of implementing a feature in a multi-touch application. Like most software development process, it starts at the design stage. Once the interaction approach is decided, the developers start implementing it. Then, different types of automated and manual testing processes are used to validate the implementation to ensure quality of the software.

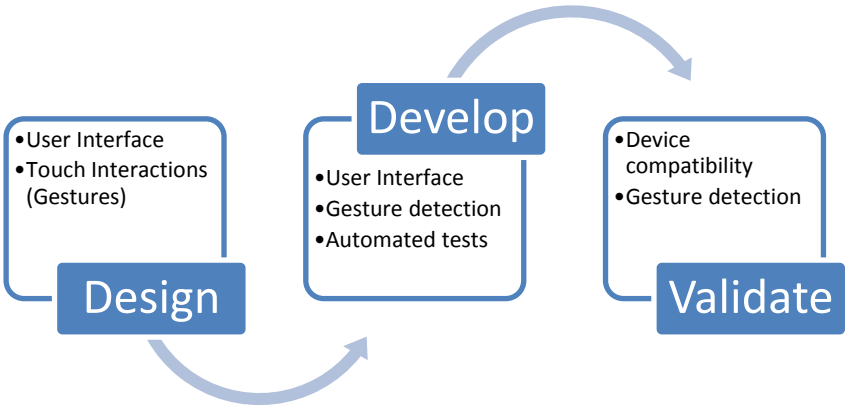


Figure 2: Implementing a feature in a multi-touch application

To understand the development process and requirements of gesture-based systems, we studied existing touch based applications, research [22] [29] on gestures for multi-touch surfaces and also interviewed experienced multi-touch application developers. The details are explained in Chapter Three. From these exploratory studies and based on our own experience, we found that developers are facing a number of challenges in developing multi-touch applications which we discuss in the following sections:

1.1.1 Tools to Define a Gesture

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 research, we primarily focus on simple as well as complex finger based touch interactions. Application developers sometimes need to develop gesture recognizers to support new gestures that are natural and meaningful to the application context. 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. It has been found that application developers and touch interaction designers are not generally domain experts in gesture recognition [31]. As gesture recognition often involves platform-specific, complex algorithms, this can represent a significant amount of work. As a result, developers often select gestures based on implementation complexity instead of usability.

1.1.2 Consistent Visual Feedback

Visual feedback is an important part for any multi-touch application. A feedback is generally provided in the form of visual effects on the touch screen in response to touch or gesture. While touch feedbacks are used for general response to any arbitrary touch, gesture feedbacks are more specific to application commands. As devices from different vendors often provide these feedbacks in different way, it becomes a challenge for application developers to maintain consistency in user interface across devices from different vendors.

1.1.3 Tool Support for Debugging

Like any other software, multi-touch applications also need to be debugged to fix a problem. However, for multi-touch applications the developers often need an actual device to do the debugging. These devices are generally expensive and often a team of multiple developers get one device to work with. Also, the physical design of the device (e.g. horizontal tabletop) is sometimes not best for long term development work. As a result, developers need to go back and forth between the device and their development computer every time they need to test a piece of code. Tools to simulate touch interactions on the development computer could reduce the need to move between devices to a great extent. A simulator can also help to simulate multi-user scenarios and therefore reduce the need to additional users to test concurrency issues.

1.1.4 Collecting Bug Reports

Sufficient technical detail is essential for fixing any software defect. For most multi-touch applications, touch is the key interaction medium. The way a person uses finger based gestures could depend on his/her background (e.g. style of the written form of their first language, left or right handed). The best way to determine why a particular touch was not recognized by the gesture recognizer is to run that same touch interaction through the step by step debugging process. But due to the lack of necessary tools the developers currently rely mostly on user comments which make the fixing process much difficult. A tool to record the interaction as part of a bug report could greatly simplify the process of fixing this sort of problems.

1.1.5 Device Independence

Multi-touch interactions [33] 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.

1.2 Research Objectives

This thesis focuses on finding the challenges of developing multi-touch applications and possible ways to reduce the complexities of development and testing touch based applications. Two of the goals of this research are:

- Design a domain-specific language to simplify the process of defining new gestures, and
- Develop a device independent application framework for multi-touch applications that supports the gesture definition language and provides:
    - consistent visual feedback across devices
    - tools for debugging touch interactions
    - framework for automated testing

To address the research goals, a gesture definition language (GDL) is designed as part of this research which allows defining multi-touch gestures including multi-user and multi-step scenarios. The details are described in Chapter four. A framework, GestureToolkit, is developed to address the research goals for reducing a number of development challenges and to implement the GDL.

1.3 Document Structure

The reminder of this document is structured in the following chapters:

Chapter Two: Related work - takes a detailed view at related fields of research and the existing work within them.

Chapter Three: Exploratory Study - describes the study used to collect the requirements including study of related research work, existing multi-touch applications and interviews with experienced multi-touch application developers.

Chapter Four: The Gesture Definition Language - describes the design and structure of the domain-specific language for defining gestures.

Chapter Five: GestureToolkit - gives an overview of the implementation details including the language design and implementation of the features of GestureToolkit.

Chapter Six: Technical Challenges - discusses the technical challenges we encountered throughout this research starting from design decisions to implementation choices and adaption of tools to ensure future sustainability of the project.

Chapter Seven: Evaluation - describes the process and results of the user study on GestureToolkit framework and the gesture definition language.

Chapter Eight: Conclusion - provides a summary of contribution, the limitations and future plan for this research and the open-source project – GestureToolkit.

Chapter Two: **Related work**

Multi-touch is a technology where both the hardware and the software platform are still evolving at a great pace. While the hardware support is essential, the right application is also a driving factor to bring meaning for these devices to general people. In this chapter, we describe the existing work on supporting tools and frameworks for multi-touch applications. The related works are categorised into two sections: a) application frameworks, and b) tools for development and testing. Application frameworks help to reduce the development complexities by abstracting the low level implementation details behind the high level application programming interfaces (API). While these frameworks also help developers build applications faster by providing reusable components, tools like device simulators can simplify the process of development and testing in many ways.

2.1 Application Frameworks

Frameworks are software libraries that provide reusable abstractions of code wrapped in a well-defined Application programming interface (API). Our GestureToolkit is a framework for multi-touch applications. It decouples the actual hardware from the application by providing a hardware abstraction layer. This layer includes a hardware agnostic interface for capturing multi-touch inputs. The framework also provides a domain-specific language to define gestures. In this section, we compare our work with existing work on hardware independence and gesture recognition systems.

2.1.1 Hardware independence

Multi-touch devices often require developers to write device specific implementations because of the differences in underlying hardware and vendor specific software development kits (SDKs). However, one possible way to achieve platform independence

is through abstracting the communication interface between the actual hardware and the application. We need this hardware independence to reuse the same multi-touch applications across different devices.

While the tabletop hardware vendors provide tool support for the development and testing of tabletop applications specific to their device, they are not interoperable; as a result the applications developed using these SDKs cannot be readily used on other platforms. For example, Microsoft Surface provides an SDK for developers to simplify the touch-related development complexities like managing concurrent touch points, touch friendly widgets, components to detect special tags, and the like. Similarly, SMART Technologies provides an SDK for their multi-touch devices. However, the widgets and other features provided by these SDKs only work on their devices.

Echtler [8] provided an abstract architecture design and an implementation thereof to improve the interoperability of multi-touch applications among various devices. It has an interpretation layer that decouples the hardware specific inputs from the software application code. Several other projects also provide tool support for abstracting multi-touch interactions. For an example, PyMT [10] is a framework for rapid prototype development of multi-touch applications. It provides consistent low level touch data to the application widget layer from different touch devices. Pointer [11] also proposed a hardware abstraction approach. Touchlib [9] is a library for capturing images and processing data from frustrated total internal reflection (FTIR) based devices. It provides basic events like finger down, finger up, and finger move.

In GestureToolkit, we have a similar approach as PyMT and Pointer in the device abstraction part. The hardware abstraction layer provides an extensibility framework that

allows adding support for new devices. Applications developed on top of GestureToolkit will run on these new devices without any change in the application code. GestureToolkit also allows connecting multiple devices at the same time including virtual devices to simulate touch interactions.

2.1.2 Gesture Recognition

The motion of meaningful touches to interact with the system is called a gesture. A gesture may include touches of multiple fingers, hands and arbitrary tangible objects. Recent versions of the widely used operating systems are providing native support for touch based gestures. For an example, Microsoft Windows 7 supports Zoom, Pinch, Rotate and some other gestures out of the box. Mac OS Snow Leopard also provides gesture support to some extent. However, this operating system level support can only be utilized if there is a device driver, which is not yet available for all touch enabled devices. For an instance, to use the Windows 7 Touch API on Microsoft Surface one will need to write a device driver, as it is not available yet. Also, developers need to handle operating system specific differences for cross platform applications. SDKs for specialized multi-touch devices like the Microsoft Surface SDK [39] provides gesture recognition for a small set of gestures; however, the gestures are hardcoded into the system and are often embedded inside widgets. Therefore, developers need to write recognition algorithms from scratch to implement any new gesture.

The challenge of implementing gesture recognition system is a well-known problem in the research community. Significant amount of research has already been done in the area and also researchers are actively working on improving gesture recognition systems. We discuss them in the following sections.

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. This approach, however, has 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. Also, it cannot recognize gestures that are based on rules on touch movements instead of fixed touch patterns (e.g. rectangle or circular shaped stroke) saved in templates.

Kartz [13] 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 suffers from limitations like smaller gesture vocabulary size and it cannot process gestures with continuous motion.

The Hidden Markov Model (HMM) is a statistical model that is often used in sketch recognition. Sezgin [14] proposed an HMM-based sketch recognition system that was motivated by static and dynamic characteristics of sketches. Cao [15] also used a HMM to present an evaluation of a hybrid gesture interface framework that combines online adaptive gesture recognition with a command predictor. Anderson [16] proposed sketch-based symbol recognition using a HMM, but false positives are an unavoidable aspect of this approach.

Interaction designers are not generally domain experts in gesture recognition [42]. To simplify the process, there are a number of frameworks and toolkits available for pattern and gesture recognition, such as Weka [43] and GT2K [44]. While these are mostly libraries of techniques, tools are also available for designing gestures such as MAGIC [45] and quill [46]. MAGIC uses recorded data as samples and quill also uses recorded

data for training purpose. Although the recording feature greatly simplifies this for a set of single touch gestures, multi-touch gesture are often based on certain gestural condition (e.g. lasso gesture, five-finger-select and drag) that are not possible to define by sample dataset.

To address the limitations of existing gesture recognition systems, GestureToolkit provides a gesture definition language that includes multi-user, multi-touch and multi-step gestures. This also allows the developers to easily test different steps of the gesture recognition process and edit them as necessary.

2.2 Tools for Development and Testing Touch Interactions

The right set of tools is essential to ensure the quality of the software developed within a reasonable amount of time. Unlike traditional software that runs on standard computers, multi-touch applications are generally developed for special devices. However, developers mostly use their desktop computers to develop these applications which generally don't have the touch capability. As a result, to test the application being developed, they need to move between the development machine and the actual device back and forth. Also, manually testing the features every time something has been changed is both time consuming and resource intensive. Device simulators can provide the option to test multi-touch applications from non-touch enabled systems to a great extent and tools to automate the testing process can also help to ensure quality of the software. We discuss the existing work on these related areas in the following sub-sections.

2.2.1 Device Simulators

The Microsoft Surface SDK provides a record and playback tool that allows developers to test their applications using recorded touch interactions. However, this tool can only be used for applications that are built using the Microsoft Surface SDK as it only works inside their simulator or on an actual Microsoft Surface device. Although the tool provides a recording feature that helps the manual tests to some extent, it does not provide any support for using those recordings in automated tests. Pointer [11] also proposed a record and replay based automated testing approach.

DART [47] is a tool that uses the capture/replay concept to simplify the process of working with augmented reality. It allows designers to specify complex relationships between the physical and virtual world and allows designers to capture and replay synchronized video and sensor data to work off-site and to test specific parts of their experience more effectively. FauxPut [48] is another testing tool for interaction designers that can wrap input device APIs and provide an interface for recording, simulating and editing inputs for recognition based interactions. It also allows creating simulation of sensor data along with other actual device data in parallel.

Mouse 2.0 [5] or toolkits like Multi-Mice [6] and Multi-Touch Vista [1] can add the ability to use multiple pointers in regular computers to simulate touch points to some extent. However, a problem with this approach is that you can only simulate two moving touch points at a time through mice. OpenInterface [56] also provides a similar environment to work with simulated components from a component repository (e.g. speech recognition, video ``finger tracker'').

GestureToolkit follows a similar approach with an extension that it allows developers to debug and write automated tests of applications independent of the underlying hardware. This also allows simulating multi-user scenarios using multiple recorded interactions and helps to overcome the need of an actual device to a great extent.

2.2.2 Test tools

Although not directly applicable to tabletop applications, there is tool support available for automated testing of traditional mouse and keyboard based user interfaces (UI). For example, CodedUI Test [49], Project White [50], Selenium [51] and QFTest [52] are used to automate UI testing of regular desktop and web applications. Some of these tools follow a record and replay based test automation while others rely on a programmatic approach only. Although these tools and most other UI testing tools can automate the UI events from mouse and keyboard, we haven't seen a test automation tool that works for touch inputs even though the underling operating system (i.e. Windows 7) natively provides the support.

While large scale multi-touch devices are fairly new and still mostly used for research purposes, smaller handheld multi-touch devices like smart phones and other portable devices are quite common to general people. Froglogic [53] is working on Squish - an automated graphical user interface (GUI) testing tool for different platforms including Apple's iPhone and iPad to support the testing of Cocoa Touch [54] applications. Vimov [55] provides another multi-touch testing tool for iPhone and iPad applications. It can simulate device features through another device like using an iPhone as a multi-touch controller for Apple's iPad simulator. Although these tools help the testing of handheld

multi-touch devices, we cannot use them for automated testing of large tabletop interfaces.

However, GestureToolkit also has a virtual hardware simulator similar to OpenInterface to simulate actual device inputs for testing and debugging multi-touch applications. This also provides a unique feature for continuous integration (CI) systems to run automated test scripts to validate gesture detection without actually running the application on a physical device or simulator.

