Tool Support for Testing Complex Multi-Touch Gestures

Shahedul Huq Khandkar, S. M. Sohan, Jonathan Sillito and Frank Maurer
Department of Computer Science
University of Calgary, Canada
{s.h.khandkar, smsohan, sillito, frank.maurer}@ucalgary.ca

ABSTRACT
Though many tabletop applications allow users to interact with the application using complex multi-touch gestures, automated tool support for testing such gestures is limited. As a result, gesture-based interactions with an application are often tested manually, which is an expensive and error prone process. In this paper, we present TouchToolkit, a tool designed to help developers automate their testing of gestures by incorporating recorded gestures into unit tests. The design of TouchToolkit was informed by a small interview study conducted to explore the challenges software developers face when debugging and testing tabletop applications. We have also conducted a preliminary evaluation of the tool with encouraging results.

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

General terms: Verification, Design, Economics, Reliability, Standardization.

Keywords: Gestures, Multi-touch, Multi-user, Tabletop, Testing, Debugging.

INTRODUCTION

Automated software testing is important for developing robust software applications in a cost-effective manner. An automated test suite helps with the maintenance of applications by reducing the likelihood that changes introduce errors in previously working features of the application. Such testing is common in the software industry and it is well supported by tools in many domains. However, tool support for automating the testing of applications that use touch interface remains limited. In particular, applications that involve complex multi-touch gestures in a multi-user environment with lots of possible concurrent interactions are difficult to test in a cost effective way.

While recent operating systems and some multi-touch application frameworks provide gesture recognition for a set of common gestures, developers of tabletop applications often have to write code to recognize and respond to application specific gestures (e.g. [3]). Gesture recognition code can be complex and difficult to correctly implement [4]. As a step toward improving tool support for automating testing that involves gesture recognition code, we have developed a framework called TouchToolkit. TouchToolkit is a hardware abstracted gesture recognition tool with a test automation component. This proof of concept tool can automate the testing of complex multi-touch gesture recognition. When using this tool, developers can record complex multi-touch gestures that can be used as part of unit tests.

The design of the TouchToolkit framework was informed by a small study in which we interviewed tabletop application developers to explore the challenges they face in debugging and testing of tabletop applications. In this study, we found that developers rely on manual testing for their tabletop interfaces because today's tool support for test automation is limited. Also, behavioral differences between the actual hardware and the device simulator forced them to frequently move between their development workstation and a tabletop computer in order to ensure that the application is working as expected.

As a preliminary evaluation of the test component of TouchToolkit, we carried out a user study to explore the toolkit's usability and applicability. Participants were experienced tabletop application developers in a lab environment and familiar with unit testing practices. We found that all of the participants could successfully write automated test code using this toolkit after a demonstration. Our participants also provided encouraging feedback about the record/replay approach of TouchToolkit.

In reporting this work, this paper makes two key contributions. First, we present a list of commonly experienced debugging and testing difficulties associated with tabletop applications. These can be used to inform the tool development. Second, TouchToolkit, demonstrates that record/replay based testing can be used as a possible paradigm for automating aspects of tabletop interface testing.

The remainder of this paper is organized as follows. We start by comparing our work to earlier work. Next, we discuss the exploratory study along with our main findings from the study. We then discuss the TouchToolkit architecture and how this tool can be used for test automation. Next, we present a preliminary evaluation of the tool and demonstrate the test framework. Finally, we discuss the limitations and future work that is needed before presenting a brief summary of the paper.

ITS'10, November 7-10, 2010, Saarbrücken, Germany.

RELATED WORK

TouchToolkit has a hardware abstraction module and a test framework that uses this module. In this section, we compare our work with existing work on hardware independence and testing of multi-touch applications.

Hardware Independence

Multi-touch tabletop 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 tabletop applications across different devices.

Tabletop hardware vendors provide tool support for the development and testing of tabletop applications specific to their device. 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 are not interoperable, as a result the applications developed using these SDKs cannot be readily used on other platforms.

Echtler [16] proposed 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 [17] is a framework for rapid prototype development of multi-touch applications. It provides consistent touch data to the application widget layer from different touch devices. Pointer [13] also proposed a hardware abstraction approach. OpenInterface [9] provides an environment to work with simulated components from a component repository (e.g. speech recognition, video "finger tracker").

In TouchToolkit, we have a similar approach as PyMT and Pointer in the device abstraction part. However, TouchToolkit also has a virtual hardware simulator similar to OpenInterface to simulate actual device inputs for testing and debugging multi-touch applications.

Recent versions of some operating systems are providing native support for touch based inputs. 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 in their applications. TouchToolkit abstracts out the input layer from the application to solve this problem for the supported devices & operating systems. In addition, TouchToolkit provides a gesture definition language [15] to edit existing or define new gestures whereas most frameworks an operating systems (e.g. Windows 7) comes with a fixed set of gestures.

Also, there is tool support to bring limited multi-touch capabilities to regular computers. For example, Mouse 2.0 [21], Multi-Mice [19] and Multi-Touch Vista [1] can add the ability to use multiple mice as touch points. Similarly, Johnny Lee's Wiimote project [18] can turn any regular display into a multi-touch enabled system using Infrared pens. However, using mice or pens to mimic finger touches can only be used for a maximum of two active touch points which is significantly less than the available tabletop devices. To overcome this limitation, we implemented a record and replay feature that can simulate actual multi-touch interactions using a virtual device.

Testing of Multi-Touch Applications

Interaction designers are not generally domain experts in gesture recognition [4]. To simplify the process, there are a number of frameworks and toolkits available for pattern and gesture recognition, such as Weka [5] and GT2K [6]. While these are mostly libraries of techniques, tools are also available for designing gestures such as MAGIC [7] and quill [8]. MAGIC uses recorded data as samples and quill also uses recorded data for training purpose. In contrast, TouchToolkit provides a gesture definition language [15] that includes multi-touch and multi-step gestures. This allows the developers to easily test different steps of the gesture recognition process and edit them as necessary.

DART [11] 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 [12] is another testing tool for interaction designers that can wrap input device APIs and provide 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.

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 [13] also proposed a record and replay based automated testing approach.

TouchToolkit follows a similar approach with an extension that it allows developers to debug and write automated tests of applications independent of underlying hardware. This also allows to simulate multi-user scenarios using multiple

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

recorded interactions and helps to overcome the need of an actual device to a great extent.

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 [2