Chapter Three: **Exploratory Study**

An exploratory study was conducted to assess the problem domain. This study consists of three sections: 1) review existing research work in academia and industry, 2) study some existing touch based applications to understand their requirements, and 3) conduct a semi-structured open-ended interview with experienced multi-touch application developers. These three sections are described in the subsequent sections. The result of this study is used to set the requirements of GestureToolkit.

3.1 Review existing research in academia and industry

Research labs in universities and software industries are actively investigating possible interaction techniques for multi-touch surfaces. To better understand the requirements of multi-touch applications, we studied the results of C. North's [22] and J. O. Webbrock's [29] research on multi-touch manipulation and user defined gestures for surface computing. Then we compared the list of useful gestures with the supported gestures from some of the popular multi-touch application frameworks like Windows Presentation Foundation (WPF), Microsoft Surface SDK and GestureWorks – a multi-touch framework for Flash based applications. Table 1 shows the summary of our findings.

Table 1: Gesture Support in Different Application Frameworks

Action	Gesture	Multi-touch application frameworks		
		WPF 4.0	Microsoft Surface SDK	GestureWorks
Select	Tap	*	*	*
	Lasso			

	5-finger select			
	Tap and hold	*	*	*
Rotate	2 finger rotate	*		*
	1 finger rotate		*	
Move	Drag	*	*	*
	Jump			
	Drag hand			
	Drag corner		*	
	One hand shove			
Scroll	Two finger scroll			*
	One finger scroll	*		
Cut	Slash			
Duplicate	Tap source and destination			
Delete	Drag off screen	*	*	*
accept	Draw check			
reject	Draw ‘X’			
Undo	Scratch out	*		
Enlarge	Pull apart with hands			

	Pull apart with finger	*	*	*
	Pinch	*	*	*
	Spray fingers			
Open	Double tap	*	*	*

From the comparison, we found that many of the useful gestures are not support by the frameworks out of the box. While some of these gestures may be useful for few specific types of applications, but when application developers do need to recognize these gestures they often either end up implementing them from scratch or choose to use other alternative gestures and therefore compromise on application usability. In one of the case studies (i.e. AgilePlanner), we have seen that the developer didn't provide the gesture that would be most meaningful and natural to the application context due to lack of gesture recognition support from the framework. We discuss these case studies in the following section.

3.2 Study existing touch based applications

We choose three touch based applications that use touch as a primary input system. In all three cases, the predefined gestures that are available in Windows Presentation Foundation were not sufficient. The developers had to implement gesture recognition modules to support new gestures that are most appropriate to the application context. These applications are: 1) SmartUML, 2) Agile Planner and 3) eGrid. The details of each application are described in the following sections:

3.2.1 SmartUML

SmartUML [3] is a free-hand sketch-enabled multi-user UML Diagram designer. It offers natural freehand drawing with pen interface and on-the-fly drawing detection. The tool currently supports Use Case, Class Diagram and Activity Diagram.

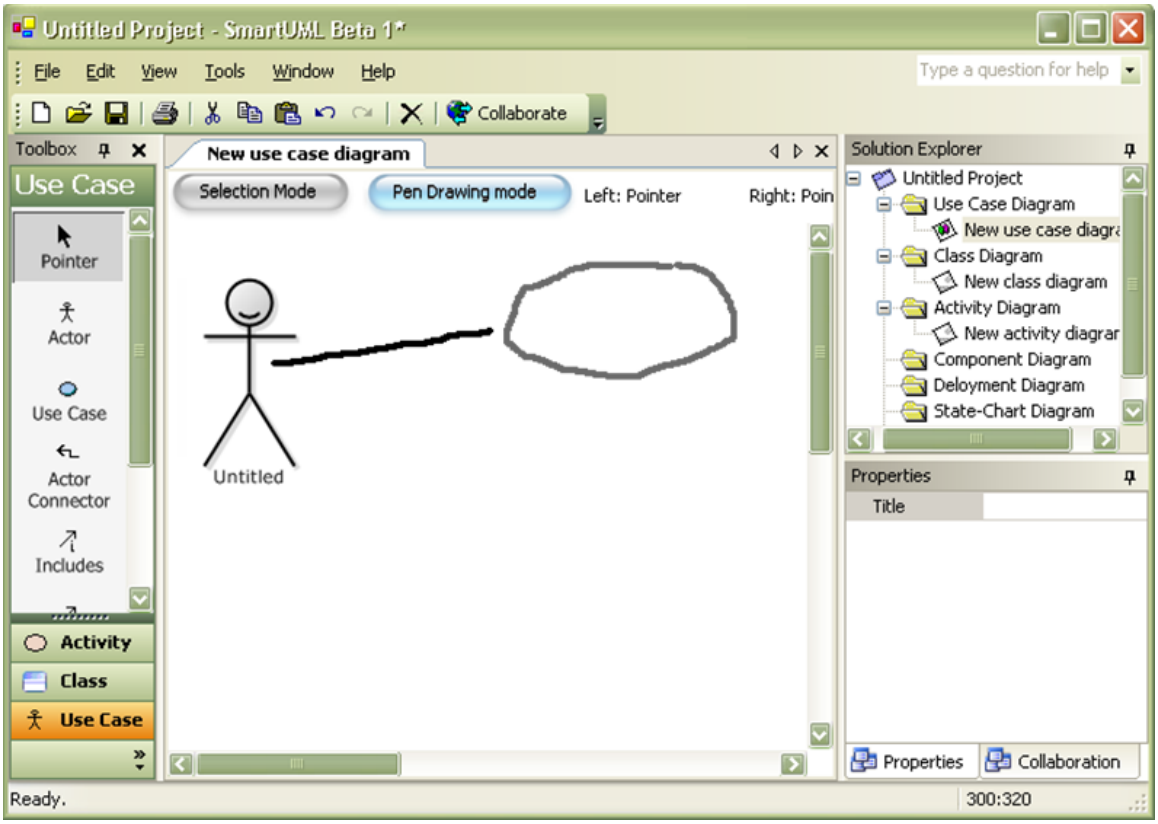


Figure 3: SmartUML - a free-hand sketch-enabled multi-user UML designer

SmartUML is an open source project hosted at sourceforge.net. It uses custom gesture recognition algorithms which includes detecting various geometric shapes and intersections of multiple shapes in a certain logical sequence. After successful gesture detection, the application layer requires the position, size and often the bounding box of the gesture to place the appropriate object on the screen.

So, a gesture definition language that allows defining touch interactions that represents different geometric shapes in a certain structure could have simplified the development process to a great extent.

3.2.2 AgilePlanner

AgilePlanner [2] is a rich client based on the .NET/WPF framework that supports vertical displays as well as digital tabletops. It supports synchronous distributed planning meetings by providing a shared workspace for creating, organizing and editing electronic index cards. Changes made by one team member become visible immediately on connected clients all over the world.

Figure 4: AgilePlanner on a large horizontal display

While the developer used a custom gesture recognizer to detect straight lines, for the rest of the touch interactions he chose to use the predefined gestures available in the Windows Presentation Framework. As a result, the application did not provide the gestures that would be most useful in some cases. For example, to move tasks (the red and yellow rectangles) to a specific iteration (the large blue rectangle) a user has to drag one object at a time. The user can at most drag two items in parallel as the device supports maximum

two concurrent touch points. However, with the ability to easily define new gestures, the developer could incorporate the lasso gesture to select multiple tasks with one gesture and move all selected tasks to the desired iteration using one more touch.

3.2.3 eGrid

eGrid is a collaborative application for utility companies to faciltate the collaboration of control center team members in their daily tasks. In addition to application specific gestures, another requirement of eGrid was to support multiple hardware platforms. The first version of the application was developed using Microsoft Surface SDK. As a result, it was too dependent on Microsoft Surface and it could not support other devices like Dell XT tablets or the SMART tabletop which was also part of the requirement. Some other differences between these devices include inconsistent visual feedback. For example, Microsoft Surface provides different visual effects than Dell XT (Windows 7 based system) in response to touch input; and SMART tabletop leaves this for the application and does not provide any visual feedback.

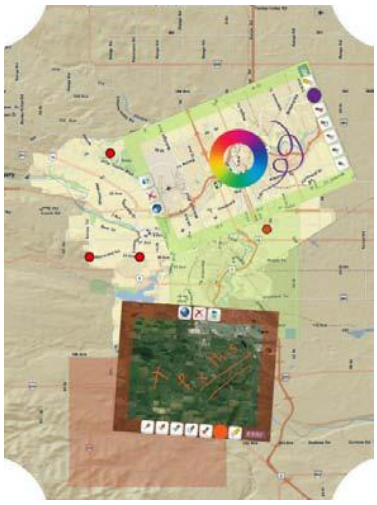


Figure 5: eGrid - a collaborative application for utility companies

The second version of eGrid is currently being developed and is using the GestureToolkit framework; it now supports multiple devices including Microsoft Surface and Dell XT2 tablets. It also uses the GDL to define custom gestures (e.g. lasso).

3.3 Exploratory user study on experienced multi-touch application developers

We interviewed 3 participants who developed tabletop applications in a university lab environment. All participants had prior software development experience and more than one year of tabletop application development experience. The participants were from the same lab where the toolkit was developed but they had not used the toolkit before this study. We refer to those participants as P1, P2 and P3. Table 2 provides a summary of their experience levels.

Table 2: Participants' Experience

Participant	**# of Tabletop Apps.**	**Years of Tabletop Experience**	**Years of Development Experience**
P1	2	1	5
P2	3	2	8
P3	1	1	3

P1 developed GIS-based tabletop applications with an industry partner. P2 developed a multi-player table-based game and P3 developed and maintained an existing collaborative tabletop application. Both P1 and P2 developed software for the Microsoft Surface and P3 developed software for SMART tables.

3.3.1 Data Collection and Analysis

Each participant was interviewed independently for 25 minutes. The interviews were semi-structured and organized around three main topics. During the interviews each topic was introduced using starter questions:

- Please tell me how you tested your application.
- Is there anything that was difficult to test?
- How did you test multi-user scenarios?

Each interview was audio-recorded and transcribed for analysis. Our analysis involved two stages. In the first stage we performed open coding on the transcribed data. Open coding is an analytic process to identify concepts in the collected data [35]. In the second stage of our analysis, we grouped the coding into five categories that capture the main challenges our participants faced.

3.3.2 Findings

The following discussion of our findings is organized around the five categories that emerged as we analyzed our study data.

Figure 6 shows the tabletop application testing workflow. This example workflow demonstrates that the developers carry out their debugging at two different locations, i) at their workstations using the simulator and ii) at the actual table (the shaded region in the figure). This process is described by P1 in the following response:

"I usually used the simulator to test only the initial test to see how it looks like. Then I had to move it to the actual hardware and then test it because the experience is much different".

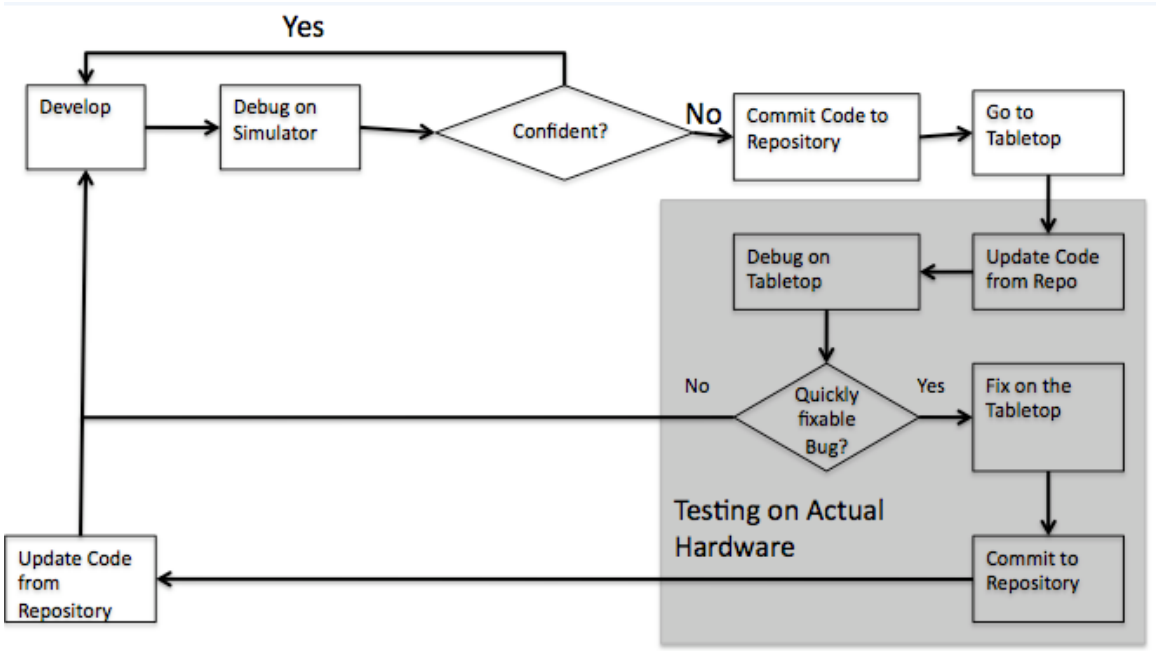


Figure 6: An example of a tabletop application testing workflow

This workflow indicates that the testing and debugging effort is increased when working on tabletop applications because the developers need to move between their workstation and the actual hardware and perform repetitive testing.

3.3.2.1 Testing Approach

Although all participants of this study used automated unit tests to automatically verify their application logic, none of them used any automation for testing the tabletop interfaces. In fact, none of the participants were even aware of any automated testing tools. As a result they spent a considerable amount of time on manual regression testing, which involves carrying out the same tests over and over again. This was particularly time consuming for participant P3 who was developing an application for two different tabletop devices with different physical sizes. So, P3 had to manually test on both tables whenever there was a significant change in the application.

3.3.2.2 Limitations of the Simulator

Tabletop hardware vendors often ship device simulators. Although these simulators can mimic the hardware on a standard PC to some extent, the developers still run into issues as a result of differences between the simulator and the actual tabletop. For example, in response to a question on the difference between testing alone at the workstation and with multiple users at the tabletop hardware, participant P1 mentioned the following:

"... if you are trying to create a new window, you can't do it more than once at the same time because you have only two hands (two mice at the simulator). So if two people are trying to test at the same time (on the actual hardware) maybe they will check occurrences like doing this at exactly the same time."

Table 3: Feature comparison between a Simulator and an Actual Tabletop

Feature	**Microsoft Surface Simulator**	**Microsoft Surface**
# of touches	# of Mice	52+
Physical objects	Limited	Almost any shape
Sensitivity	Mouse is very Precise (300-800 DPI)	*fat-finger* Finger is less Precise
# of Testers	# of Mice	More than one
Physical orientation	Vertical	Horizontal

Table 3 summarizes the key differences between these two environments (in this case the Microsoft Surface Computer and the Microsoft Surface Simulator). From the above table

we see that the simulator supports a limited capability multi-touch and multi-user environment compared to the target tabletop. As a result, a significant amount of testing and debugging work needs to be carried out on the actual table, especially when complex concurrent interactions need to be considered.

3.3.2.3 Testing Multi-User Scenarios

Multi-user scenarios typically involve a large number of possible concurrent interactions by different users on the same interface. Manually testing such interfaces require multiple users, which is an often difficult to find every time a feature needs to be tested.

For an example, P3 mentioned a multi-user scenario that he developed where multiple users could vote by placing a tap gesture on a specific interface element. He prepared the test plan to test for the following scenarios: 1) single user votes, 2) multiple users vote sequentially and 3) multiple users vote concurrently. However, multi-user interactions can go beyond a single interaction on a single element. In that situation, manual testing becomes even harder as there is an explosion of possible states.

Multi-user scenarios often introduce unseen performance issues as well. P2 and P3 mentioned that at times they experienced severe performance degradation when multiple users were concurrently using their systems. But a single developer or tester, when doing manual testing can only explore a limited set of possible concurrent scenarios.

3.3.2.4 Bringing Code to the Tabletop

In most development teams that our participants worked in, digital tables are shared by multiple developers. As a result, developers typically need to move code between their PC and the shared tabletop so that they can test the features in the target environment. Our study participants use source code repositories or USB memory sticks as

intermediate storage between the two environments. This process of going through an intermediate medium slows down the familiar workflow of the develop-debug-develop cycle. Also, it requires developers to commit untested code to the shared repository, which often breaks a working build. As P1 mentioned:

(The process of transferring code to the table) is not comfortable because sometimes you make some changes but you are not confident to commit it, as it's not a final change.

Participants P1 and P3 mentioned that developing on the tabletop with an additional vertical display was faster as the outcome of the work could be loaded and debugged immediately. To boost productivity, we recognize that it is important to provide developers with tools so that they can get immediate feedback about their work-in-progress code.

3.4 Summary of Exploratory Study

Based on the results of comparison between the list of useful gestures for multi-touch surfaces [22] [29] and predefined gestures available in existing multi-touch SDKs, we see that a significant number of gestures are not available out of the box. However, the requirements of gestures also depend on the application context. As there is a wide range of possibilities for multi-touch applications, it may not be practical for a framework to provide every single gesture predefined out of the box. Instead, a domain-specific language to define custom gestures is a more appropriate solution. Our study on several multi-touch applications (section 3.2) also supports the fact that applications often need gestures that are not available in existing SDKs. It also suggests that developers often need to build applications for devices that provide similar feature but may come from different vendors with minor variations. As a result, developers need an application

platform that would work across multiple devices and provide consistent visual feedbacks.

The findings of semi-structured interview with experienced multi-touch application developers show that tools for automated testing and simulating touch interactions including multi-user scenarios could significantly simplify the development process. Based on our own experience, we believe a tool to capture touch interactions and later use them in debugging would also help developers fix application defects.

Chapter Four: **The Gesture Definition Language**

The results of the exploratory study shows that developers often need to define new gestures that are not available in existing application frameworks out of the box. To simplify the process of defining new gestures, we present the gesture definition language (GDL) that hides low level implementation complexities from application developers without compromising the flexibility of gesture definitions. The design of the language focuses on the following four goals:

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

GestureToolkit, the underling framework that compiles and executes the gesture definitions defined using GDL, decouples the hardware specific issues from rest of the system. The internal design and implementation details of GestureToolkit are described in the next Chapter. We discuss the first three objectives of the language and how they are implemented in the following sections.

4.1 The Objectives of GDL

GDL is a domain-specific language designed to streamline the process of defining gestures. In this section, we describe the objectives of the gesture definition language as mentioned above.

4.1.1 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 often also need to write the gesture recognition modules that recognize a gesture from raw touch data. Gesture recognition 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 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 defining gestures can hide the low level implementation complexities by encapsulating complex mathematical calculations, pattern recognition algorithms and the like. This can help the developers focus on designing the gesture at a higher level without worrying about the implementation details.

4.1.2 Flexibility

A specially designed DSL for gestures can help developers focus on application design instead of low-level gesture 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.

4.1.3 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" [32], 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.

4.2 Implementation of GDL

The objective of the language is to provide a high level framework for application developers to define new gestures that will hide the low level implementation details without compromising the flexibility of gesture. The development process can be divided into following four parts:

(1) design of the language,

(2) the language parser,

(3) execution process, and

(4) support for integrated development environment (IDE).

The subsequent sections describe each of these key parts in detail.

4.2.1 Language Design

Figure 7 shows the structure of a gesture definition in GDL. The 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 7: The structure of a gesture definition

The *name* must be unique within the scope of the application. GDL is part of GestureToolkit 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. The full list of predefined gestures available in GestureToolkit is listed in Appendix A. 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 logic is defined using a combination of primitive conditions, the smallest units to evaluate raw data provided by the hardware abstraction layer.

Primitive conditions can be of different types. Table 4 shows some examples of primitive conditions that can be used to define a pattern of touch points movement (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). There are currently 11 primitive conditions available out of the box. Table 5 shows a detailed

list all primitive conditions available in GestureToolkit. Developers can also create their own primitive condition (which is described in section 4.2.2) and thereby extend the GDL based on their requirements.

Table 4: Example of primitive conditions

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

Multiple primitive conditions are virtually connected like a chain using the logical operators (i.e. and). The validation process follows a 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.

Table 5: List of primitive conditions available in GestureToolkit

Primitive Conditions	**Syntax**	**Description**
Closed loop	`Closed loop`	Returns the set of touches that represents a closed loop.
	`Not closed loop`	
Distance	`Distance between`	Returns the set of touches where the

between points	`points: 1..10`	distance between touch points are within the specified range
	`Distance between points: unchanged 10%`	Returns the set of touches where the distance between touch points are unchanged to a give threshold. For example, in this case 10% change in distance is acceptable.
	`Distance between points: increasing`	Returns the set of touches where the change of distance between touch points is following a pattern. The possible patterns are: increasing and decreasing.
Enclosed area	`Enclosed area: 100..300`	Returns the set of touches where the enclosed area of the touch-paths is within the specified range. The default unit is in pixel.
On same object	`On same object`	Returns the set of touches that are on the same object.
Touch area	`Touch area: Rect 50x50`	Returns the set of touches that are within the specified area. The default unit is in pixel. The shapes of the area can be of three types: Rectangle (Rect), Circle or Ellipse
Touch path length	`Touch path length: 100..200`	Returns the set of touches where the length of the touch path is within the limit. The default unit is in pixel.
Touch shape	`Touch shape: Line`	Returns the set of touches that represents the specified shape It currently supports line, rectangle and circle shape.

Touch step	`Touch step: 2 touches within 1 sec`	Returns the set of touches that occurred within the specified time window. The default unit is in pixel. The primitive condition supports both second and millisecond units.
Touch limit	`Touch limit: 1..4`	Returns the set of touches within specified limit. In case of more active touches, it will return different combination of touch points with the size specified.
Perpendicular to	`L1 perpendicularTo L2`	Returns the set of touches if they intersect perpendicularly.

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. Table 6 shows the list of all available return types in GestureToolkit out of the box.

Table 6: List of return types available in GestureToolkit

Return type	**Description**
Bounding box	Returns the smallest rectangle (aligned to the X and Y axis) that can bound the selected touches
Distance changed	Returns the amount of distance changed since last event

Info	Returns the message specified in definition
Slope changed	Returns the angle of slope changed (in degree)
Touch actions	Returns the last touch actions of selected touches
Touch Ids	Returns the unique identifies of the selected touches
Touch paths	Returns the paths (as an array of points) of the selected touches
Touch points	Returns the last points of the selected touches

We now describe how GDL addresses some of the key issues of multi-touch application development including defining new gestures and extending the language to support additional features.

4.2.1.1 Hiding Low Level Complexities

Let's consider a scenario where a user may use a lasso gesture (Figure 8) to select some objects from a scattered collection of objects.

Figure 8: 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 9 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.

```
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 9: Defining the lasso gesture using GDL

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.

4.2.1.2 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 78[3]) the user would do the following touches to create an "Actor" object.

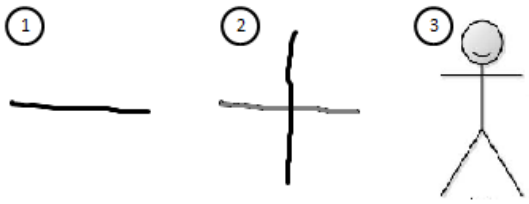


Figure 10: Sequence of touch strokes to create an "Actor"

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. These multi-touch scenarios may involve multiple users and some of these partially validated results could end up representing different gestures too.

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 preceding code snippet (Figure 11) defines the actor gesture. 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.

```
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 11: Defining Actor gesture for Use Case diagram

4.2.1.3 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 [19]. Now some devices can provide touch directions (e.g. Microsoft Surface) and information about the pressure of a touch [21]. The extensibility framework of GDL allows creating new primitive conditions as well as return types to extend the language with additional recognition algorithms and device features.

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 12 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 12: The code snippet of GDL grammar for "Touch Step" primitive condition

Then, create a validator that takes raw touch data as input and does the validation. A class implementing the `IPrimitiveConditionValidator` interface written in any .NET supported language can contain the computation logics.

```
public class TouchStateValidator : IPrimitiveConditionValidator
{
    ...
    public void Init(IPrimitiveConditionData ruleData)
    {
        _data = ruleData as TouchState;
        if (_data != null)
            foreach (string touchState in _data.States)
                _requiredTouchStates.Add( new TouchActionResult()
                    {
                        Action = GetTouchActionType(touchState)
                    });
    }

    public ValidSetOfPointsCollection Validate(List<TouchPoint2> points)
    {
        var sets = new ValidSetOfPointsCollection();
        if (IsValid(points))
            sets.Add(new ValidSetOfTouchPoints(points));

        return sets;
    }

    private bool IsValid(IList<TouchPoint2> points)...
}
```

Figure 13: Code snippet for the TouchState primitive condition validator

Figure 13 shows a code snippet for the TouchState. The `Init()` and `Validate()` are the most important methods that need to be implemented for any new primitive type. The

gesture processor sends any parameters specified in the gesture definition to the `Init` method when the class is instantiated at runtime. It is this methods responsibility to persist the data for later use as necessary. Then the gesture processor sends the raw touch data to the `Validate` method for processing and also captures the return values from this method to pass the valid set of touch points to the next primitive condition as specified in the gesture definition.

Similar to primitive conditions, the return types can also be added in the language. Figure 14 shows the GDL grammar that is responsible for parsing the return types from the gesture definition. The highlighted area in the figure shows how the "Slope changed" return type is defined in the grammar.

```
token ReturnTypes
    = r: "Distance changed" => r
    | r: "Slope changed" => r
    | r: "Acceleration" => r
    | r: "Distance" => r
    | r: "Position" => r
    | r: "Position changed" => r
    | r: "Pressure" => r
    | r: "Direction" => r
    | r: "Enclosed path" => r
    | r: "Touch points" => r
    | r: "TouchID" => r
    | r: "Info" ":" v:ValidName=>r+":"+v;
```

Figure 14: A code snippet of the GDL grammar to parse the return types

Each return type refers to two classes defined in framework. These classes can also be defined in any of the .NET supported languages as long as they implement the required interfaces. First, `SlopeChanged` is the class which carries the calculated values for the return type. Client applications receive an instance of this class when the appropriate gesture is invoked.

```
public class SlopeChanged : IReturnType
{
    /// <summary>
    /// New slope value in degree
    /// </summary>
    public double NewSlope { get; set; }

    /// <summary>
    /// The amount of slope changed (in degree)
    /// </summary>
    public double Delta { get; set; }
}
```

Figure 15: Data container class for SlopeChanged return type

Next is the calculator class which is responsible for doing the calculation using the final set of touch points. It stores the result into the data class as mentioned above and then passes the data object to gesture processor. Figure 16 shows the calculator class for the SlopeChanged return type.

```
using TCH = TrigonometricCalculationHelper;
public class SlopeChangedCalculator : IReturnTypeCalculator
{
    public IReturnType Calculate(ValidSetOfTouchPoints set)
    {
        ...
        // Check if enough history data is available
        var lenOfS1 = set[0].Stroke.StylusPoints.Count;
        var lenOfS2 = set[1].Stroke.StylusPoints.Count;
        if (lenOfS1 > 1 && lenOfS2 > 1)
        {
            // Calculate slope for last position
            var prevSlope = TCH.GetSlopeBetweenPoints(
                set[0].Stroke.StylusPoints[lenOfS1 - 2],
                set[1].Stroke.StylusPoints[lenOfS2 - 2]) * 180 / 3.14;

            sc.Delta = sc.NewSlope - prevSlope;
        }

        return sc;
    }
}
```

Figure 16: The code snippet of the SlopeChanged calculator class

4.2.2 Language Parser

The language parser uses an MGrammar [41] compiler to parse and build the abstract syntax tree (AST) from the user-defined GDL. Developers can use it from the command line or Visual Studio extension so that whenever the application is compiled, the language parser will also run and compile the gesture definitions. If an error occurs, a notification is provided via a console message.

The parser uses the API provided by Microsoft to parse and generate the AST from gesture definitions using the syntax rules defined in the language grammar. Next, the MGraphXamlReader [34] library dynamically instantiates the .NET classes to build the object model from AST nodes. Then, the object model is serialized in java script object notation (JSON) format and saved into the application deployment directory as an embedded resource so that the framework can directly load gesture definitions at runtime. As this process uses the precompiled objects of gestures definitions, it saves the compilation of gesture definitions during application initialization and improves the application loading time.

4.2.3 Gesture Validation Process

Figure 17 shows a high level 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.

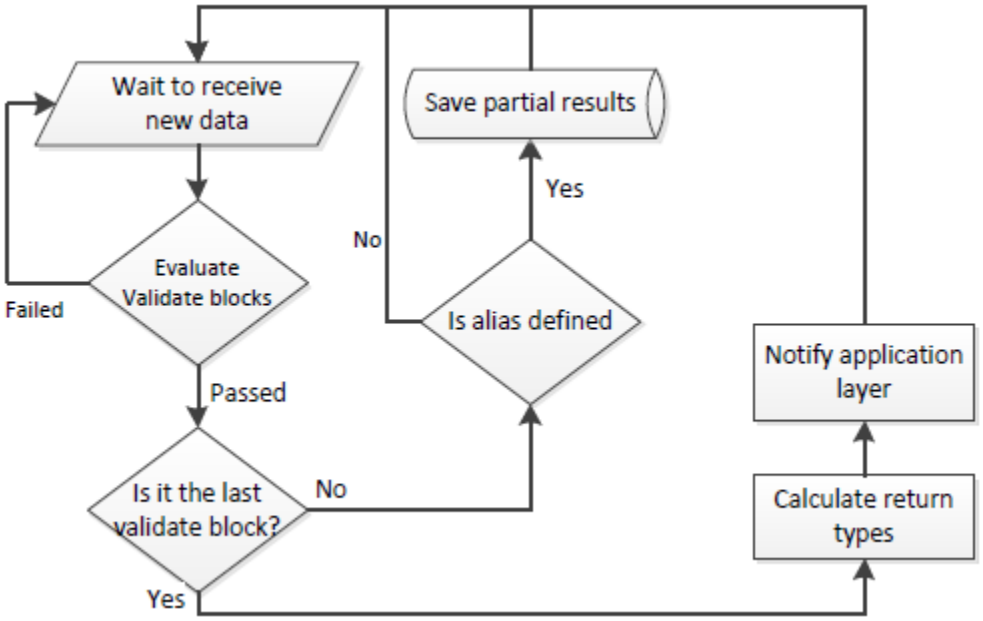


Figure 17: The workflow of gesture recognition

A validate block consists of one or more primitive conditions connected to each other like a chain. 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.

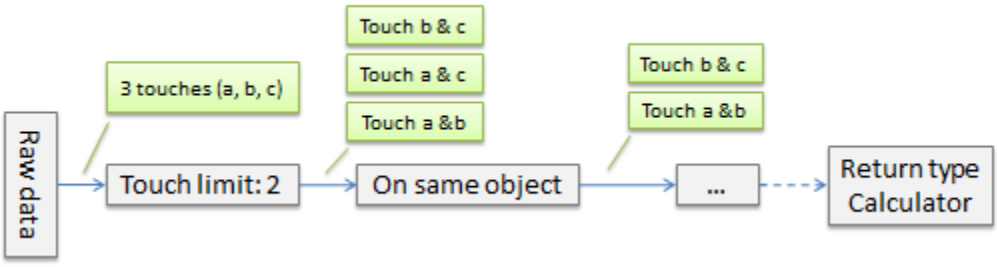


Figure 18: The execution process of primitive types

Figure 18 shows the execution process of primitive conditions in each validate block where the process starts from the first primitive condition which receives the raw touch

data from provided by the framework. The primitive condition then processes the data and passes only the valid set or sets of touch points to the next primitive condition. In the above figure, the *touch limit* primitive condition receives the three active touch points. Since its internal rule defines that the particular gesture requires two touch points, it creates new datasets from the raw touch data which is three sets (touch point b & c, a & c and a & b) and passed to the next primitive condition. In the same way, the *on same object* primitive condition found that only two of the three sets of data is valid according to its rule. So, it passes only those two sets of touch points to the next one. Finally, the last primitive condition will pass the final dataset to a gesture validator which will then send the data for calculating return types.

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.

4.2.4 Integrated Development Environment (IDE)

The exploratory study and preliminary evolution shows that application developers expected deep IDE integration for the gesture definition language including integrated compilation, syntax highlighting and on-the-fly error tracking. To address these issues, we developed extensions for Visual Studio 2010 to support integrated compilation and syntax highlighting. We selected Visual Studio as it is the most popular IDE to application developers for any .NET based development.

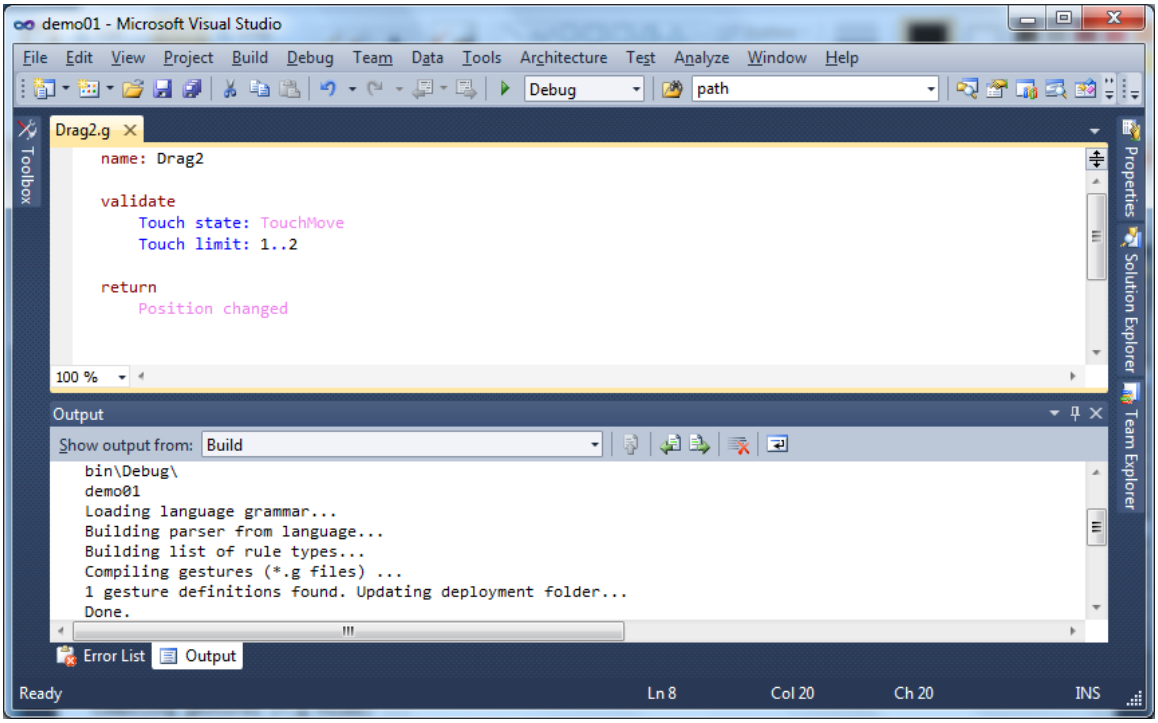


Figure 19: Syntax highlighting and integrated compilation support for GDL in Visual Studio

While writing new gesture definitions is one of the most frequent tasks related to multi-touch application development, there are times when the current set of primitive conditions and return types available in the framework is not sufficient to define the gesture that is most meaningful to the application context. In such case, developers may have to create additional primitive conditions and return types. Having this in mind, the Visual Studio Extension for GestureToolkit provides the necessary infrastructure to add additional primitive conditions & return types from the same Visual Studio project.

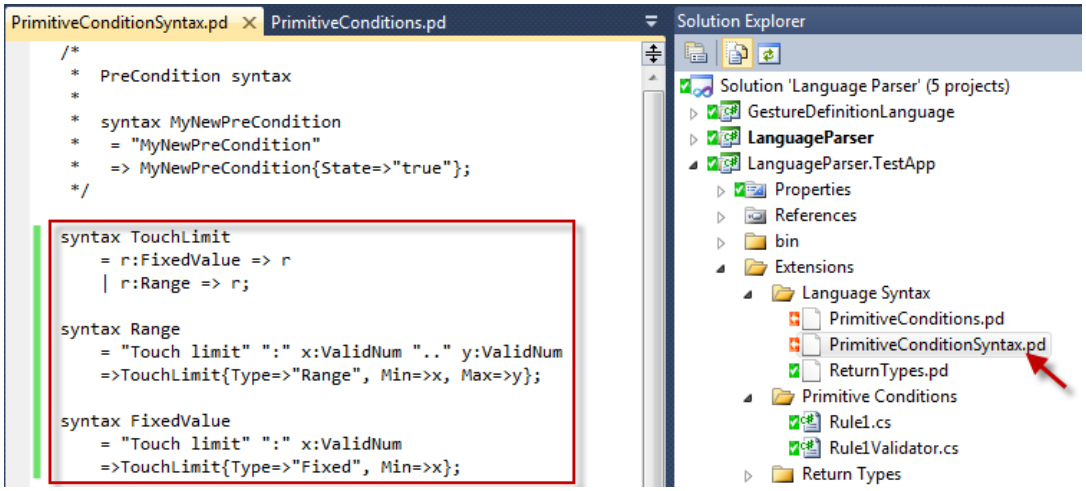


Figure 20: Extending GDL from Visual Studio

The Visual Studio Extension for GestureToolkit also provides project and item templates to simplify the process even further. At present, templates are available for defining gestures, primitive conditions, return types, automated tests. In addition to individual item templates, the extension also includes project templates for Windows 7, Microsoft Surface, SMART Tabletop, Silverlight and TUIO based applications.

While the extension in Visual Studio currently don't support on-the-fly error tracking (e.g. red squiggly lines), the same functionality is available when the IntelliPad [60] is used.

IntelliPad is a free IDE from Microsoft and it supports syntax highlighting and on-the-fly error tracking when MGrammar is used to parse the language. Figure 21 shows an example of syntax error highlighted using red squiggly mark in the IntelliPad IDE.

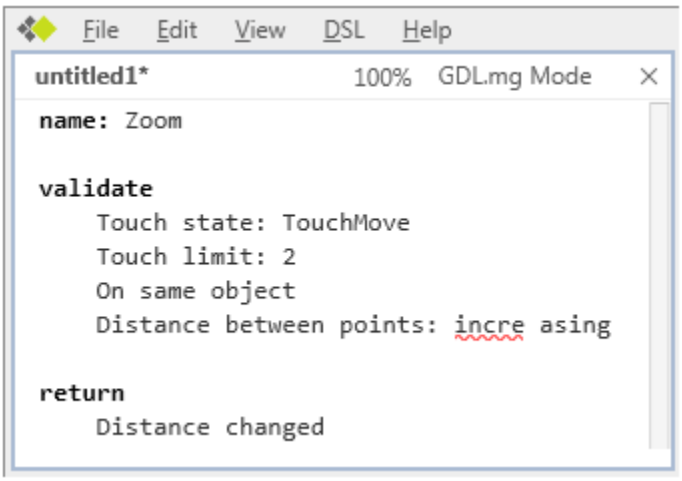


Figure 21: On-the-fly error detection in IntelliPad IDE

4.3 Summary

This chapter describes the design details of the gesture definition language (GDL). It also explains the process of extending the gesture definition language and the related tools that can be used to add additional primitive conditions and return types. Then, the internal process of validating gesture definitions is explained. Finally, The extensions of popular integrated developer environments (IDEs) that helps develops to write gesture definitions like Visual Studio and IntelliPad are described with examples.

Chapter Five: **GestureToolkit**

In the exploratory study, four major challenges were addressed: device independence, gesture definition support, device simulator and proper test framework. To address these challenges, GestureToolkit – a multi-touch application framework was developed. In addition to providing tools to compile and execute the gesture definition language described in Chapter Four, GestureToolkit also aims to address the following development and testing challenges:

1) a hardware abstraction layer that separates the application from the device hardware,
2) tools to compile and execute the gesture definition language,
3) visual feedback framework for touch interactions,
4) device simulator that provides a number of features including a way to debug and test touch interactions without an actual device, and
5) an automated test framework that allows to write automated regression test scripts to validate gestures.

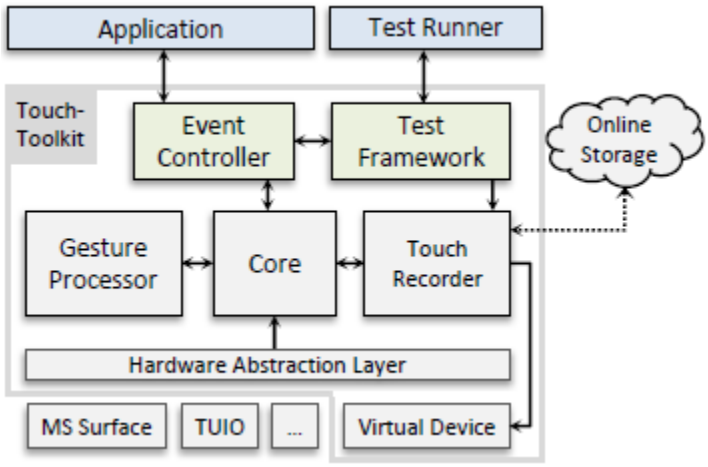


Figure 22: Components of GestureToolkit Framework

The component diagram of the GestureToolkit framework is shown in Figure 22. The six key components of the toolkit are:

(1) the hardware abstraction layer which exposes a hardware agnostic API for the application,

(2) the gesture processor that recognizes gestures from the raw touch data,

(3) the core that acts as a bridge among the components,

(4) the touch recorder that stores the raw data from the hardware abstraction layer,

(5) the test framework that executes the automated test scripts, and

(6) the event controller that keeps track of all gesture event requests from applications.

We describe these key components in the subsequent sections.

5.1 Hardware Abstraction Layer

We followed a similar approach as Echtler [8] to decouple the actual hardware from the application layer. This allows the gesture definition to be device independent. This module provides a hardware agnostic interface for capturing multi-touch inputs. This interface can be implemented for wide range of hardware platforms. GestureToolkit currently has implementations for Microsoft Surface, SMART Tabletop, Windows 7 (WPF 4.0 and Silverlight 4.0), Anoto Pen and TUIO protocol. The framework also has an implementation for a virtual device that can be used to simulate multi-touch inputs. The virtual device can playback the recorded interactions and run automated tests.

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, the framework 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. Figure 23 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.

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

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

Figure 23: 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.

The hardware abstraction layer receives low level touch data from touch input providers. A provider is responsible for translating device inputs into a generic data format supported by the framework. Each supported device has its own implementation of an input provider which is developed by extending the base input provider class. The base input provider provides easy access to reusable utility functions and common features like data caching to simplify the development and maintenance. The additional code in the device specific provider is mainly responsible for converting device specific data into a generic data format that rest of the framework can process.

Depending on the hardware interface and drivers, devices can send touch input data in various formats. For example, Microsoft Surface SDK provides an event model that others can subscribe to receive touch inputs in an asynchronous fashion whereas other devices (e.g. AnotoPen) uses TUIO protocol that sends data over network layer. Device specific providers are responsible for converting these data into three events: `SingleTouchChange`, `MultiTouchChange` and `FrameChange` as per the design requirement of the GestureToolkit framework. Appendix D describes how to develop a provider with a step by step code example.

5.2 Gesture Processor

The gesture processor has two key responsibilities. First, it is responsible for parsing the gesture definitions, compile and embed the output files into application assemblies. Then at runtime, it also processes the raw touch input data received from the core component and validates the inputs using the logic defined in the compiled gesture definitions. On successful detection of a gesture, it computes the results as expressed in the return block of the gesture definition and finally, sends the results to the core component. The core component is responsible for notifying the applications via the event controller.

As explained in Chapter Four, the rules and syntax of the gesture definition are defined using the MGrammar [41] language definition tool. The gesture processor uses the parser provided by the MGrammar to parse the code in a gesture definition file. Once the entire abstract syntax tree (AST) is generated for the code, the gesture processor uses MGraphXamlReader library to instantiate objects of respective primitive conditions with the values defined in the definition. Finally, the entire collection of primitive conditions are serialized into java script object notation (JSON) format and stored in a text file so

that at runtime the application can directly get the collection of primitive conditions from the serialized form.

At runtime, when the application code subscribes for a particular gesture for the first time, the gesture processor instantiates the set of primitive conditions from the serialized form. However, if the same gesture is subscribed multiple times, the framework reuses the same instances. As a result, when input is received, the gesture processor only evaluates the primitive conditions that are needed for active gestures.

5.3 Core Component

The core component acts as a bridge among the components. It is also responsible for maintaining history to touch data and execute other general components including visual effects. The touch history is managed by the *Data Queue* module and visual effects are managed by the *Visual Feedback* module, both part of the core component.

5.3.1 Data Queue

The data queue maintains current state and recent history of raw touch data. It provides touch-related information to rule validators in various forms, including recent touch history, touch path, time stamps, and age of a touch. During system initialization, the gesture processor notifies the data queue about the possible longest history of data that could be requested by any active gesture. This information may change during runtime as developers can dynamically add new gestures at any point in the application runtime. If a new gesture event is registered, a message from the event controller via the gesture processor will notify the data queue. The data queue deletes all touches from the history that are outside of the time frame of interest.

5.3.2 Visual Feedback

Like any communication, feedback is important for multi-touch systems. While a system can provide feedback to users via audio, visual or tactile medium, the visual feedback is the most widely used in todays' applications. GestureToolkit provides two types of visual feedback:

1) touch feedback that provides visual feedback for any arbitrary touches, and
2) gesture feedback which provides a visual feedback when the desired gesture is detected.

5.3.2.1 Touch Feedback

While the framework provides a default feedback for touch interactions, it also allows the developer to create their own custom feedback. The framework provides a plug-in architecture that allows developers to use visual feedback from external sources as well as share their own custom feedbacks with others.

Figure 24: Example of a Touch Feedback

Figure 24 shows a touch feedback named *BubblesPath* that comes out of the box with GestureToolkit. The plug-in architecture simplifies the development of touch feedbacks

to a great extent. For example, a specific touch feedback plugin only defines a single instance of a touch point (e.g. one gray bubble in Figure 24) and how it will animate. The framework internally handles implementation complexities like multiple instances for long touch path as well as multiple touch points, background threading, frame rates for performance and so on. Appendix E shows the implementation of *BubblesPath* touch feedback.

The framework also simplifies the process of managing touch feedback within the application by exposing public interfaces. Figure 25 shows a code snippet to add the bubbles path touch effect.

```
//Add touch feedbacks
GestureFramework.AddTouchFeedback(typeof(BubblesPath));
```

Figure 25: Code snippet to add touch feedbacks

5.3.2.2 Gesture Feedback

Like touch feedback, application developers can also create their own gesture feedbacks or use visual effects that come with the toolkit out of the box. While a touch feedback is visible whenever a touch data is received, the gesture feedback is used to notify the user when the specified gesture is recognized.

Figure 26: The gesture feedback for the lasso gesture

Figure 26 shows the `HighlightSelectedArea` gesture feedback that is used to inform the user about the area selected the lasso gesture. The source code for this gesture feedback with step by step explanation can be found in Appendix F.

5.4 Touch Recorder

To record interactions, the Touch Recorder subscribes to lower level input from the hardware abstraction layer through the core component and saves the data into an online storage and also caches it locally to improve performance. This allows automatic synchronization of data between developer machines and actual devices.

During playback this module reconstructs the touch data object from the XML content and sends the data to the system through a virtual device so that it appears to the rest of the system as if it is coming from the actual device. This allows the developers to test applications that require multi-touch interactions on their development machine. Figure 27 shows example of using the recorder module to individually record the lasso and zoom gesture and later play them in parallel to test multi-user scenarios.

This touch recorder can also be used in applications to implement features like interactive tutorials, touch data collection and the like. The touch recorder provides following API methods which application developers can use:

Method Name	**Description**
StartRecording()	Starts recording all touch interaction data
StopRecording()	Stops recording and returns the recorded data in xml form
StartPlayback(…)	Starts playback of the recorded touch interactions. This method has a number of overload methods including options

	to play single set of data as well as multiple datasets by merging them into one timeline.

5.4.1 Storage Module

The data is stored in an XML format (Figure 28). The recorder can record and store interactions from any device that is supported by the hardware abstraction layer, including basic touch information (i.e., coordinates, touch ID) and any additional device specific data provided by the hardware.

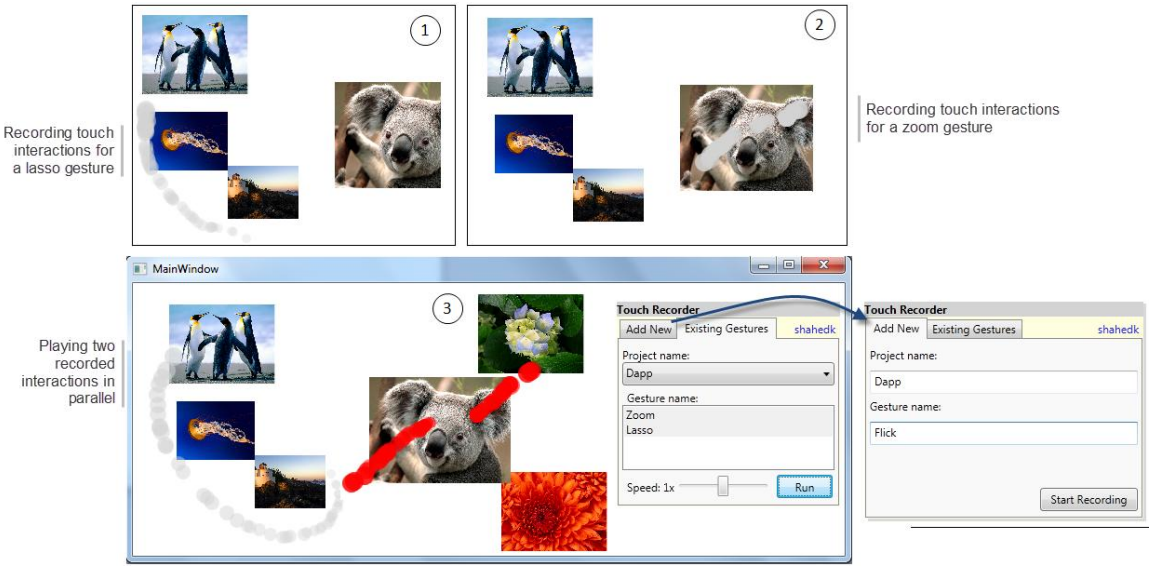


Figure 27: GestureToolkit: Simulating multi-user touch interactions

```
<FrameInfo>
  <TimeStamp>10926403</TimeStamp>
  <Touches>
    <TouchInfo>
      <ActionType>1</ActionType>          Basic
      <Position>                          touch data
        <X>451.14</X>
        <Y>107.29</Y>
      </Position>
      <TouchDeviceId>10</TouchDeviceId>
      <Tags>                              Device
        <Tag>                             specific
          <Key>Size</Key>                 touch data
          <Value>10</Value>
        </Tag>
      </Tags>
    </TouchInfo>
  ...
  </Touches>
</FrameInfo>
```

Figure 28: An XML code fragment representing a part of a touch interaction

Figure 28 shows an XML code fragment generated by the touch interaction recorder. The recorder records both basic touch data that are common to all supported devices and also the device specific data (e.g. touch size, touch direction) under the `Tags` node.

5.4.2 Local Cache

GestureToolkit supports both web based applications and regular desktop applications. For web applications, it uses the Silverlight framework. A difference between the desktop and the web platform is that web applications do not have access to local file systems which is needed to maintain the cache data. To overcome this, the framework uses isolated storage when running on web browsers and the file system for regular desktop applications. However, this is internally handled by the storage manager and application developers only need to code against the `StorageManager` class. Figure 29 shows the internal structure of the `StorageManager` class.

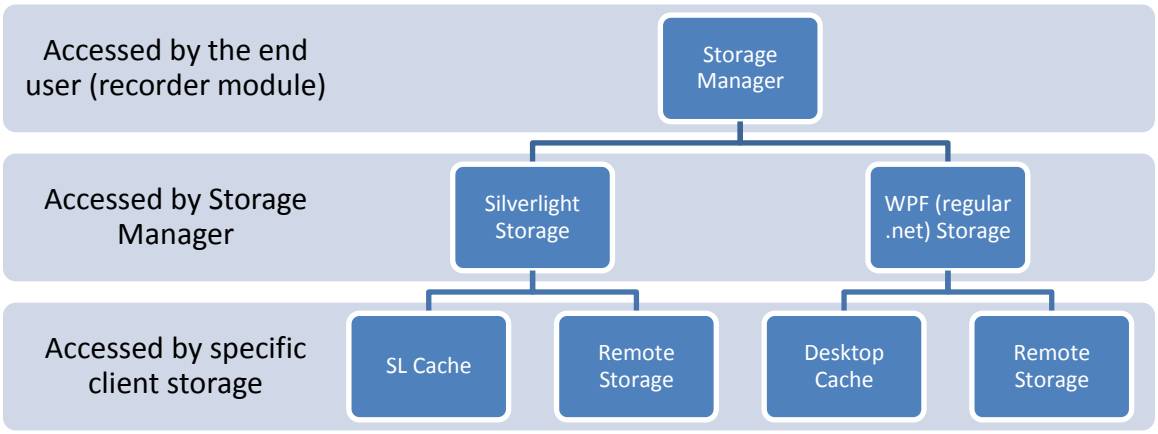


Figure 29: Internal Structure of Storage Manager

5.4.3 Remote Storage

The remote storage uses a relational database at the backend to store the data and exposes an XML web service which is publicly accessible and authenticated by user credentials. The web service is developed using ASP.NET and it uses a Microsoft SQL Server database.

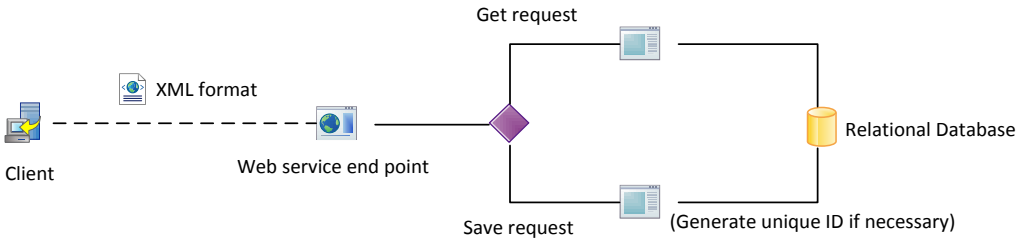


Figure 30: Internal Design of Remote Storage

Figure 30 describes the internal design of the remote storage system. The web service in remote server transfers data between the server and the client in text form which contains XML data. When client requests to save a new touch interaction data, the server ensures that the key provided by the client is unique. Since there could be multiple clients

communicating with the server at the same time, if a duplicate key is found the server adds a timestamp at the end of the key to make it unique.

5.5 Automated Test Framework

Automated unit testing is a well-known way to increase the effectiveness, efficiency and coverage of software testing [40]. It is one of the industry standard methods for repeatedly verifying and validating individual units of the application in regression testing. Though there are some simulators available to manually test tabletop applications, tool support for unit testing multi-touch gestures is limited.

Record and playback can be used for both manual and automated testing. While manual test may involve gesture detection as well as other UI related functionality testing, the automated test framework focuses specifically on validating gesture detection code. Most automated Unit Test systems do not have the option to use an active UI during test. However, gestures are directly related to the UI and testing them often requires UI specific functionality. To mimic a realistic application scenario, the test framework creates an in-memory virtual UI layer and subscribes to gesture events in the same way that an application would. The test framework can be used to test any type of gestures that is defined using the gesture definition language, including complex multi-touch gestures that involve touch interactions with multiple steps.

Figure 31 shows the workflow of an automated test in GestureToolkit. To start, it creates the virtual application and registers the necessary gesture events during the initialization process. Then the *TouchRecorder* loads the data from storage and starts the simulation process. If a test involves simulating multi-user scenarios using multiple touch interactions then the system merges frames from individual recorded data into one time

line. The virtual device continues to send simulated device messages to the framework. As soon as the desired gesture is detected, it invokes the user defined validation code. Depending on the type of gesture, the user defined code can be invoked multiple times. Regardless of the status of gesture detection, the framework also invokes the "Playback Completed" test code defined by the user at the end.

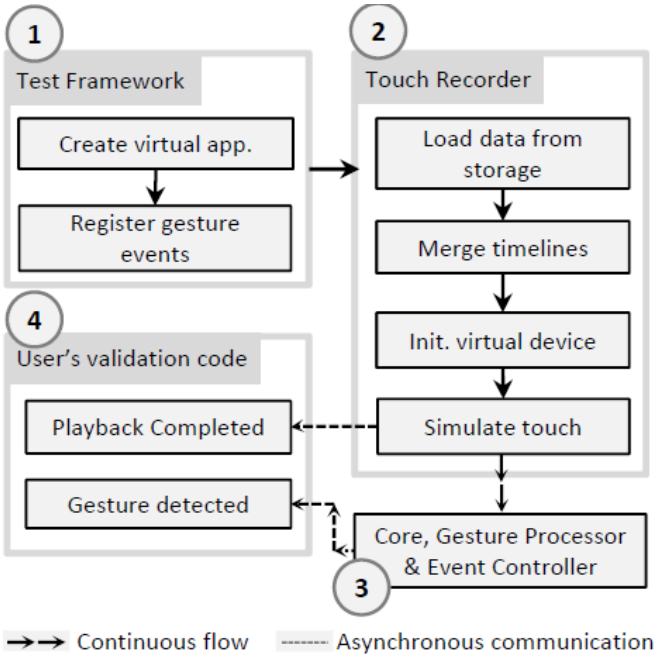


Figure 31: Workflow of Automated Test Framework

5.5.1 Asynchronous Test Environment

Traditional unit tests execute sequentially, however, multi-touch gesture based user interactions require asynchronous processing. For example, a "Flick" gesture requires a certain period of time to complete, so a test to validate the recognition of that gestures must wait until the gesture is completed. Such tests require asynchronous execution which is supported by the GestureToolkit test framework API.

Multi-touch gesture interactions can trigger continuous events. For an example, a photo viewer application needs to keep responding to the “Zoom” gesture in “real-time” as long as the zoom interaction continues. So, a test for this scenario needs to validate the zoom interaction continuously instead of just once it is completed. The GestureToolkit test framework allows a developer to write unit test cases for such continuous touch-interactions.

To support asynchronous and continuous interaction testing, our test framework is event driven. The core of the API is the *Validate* method which takes the following parameters:

- *expectedGestureName*: The name of the gesture to detect (e.g., Zoom).
- *savedInteraction*: The identifier or the recorded interaction that should produce the expected touch interactions.
- *gestureDetectedCallback*: The method to call when the gesture is detected. Developers can write custom validation code inside this method.

```
1   [TestMethod]
2   public void TestZoom()
3   {
4     bool gestureDetected = false;
5     var threadHolder = new AutoResetEvent(false);
6
7     GestureTestFramework.Validate("Zoom", "TouchInteraction02",
8       // On successful gesture detection
9       (sender, e) =>
10      {
11        gestureDetected = true;
12        if (e.Error == null)
13        {
14          var distanceChanged = e.Values.Get<DistanceChanged>();
15          // User defined validation code
16        }
17        else
18        {
19          Assert.Fail(e.Error.Message);
20        }
21      });
22
23    threadHolder.WaitOne();
24    Assert.IsTrue(gestureDetected, "Failed to detect the gesture!");
25  }
```

Figure 32: Example unit test code using Touch Toolkit

Figure 32 shows a code fragment that is used to test if the "Zoom" gesture is detected as a result of playing the saved interaction named "TouchInteraction02". In line 5 and 23 of the code, we use an existing class named AutoResetEvent from the System.Threading class library in the Microsoft .NET framework to implement the asynchronous unit test execution. AutoResetEvent allows threads to communicate with each other by sending signals. The Validate API as discussed above is used in line 7. In addition to just the detection of the "Zoom" gesture a developer can provide additional validation code around line 15.

```
// Custom validation code
if (previous != null)
{
  Assert.IsTrue(current.Distance>previous.Distance,
                "Invalid Zoom");
}
previous = current;
```

Figure 33: User defined gesture validation code

Figure 33 shows how to write unit test for continuous interactions such as “Zoom”. Here the purpose of the validation is to see if the distance between subsequent touch points during a “Zoom” gesture is increasing. Using this same approach GestureToolkit allows developers to write unit test code for validating multi-touch interactions.

5.6 Event Manager

The framework provides an interface to subscribe to events for specific gestures in a method similar to how applications receive messages for mouse or keyboard events. shows the code snippet to subscribe a gesture named “zoom”. It also allows defining the scope of the gesture which is the image object “image1” 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 34: 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 35 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.

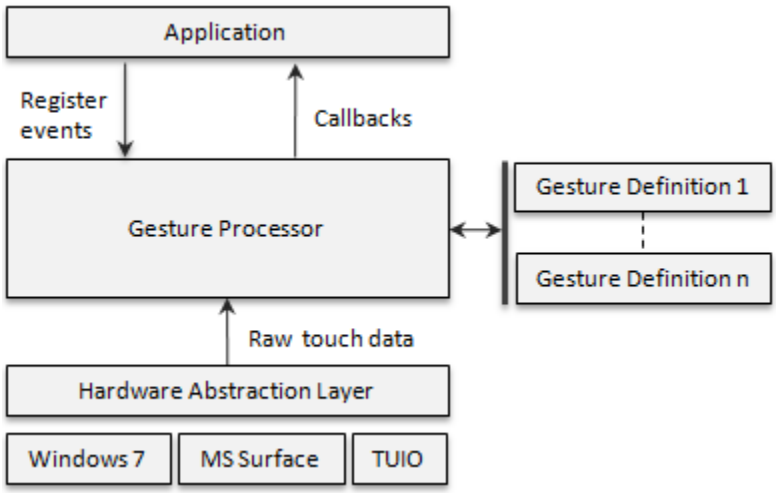


Figure 35: The simplified architecture of gesture processor

The 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.

5.7 Summary

This chapter focuses on describing the implementation details of GestureToolkit. It also explains how the touch interaction recorder can be used to simulate multi-user scenarios and write automated test scripts. The technical challenges that were faced during the development of GestureToolkit are discussed in the next chapter.

Chapter Six: **Technical Challenges**

From planning to designing and throughout the implementation stage, we faced different types of technical challenges including choosing the right platform, future maintenance, usability of the toolkit and many more. We describe these challenges in the following sections.

6.1 Designing the Language

While choosing the platform for the language, we found several possible approaches: first, use an existing general purpose programming language (i.e. C++, C#, Java) to define gestures; second, use functional programming languages (i.e. Scala, F#) to define gestures; third, create a new domain-specific language from scratch. All of these approaches have trade-offs in terms of ease of use, runtime performance, expressiveness, flexibility and ease of implementation.

The key advantage of using an existing general purpose programming language is that it has very limited or almost no learning curve. An existing language also reduces the implementation complexity, considering that the language grammar, compilers, and developer tools are already available. The main reason behind not choosing this approach is the inability to add new primitives to the language to ensure simplicity and readability. For example, the zoom gesture definition may use keywords like “increasing” to define touch patterns and reader friendly names for return types. It would not be possible to provide such custom syntax support with C++ or any other general purpose language.

The features of new functional programming languages like Scala [23] or F# are impressive. The syntax of F# carries less noise (i.e. no curly brackets, simpler parameter passing) compared to C++ or similar languages. It also provides unique features, like

method chaining, that are important to GDL. Finally, under the hood it's just another language in the .NET family and it compiles into a regular common language runtime class, which makes the integration with existing UI frameworks quite straightforward. Although the features of F# look appealing, it suffers from limitations similar to general purpose languages including the inability to add new coding styles, keywords, and the like.

Considering the alternatives and requirements of the language, we chose to design a new language from scratch. Mernik [27] provides some good explanations on when and how to develop a new DSL along with common challenges. There are a few tools available to design DSLs, including ANTLR [25] and MGrammar from Microsoft DSL tools (codenamed "Oslo") [26]. We decided to use MGrammar for a number of reasons; it provides built-in support for IntelliPad, a free IDE for language design, and it supports on-the-fly error checking and syntax highlighting for DSLs without much additional effort.

We developed the language parser and framework using Microsoft .NET. It can be used in both web applications via Silverlight and desktop applications like Windows Forms or Windows Presentation Foundation. Silverlight provides comprehensive support for user interface design and native multi-touch, and runs on both Mac and Windows operating systems. So, we believe it can reach a greater range of users and developers compared to other platforms.

6.2 Hardware independence with device specific support

Windows 7 is the first operating system from Microsoft that has native support for touch devices. So as long as a multi-touch device is supported by the Windows 7, the

application developers can rely on touch messages from the operating system and build device independent applications. However, the reality is that a number of popular multi-touch devices including the Microsoft Surface do not have the necessary driver support for the Windows 7 operating system. As developing applications using the vendor provided SDK (e.g. Microsoft Surface SDK, SMART SDK) would make the application bind with that particular device, creating a device independent multi-touch application framework was a challenge.

To build a framework that would work with different multi-touch devices, the design of the communication modules that passes the touch related messages into the framework needs to be decoupled and extensible. So that the rest of framework and applications developed using the framework can work without any change in code. To achieve this, we followed the provider model design pattern. In GestureToolkit, the modules that responsible for translating device specific data into a generic format are called input providers. Each provider is extended from a base provider and the framework uses one of these providers depending on the device the application is running on.

6.3 Extensibility

During the development and evaluation process we observed that the gesture definition language needs an easy-to-implement extensibility framework that would allow a knowledgeable application developer to create or share new primitive conditions and return types. We found two key challenges to provide this functionality. First, the language parser needs to be updated whenever a primitive condition or return type is added. Second, the framework should be able to get the related class files from the

application assembly so that they don't have to add the classes in framework assemblies which would require recompilation of the framework.

To address the first challenge, the parser is designed in a way that application developers can only provide the parsing logic for the new language components inside the application assembly and at runtime the parser will update the parser definition, recompile and build the parser module and then start parsing the gesture definitions. The details are explained in section 4.1.3.

To address the second challenge, the framework uses reflection[1] to find out and dynamically load the dependent classes related to primitive conditions and return types that are used in the gesture definition. This process allows the external developers to independently implement new recognition algorithms and contribute to GestureToolkit framework. These language components follow a plug-in approach which allows application developers to freely share them across projects.

6.4 Developer Expectations

From the exploratory study, developer feedback in preliminary evaluation and based on our own experience, we found that developers have a wide range of expectations from a toolkit. These expectations can be grouped into two categories: Productivity Tools and Implementation Details.

6.4.1 Productivity Tools

A common developer expectation is good IDE support including warning message, templates, syntax highlighting, auto-complete and integrated build. To address these

[1] A technique by which a program can observe and modify its own structure and behavior at runtime.

issues, GestureToolkit provides these features except auto-complete for Visual Studio 2010 – the most widely used IDE in .NET developer community. The features related to integrated developer environment are described in Section 4.2.4.

6.4.2 Implementation details

We observed that experienced developers are often interested in low level implementation details, performance issues and access to the framework via an application programming interface (API). To address this, we ensured that all public methods in the framework have comments about its functionality. They were written by the same developer who developed it. The framework also provides a rich set of low-level APIs that allows knowledgeable developers to directly use internal framework features. For example, the same recorder module that is used by the automated test framework to store application specific touch interactions can be used to build interactive tutorials for custom applications.

Chapter Seven: **Evaluation**

After completing the implementation of all major features, a preliminary evaluation was conducted with the same participants of the initial exploratory study to assess the usability and appropriateness of the toolkit.

We performed the preliminary evaluation with a focus on getting early feedback from experienced multi-touch application developers about the usability and appropriateness of the toolkit. All of the studies were done individually. We used an audio recorder to record the conversations and a screen capturing tool to record a user's activities on the screen during the coding tasks. The evaluation was done three months after the initial exploratory study using the same participants.

7.1 Data Collection

Each session lasted 50 minutes and consisted of three sections as follows:

First, we showed the participant a seven minute introductory video describing the main features of our framework and spent 1-2 minutes on follow up discussion. Then, we asked the participant to perform the following three tasks to a partial implementation of a tabletop photo viewer application:

- Add a photo resize features using the predefined zoom and pinch gestures.
- Record the resize interaction for later be use in automated testing of the application.
- Write a UnitTest to automatically test the zoom gesture in an existing visual studio test project.
- Define a new gesture using GDL

Finally, we conducted a semi-structured interview to collect feedback on the toolkit and to better understand each participant's experiences using the toolkit.

7.2 Findings

At a high level, our preliminary evaluation has suggested that the GestureToolkit test framework:

- can be used to write unit tests for touch interactions, and
- the record/replay feature can be used to overcome some of the testing and debugging challenges.
- GDL helps to simplify the gesture definition process to a great extent. However, better IDE support can improve developer efficiency.

These findings are discussed in some more detail in the remainder of this section.

7.2.1 Findings from Task 1: Adding Resize Functionality

Participants were asked to add resize functionality for the image objects. The required gestures (zoom and pinch) were available in GestureToolkit's predefined list of gestures. The purpose of this task was to evaluate the usability of the gesture event subscription API.

Participants P2 and P3 were able to complete the task. However, P1 partially completed the task but faced difficulty on using the right data type from the library. We found that she was expecting that the IDE's auto-complete feature would help her to determine which types to use, however this feature was not supported in this case.

In summary, the participants were able to understand how to use the toolkit to implement an application feature. We also found that participants were expecting comprehensive IDE support not yet available in our prototypical implementation.

7.2.2 Findings from Task 2: Record the Resize Touch Interaction

This task requires one to do the following:

- Write the appropriate code to show the recording panel in the debugger.
- Use the recorder panel to record the touch interactions.

These steps were demonstrated in the introductory video. All of the participants successfully completed this task. This feedback indicates that our participants could quickly learn how to record an interaction using the tool.

7.2.3 Findings from Task 3: Writing Unit Test

Participants were asked to write a unit test to validate the "zoom" gesture. To ease the process of writing unit tests, GestureToolkit provides IDE templates of test code. However, a developer needs to write the actual test code depending on the specific interaction under test.

The purpose of this task was to see how easily a developer can understand the test API and the appropriate structure for writing unit tests for gestures. The "zoom" gesture was chosen for three reasons. First, it is a common and comparatively simple gesture and the associated test plan is straightforward. Second, this is a scenario that requires developers to write asynchronous test code and developing the test logic. Finally, "zoom" is a continuous gesture that requires the test code to react in a continuous fashion. This test scenario gave our participants an opportunity to use all of the main testing features provided by our framework.

All the participants completed the task. While participant P1 was not familiar with the concept of inline functions which simplifis the asynchronous code execution to a great extent, she was able to complete the task as the IDE template already placed the basic

code structure. This indicated that the participants, after implementing and recording an interaction using GestureToolkit, could write an automated test for the interaction. We also found that templates can help reduce the learning curve for new developers.

7.2.4 Findings from Task 4: Define a New Gesture using GDL

The participants were given a printed copy of the list of available primitive conditions and return types and asked to define a five-finger-selection gesture. The participants were given with examples of the gesture but specific rules to recognize the gesture was left out for participants to decide. The definitions written by each participant were different from the logical perspective. However, they were able to use the language to define the gesture according to their own logical concepts. This shows that developers can use the language to define custom gestures.

7.3 Summary of Preliminary Evaluation

This study has provided preliminary evidence that our test framework provides effective support for many of the challenges that our participants faced in their debugging and testing of tabletop applications. For example, Participant P3 appreciated that the GestureToolkit allowed him to test and debug without moving between the tabletop device and his workstation, which he believes will help him to be more efficient in his development:

"For sure it will save a lot of development time as you don't have to move between the device and development machine back and forth just to test a feature."

He also felt it was valuable that *"you can also interact when playback is going on"* as it makes many debugging scenarios easier. Participant P2 felt that the main benefit of the framework for him would be the ability to develop automated tests and use those as part

of continuous integration suite. Finally, participant P1 felt that the support for device independent record and replay in the GestureToolkit to be the most useful for her development work.

7.4 Community Response

In addition to formal evaluation we also received community initiated responses. The GestureToolkit project is an open-source project published under the GNU Library General Public License (LGPL) and hosted at CodePlex. The entire source code, documentation, issue tracker and discussion on future plans are available at http://gesturetoolkit.codeplex.com. Since the first release, we have received encouraging feedback from both the academic research community and industry including team members from the tabletop team at SMART Technologies and the Microsoft Surface team at Microsoft. The project has also been highlighted on a number of popular websites including www.infoQ.com. Figure 36 shows the project website activity from April 2010 to October 2010.

Page Views	13431
Visits	2843
Downloads	360
Application Runs	N/A

Figure 36: The GestureToolkit Project website activity till October 2010

Chapter Eight: **Conclusion**

This thesis presents an approach to improve the tool support of multi-touch application development and testing. First, an overview of the challenges involved in developing multi-touch applications was presented to provide the background necessary to understand the challenges in this field. Next, a discussion of previous attempts in different areas of multi-touch application development process was presented, and the strengths and weaknesses of these approaches were discussed. A domain-specific language to define gestures and a framework, GestureToolkit, were developed to reduce the development complexities and provide support for testing multi-touch interactions in both manual and automated approach. The structure of GestureToolkit and its implementation details were discussed. Preliminary evaluations were then conducted to validate the research goals described in Section 1.2 and to give insight into the strengths and weaknesses of GestureToolkit.

8.1 Thesis Contributions

The first contribution of this thesis is the exploratory study covering the challenges that the developers are facing today to build multi-touch applications. First, it compares the results of existing research on useful gestures for multi-touch surfaces with the gesture support available in existing tools and frameworks. Then, it presents an investigative report on three touch based applications to understand the touch related requirements of different applications. Finally, a semi-structured interview with experienced developers reviled the challenges a developer face during building multi-touch applications. The result of this exploratory study should make it easier to focus on framework and tool

development in the future as it can help to determine the difficulties of the development process of multi-touch applications.

The second contribution of this thesis is the gesture definition language (GDL). GDL presents a new concept for defining gestures and an approach to integrate it with existing application frameworks. The language supports multi-user, multi-touch and multi-step gestures; and also provides an extensible architecture that allows adding new primitives into the language. Preliminary evaluation shows that developers like the concept of a domain-specific language to define gestures.

The third contribution of this thesis is GestureToolkit – a software development kit for multi-touch applications. GestureToolkit fulfills the research goals described in Section 1.2. Not only GestureToolkit is the only tool currently supports a domain-specific language to define custom gestures but it is also the only tool that provides an automated test framework for gesture validation. GestureToolkit also provides a visual feedback framework that allows the developer to build applications with consistent visual feedback for touch and gestures across different devices. The device virtualization technique of GestureToolkit enables development and testing of multi-touch applications from non-touch enabled computers (e.g. regular desktop PC). Most important, however, is that GestureToolkit is a device independent framework and applications developed using it can run on any of the supported devices without any change in the application code.

8.2 Limitation

While GestureToolkit helps to simplify the multi-touch application development in many ways, it also has a few limitations. 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. While the language supports multi-step gestures, it is currently limited to gestures with sequential steps. The language can also be extended to support wider range of logical conditions (e.g. NOT, XOR).

The purpose of the preliminary user study was to generally evaluate the approach taken in GestureToolkit. We had three experienced tabletop application developers as our participants. We recognize that a comprehensive user study involving more participants can provide more generalizable insights about the approach and our tool.

8.3 Future Work

The ongoing research focuses on generating gesture definitions from sample datasets of touch interactions and a visual representation of the gesture definition. This will allow users to define a gesture from sample touch data and the visual DSL will allow non-experts to fine tune the logical conditions. However, the research is still at the early state and we would like to provide these features in GestureToolkit in the future.

We would also like to provide additional logical operators in the gesture definition language. In addition to the record/replay based testing, a programmable user interface automation testing for multi-touch applications could also be an interesting approach that we would like to investigate in the future.

References

[1] Selim, E. and Maurer, F. eGrid: Supporting the Control Room Operation of a Utility Company With Multi-Touch Tables. The ACM International Conference on Interactive Tabletops and Surfaces, November 7-10, 2010, Saarbrucken, Germany.

[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] Zabir, O., Khandkar, S. Hossain, M., and Raihan, A. 2005. SmartUML. http://smartuml.sourceforge.net

[4] Multi-Touch Vista, http://multitouchvista.codeplex.com/. 2010

[5] N. Villar et al., "Mouse 2.0: multi-touch meets the mouse," in Proc. UIST, vol. 9, pp. 33–42.

[6] U.S. Pawar, J. Pal, and K. Toyama, "Multiple mice for computers in education in developing countries," International Conference on Information Technologies and Development, 2006.

[7] J.C. Lee, "Hacking the nintendo wii remote," IEEE Pervasive Computing, 2008, pp. 39–45.

[8] Florian Echtler and Gudrun Klinker, "A Multitouch Software Architecture," NordiCHI 2008: Using Bridges, 18-22 October, Lund, Sweden.

[9] Touchlib: A multi-touch Development Kit, http://nuigroup.com/touchlib/, 2010.

[10] E. T. Hansen, J. P. Hourcade, M. Virbel, S. Patali, T. Serra. “PyMT: A Post-WIMP Multi-Touch User Interface Toolkit”. International Conference on Interactive Tabletops and Surfaces, 2009.

[11] Bastéa-Forte M., Yeh, RB and Klemmer, S.R. Pointer: Multiple Collocated Display Inputs Suggests New Models for Program Design and Debugging. In Extended Abstracts of UIST (Posters), 2007.

[12] J. O Wobbrock, A. D Wilson, and Y. Li, “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, 2007, 168.

[13] S. Kratz and M. Rohs, “A $3 gesture recognizer: simple gesture recognition for devices equipped with 3D acceleration sensors,” Proceeding of the 14th international conference on Intelligent user interfaces, 2010, pp. 341–344.

[14] Sezgin, T.M. and Davis, R. (2005) HMM-based efficient sketch recognition. Proc. IUI '05. New York: ACM Press, 281-283.

[15] Cao, X. and Balakrishnan, R. (2005) Evaluation of an on-line adaptive gesture interface with command prediction. Proc. Graphics Interface '05. Waterloo, Ontario: CHCCS, 187-194.

[16] Anderson, D., Bailey, C. and Skubic, M. (2004) Hidden Markov Model symbol recognition for sketch-based interfaces. AAAI Fall Symposium. Menlo Park, CA: AAAI Press, 15-21.

[17] Pittman, J.A. (1991) Recognizing handwritten text. Proc. CHI '91. New York: ACM Press, 271-275.

[18] Cho, M.G. (2006) A new gesture recognition algorithm and segmentation method of Korean scripts for gesture-allowed ink editor. Information Sciences 176 (9), 1290-1303.

[19] Dietz, P. and Leigh, D. DiamondTouch: A Multi-User Touch Technology. UIST 2001, 219-226

[20] D. Schmidt, H. Gellersen, “Show Your Hands: A Vision-Based Approach to User Identification for Interactive Surfaces”, International Conference on Interactive Tabletops and Surfaces, 2009

[21] I. Rosenberg, K. Perlin, C. Hendee, A. Grau, and N. Awad, “The UnMousePad: the future of touch sensing,” SIGGRAPH'09: Posters, 2009, p. 23.

[22] C. North, T. Dwyer, B. Lee, D. Fisher, P. Isenberg, G. Robertson, K. Inkpen, and K.I. Quinn, “Understanding Multi-touch Manipulation for Surface Computing,” Interact 2009

[23] The Scala Programming Language. http://www. scala-lang.org/, 2010

[24] F# at Microsoft Research. http://research. microsoft.com/en-us/um/cambridge/projects/fsharp/, 2010

[25] ANTLR: A Tool to Build Domain Specific Languages, http://www.antlr.org/, 2010

[26] Microsoft DSL tools, http://msdn.microsoft.com /en-us/data/default.aspx, 2010.

[27] M. MERNIK, J. HEERING, and A.M. SLOANE, “When and How to Develop Domain-Specific Languages,” ACM Computing Surveys (CSUR) Volume 37, 316 - 344, Issue 4, 2005, 2004, p. 694.

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

[29] J.O. Wobbrock, M.R. Morris, and A.D. Wilson, “User-defined gestures for surface computing,” Proceedings of the 27th international conference on Human factors in computing systems, 2009, pp. 1083–1092.

[30] S. Gilbert et. al. SprashUI toolkit. http://code. google.com/p/sparsh-ui/

[31] Fails, J. and Olsen, D. A design tool for camera-based interaction. In Proc. CHI, 2003.

[32] 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.

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

[34] MGraphXamlReader, http://code.msdn.microsoft .com/SQLModCTPMGXaml, 2010

[35] Strauss, A. L. and Corbin, J. Basics of Qualitative Research: Techniques and Procedures for developing Grounded Theory. Sage Publications, 1998.

[36] Dan Saffer: Designing Gestural Interfaces, 2008.

[37] PLATO (Programmed Logic for Automated Teaching Operations) was the first (ca. 1960, on ILLIAC I) generalized computer assisted instruction system. http://en.wikipedia.org/wiki/Plato_computer, 2010.

[38] Multi-Touch Systems that I Have Known and Loved, Bill Buxton, http://www.billbuxton.com/multitouchOverview.html, 2010

[39] Microsoft Surface, http://www.microsoft.com/surface/en/us/default.aspx, 2010.

[40] Beck, K. and Andres, C. 2004 Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.

[41] David Langworthy; Brad Lovering; Don Box. The “Oslo” Modeling Language: Draft Specification - October 2008

[42] Fails, J. and Olsen, D. A design tool for camera-based interaction. In Proc. CHI, 2003.

[43] Witten, I. H. and Frank, E. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[44] Westeyn, T., Brashear, H., Atrash, A. and Starner, T. Georgia tech gesture toolkit: supporting experiments in gesture recognition. In proc. ICMI, 2003.

[45] Ashbrook , D. and Starner,T. “MAGIC: a motion gesture design tool,” in Proceedings of the 28th international conference on Human factors in computing systems, pp. 2159-2168, 2010.

[46] Long, A. C., Landay, J. A. and Rowe, L. A. “Quill: a gesture design tool for pen-based user interfaces,” Eecs department, computer science division, UC Berkeley, Berkeley, CA, 2001.

[47] Kara, L. B. and Stahovich, T. F. An image-based, trainable symbol recognizer for hand-drawn sketches, Computers & Graphics, vol. 29, no. 4, pp. 501-517, 2005.

[48] Cardenas, T., Bastea-Forte, M., Ricciardi, A., Hartmann, B. and Klemmer, S. R. Testing Physical Computing Prototypes Through Time-Shifted \& Simulated Input Traces. In extended abstracts of UIST 2008.

[49] Testing the User Interface with Automated UI Tests, Microsoft Developer Network, 2010. http://msdn.microsoft.com/en-us/library/dd286726.aspx

[50] Project White, http://white.codeplex.com/white, 2010.

[51] Selenium, http://seleniumhq.org/, 2010.

[52] QF Test, http://www.qfs.de, 2010.

[53] Frog Logic, http://www.froglogic.com/, 2010

[54] Cocoa Touch Apps, http://cocoatouchapps.com, 2010.

[55] Vimov, http://www.vimov.com/, 2010.

[56] Gray, P., Ramsay, A. and Serrano, M. "A demonstration of the OpenInterface Interaction Development Environment," UIST'07 Adj. Proc.

[57] SMART Tabletop, http://smarttech.com, 2010.

[58] Windows 7 Touch, http://windows.microsoft.com/en-CA/windows7/products/features/touch, 2010.

[59] TUIO, http://tuio.org, 2010.

[60] IntelliPad, http://blogs.msdn.com/b/intellipad/, 2010.

Appendix A: List of Predefined Gestures

1. Tap

```
name: Tap

validate
    Touch state: TouchUp
    Touch limit: 1
    Touch time: 10 msec
    Touch path bounding box: 1x1..100x100

return
    Position
```

2. Drag

```
name: Drag

validate
    Touch state: TouchMove
    Touch limit: 1..3
    On same object

return
    Position, Position changed
```

3. DoubleTap

```
name: DoubleTap

validate
    Touch state: TouchUp
    Touch limit: 1
    Touch step: 2 touches within 1 sec
    Touch area: Rect 50x50 including last 1 touch within 1 sec

return
    Position
```

4. Zoom

```
name: Zoom

validate
    Touch state: TouchMove
    Touch limit: 2
    On same object and Distance between points: increasing

return
    Distance changed
```

5. Pinch

```
name: Pinch

validate
    Touch state: TouchMove
    Touch limit: 2
    On same object and Distance between points: decreasing

return
    Distance changed
```

6. Rotate

```
name: Rotate

validate
    Touch state: TouchMove
    Touch limit: 2
    On same object
    and Distance between points: unchanged 10%

return
    Info:Rotate, Slope changed
```

7. Lasso

```
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
```

8. 5-finger-selection

```
name: 5-finger-selection

validate
       Touch state: TouchMove and
       Touch limit: 3..5 and
       Distance between points: 50..250 and
       Distance between points: unchanged 10%

return
       Touch points
```

Appendix B: Ethics Approval

A scanned copy of the original ethics approval is provided for the following two user studies:

- "Testing Multi-user Multi-Touch Tabletop Applications" – File # 6384, and
- "A Gesture Definition Language for Cross-Platform Multi-Touch Applications" – File # 6400.

Appendix C: Co-Author Permission

This section provides the scanned copies of the co-authors written permission to use the content of the following publications in this thesis and to have this work microfilmed:

- *A Domain Specific Language to Define Gestures for Multi-Touch Applications.* In Proceedings of the 10th SPLASH Workshop on Domain-Specific Modeling, Reno/Tahoe, Nevada, USA, 2010.
    - *Co*-authors: Frank Maurer.
- *Tool Support for Testing Complex Multi-Touch Gestures.* In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, Saarbrücken, Germany, 2010.
    - *Co*-authors: SM Sohan, Jonathon Sillito and Frank Maurer.
- *A Language to Define Multi-Touch Interactions.* In Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces, Saarbrucken, Germany, 2010.
    - *Co*-authors: Frank Maurer
- *FitClipse: A Tool for Executable Acceptance Test Driven Development*. In Proceedings of 10th International Conference on Agile Processes and eXtreme Programming (XP 2009), Pula, Italy, 2009.
    - *Co*-authors: Yaser Ghanam, Shelly Park and Frank Maurer.

Appendix D: Adding new hardware/device support

Adding additional providers to the toolkit allows for applications to work on additional platforms. The source code described here represents the October 2010 CTP release of the Toolkit. The following steps (generically) outline how to add a provider for numerous touch-enabled platforms:

Step 1:

In Visual Studio, create a new C# file, preferably close to the device name.

Step 2:

It is important to understand the events that are supplied with the SDK of the platform you wish to support (and write the provider for). The key events to understand are typically related to "Touch down", "Touch up" and "Touch move". This example demonstrates how to write a provider for the Microsoft Surface, which uses the events "ContactDown", "ContactLeave" and "ContactChanged".

Step 3:

The 1st step in coding your provider is to import all the necessary references, which will include the following:

```
using Framework;
using TouchToolkit.Framework;
using TouchToolkit.Framework.Utility;
using TouchToolkit.GestureProcessor.Objects;
using TouchToolkit.Framework.TouchInputProviders;
```

This will allow you to use the necessary components of the linked GestureToolkit.Framework and GestureToolkit.GestureProcessor.

Step 4:

When Microsoft Surface SDK is used, the touch data can be retrieved from the SurfaceWindow object. However, different device SDKs may provide this data in different ways. So we create a private variable to keep the reference of the window object that we will use later to retrieve touch data.

```
private SurfaceWindow _window;

public SurfaceTouchInputProvider(SurfaceWindow window)
{
    _window = window;
}
```

In your application, this window is usually bound to the GestureFramework with the provider you create, which will be shown later.

Step 5:

Next, it is important to setup some manner of storing active touch points and touch information that is required for the Framework to manage multi-touch information.

Typically, the best manner is to use the dictionary implementation as shown below.

```
private Dictionary<int, TouchPoint2> _activeTouchPoints = new Dictionary<int, TouchPoint2>();
private Dictionary<int, TouchInfo> _activeTouchInfos = new Dictionary<int, TouchInfo>();
```

Step 6:

The next step, involves the linking of the events in the SDK to methods. These events are first linked to the private window provider.

```
public override void Init()
{
    // Add the necessary event handlers
    _window.ContactDown += ContactDown;
    _window.ContactChanged += ContactChanged;
    _window.ContactLeave += ContactLeave;
    _contactTarget.FrameReceived += new EventHandler<FrameReceivedEventArgs>(_contactTarget_FrameReceived);
    _contactTarget.EnableInput();
}
```

Step 7:

The UpdateActiveTouchPoints method, allows for the continuous stream of touch points provided by the hardware, to be updated in GestureToolkit.

```
104         public void UpdateActiveTouchPoints(TouchAction2 action, Microsoft.Surface.Presentation.ContactEventArgs e)
105         {
106             //Get the  point position from the ContactEventArgs (can optionally use e.Contact.getCenterPosition here for more accuracy)
107             Point position = e.GetPosition(GestureFramework.LayoutRoot);
108
109             //Create a new touchinfo which will be used later to add a touchpoint
110             TouchInfo info = new TouchInfo();
111
112             //Set the action type to the passed in action
113             info.ActionType = action;
114
115             //Set the position of the touchinfo to the previously found position from e
116             info.Position = position;
117
118             //Set the deviceid of the touchinfo to the id of the contact
119             info.TouchDeviceId = e.Contact.Id;
120
121             TouchPoint2 touchPoint = null;
122
123             //If it is contact down, we want to add the point, otherwise we want to update that particular point
124             if (action == TouchAction2.Down)
125             {
126                 //add the new touch point to the base
127                 touchPoint = base.AddNewTouchPoint(info, e.OriginalSource as UIElement);
128             }
129             else
130             {
131                 //add the new touch point to the base
132                 touchPoint = base.UpdateActiveTouchPoint(info);
133             }
134
135             // Update local cache
136             if (_activeTouchPoints.ContainsKey(info.TouchDeviceId))
137             {
138                 _activeTouchPoints[info.TouchDeviceId] = touchPoint;
139                 _activeTouchInfos[info.TouchDeviceId] = info;
140             }
141             else
142             {
143                 _activeTouchPoints.Add(info.TouchDeviceId, touchPoint);
144                 _activeTouchInfos.Add(info.TouchDeviceId, info);
145             }
146         }
147     }
```

Lines 107 - 119, illustrate capturing the position of the touch point, and creating a new touchinfo object containing the action, position and device id of the contact.

Lines 121 - 133, illustrate the addition or update of touch points. If a down contact is received, it means that a new touch point needs to be added whereas everything else (typically move, etc) should be updated.

Lines 136-145, highlight updating the local cache of active touch points and information.

Step 8:

The final and most important step is to handle the framechanged event for the SDK of your choice. Occasionally, this event may not be offered, in which case, a possible solution is to set a timer for ~ 30 msecs and capture all touches in that time frame to determine which event to raise in the SDK.

For this particular Microsoft Surface example, the SDK provides the necessary information to forgo the timer implementation.

Lines 53-54, turn the cached active touch information and points into a list, which is necessary to call the SingleTouched, MultiTouched and FrameChanged methods within GestureToolkit.Framework.

Lines 57 - 76, illustrate how to raise the appropriate events for GestureToolkit.

Lines 79 - 85, update any cached touch information and points and remove any points that are of type TouchUp, as there is no longer a touch to be associated with the information and point.

```
51          void _contactTarget_FrameReceived(object sender, FrameReceivedEventArgs e)
52          {
53              List<TouchInfo> touchInfoList = _activeTouchInfos.Values.ToList<TouchInfo>();
54              List<TouchPoint2> touchPoints = _activeTouchPoints.Values.ToList<TouchPoint2>();
55
56              // Raise "SingleTouchChanged" event if necessary
57              if (SingleTouchChanged != null)
58              {
59                  foreach (var touchPoint in touchPoints)
60                  {
61                      SingleTouchChanged(this, new SingleTouchEventArgs(touchPoint));
62                  }
63              }
64
65              // Raise "MultiTouchChanged" event if necessary
66              if (MultiTouchChanged != null)
67              {
68                  MultiTouchChanged(this, new MultiTouchEventArgs(touchPoints));
69              }
70
71              // Raise "MultiTouchChanged" event if necessary
72              if (FrameChanged != null)
73              {
74                  var frameInfo = new FrameInfo() { TimeStamp = e.FrameTimestamp, Touches = touchInfoList };
75                  FrameChanged(this, frameInfo);
76              }
77
78              // Clean up local cache
79              foreach (var touchInfo in touchInfoList)
80              {
81                  if (touchInfo.ActionType == TouchAction2.Up)
82                  {
83                      _activeTouchInfos.Remove(touchInfo.TouchDeviceId);
84                      _activeTouchPoints.Remov                    d);
85                  }
86              }
87          }
88
```

int TouchInfo.TouchDeviceId

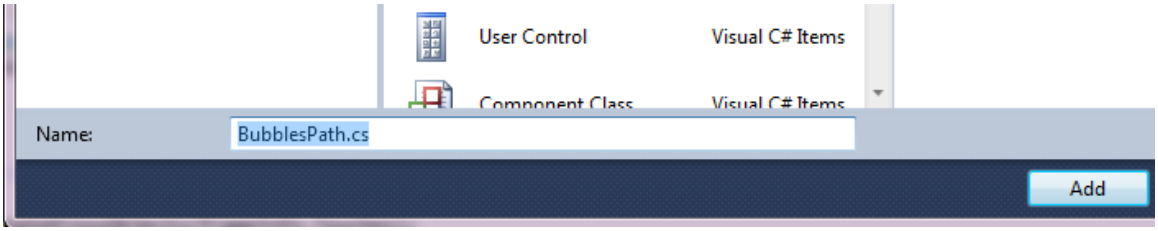
With these basic steps and Microsoft Surface, groundwork is set to for you to create your own provider.

Appendix E: Developing the *BubblesPath* touch feedback

Any touch feedback in GestureToolkit needs to implement the `ITouchFeedback` interface. The source code described here represents the October 2010 CTP release of the Toolkit. The following steps (generically) outline how to add a touch feedback:

Step 1:

In Visual Studio, create a new C# file and name it as BubblesPath.



Step 2:

Inherit from the `ITouchFeedback` interface. The interface requires you to implement two methods: `Init` and `FrameChanged.`

```
public interface ITouchFeedback: IDisposable
{
    void FrameChanged(FrameInfo frameInfo);

    void Init(Panel rootPanel, System.Windows.Threading.Dispatcher dispatcher);
}
```

The `Init` method is invoked only once by the framework during initialization. This is where we should write any initialization code. In this case, we are storing the references for the UI where the bubbles need to be rendered and the appropriate thread (i.e. dispatcher) which we should use for UI rendering. Also we are creating a timer object that we will use later for UI animation.

```
public void Init(Panel rootPanel, Dispatcher dispatcher)
{
    // Save display play
    _rootPanel = rootPanel;
    _dispatcher = dispatcher;

    // Start auto update
    var callback = new TimerCallback(UpdateUI);
    _uiUpdateTimer = new Timer(callback, null, 100, 100);

}
```

Step 3:

Next, we implement the FrameChanged method that is required by the ITouchFeedback interface. This method is invoked by the framework multiple times per second to update the UI.

```
public void FrameChanged(FrameInfo frameInfo)
{
    Action action = () =>
    {
        foreach (var touchInfo in
            frameInfo.Touches.Where(touchInfo => touchInfo.ActionType == TouchAction2.Move))
        {
            CreateProxyObject(touchInfo);
        }
    };
    _dispatcher.BeginInvoke(action);
}

private void CreateProxyObject(TouchInfo touchInfo)
{
    var po = new ProxyObject(touchInfo.Position.X, touchInfo.Position.Y, touchInfo.GroupId);
    _rootPanel.Children.Add(po);
    _proxyObjects.Add(po);
}
```

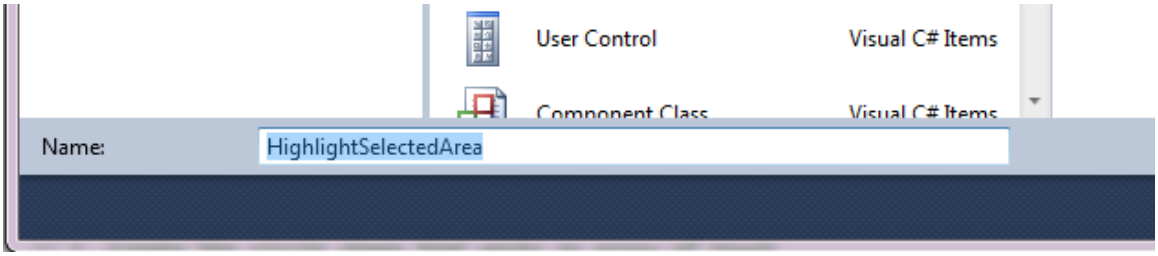
And that's all we need to create a new touch feedback. Note that the `ProxyObject` class is an UI element extended from the Grid class in WPF. The `ProxyObject` contains the implementation for the animated fadeout effect.

Appendix F: Developing the *HighlightSelectedArea* gesture feedback

Any gesture feedback in GestureToolkit needs to implement the `IGestureFeedback` interface. The source code described here represents the October 2010 CTP release of the Toolkit. The following steps (generically) outline how to add a gesture feedback:

Step 1:

In Visual Studio, create a new C# file and name it as HighlightSelectedArea.



Step 2:

Inherit from the `IGestureFeedback` interface. The interface requires you to implement two methods: `Init` and `FrameChanged.`

```
public interface IGestureFeedback
{
    /// <summary>
    /// Renders selected area on the specified canvas using the dispatcher thread
    /// </summary>
    /// <param name="dispatcher"></param>
    /// <param name="feedbackCanvas"></param>
    /// <param name="values"></param>
    void RenderUI(Dispatcher dispatcher, Canvas feedbackCanvas, List<IReturnType> values);
}
```

The `Render` method is invoked only once by the framework when a gesture is detected. This is where we should write code to start the UI rendering task. In this case, we check for valid set of inputs which is a collection of touch points that represents the touch path. Then we render a polygon on UI and use appropriate timers for the animated fadeout effect.

```
public void RenderUI(Dispatcher dispatcher,
    Canvas feedbackCanvas, List<IReturnType> values)
{
    _dispatcher = dispatcher;
    _feedbackCanvas = feedbackCanvas;

    var touchPaths= values.Get<TouchPaths>();
    if (touchPaths == null || touchPaths.Count == 0) return;

    DrawLassoArea(touchPaths[0]);
    StartAnimation();
}


private void DrawLassoArea(TouchPath touchPath)
{
    _polygon = new Polygon
                    {
                        Fill = new SolidColorBrush(Colors.LightGray),
                        Opacity = 0.7,
                        Tag = "SELECTED_AREA"
                    };

    foreach (var point in touchPath.Points)
    {
        _polygon.Points.Add(point);
    }

    _feedbackCanvas.Children.Add(_polygon);
}

private void StartAnimation()
{
    // Start auto update
    var callback = new TimerCallback(UpdateUI);
    _uiUpdateTimer = new Timer(callback, null, 40, 50);
}
```

And that’s all we need to create the gesture feedback that highlights a specified area.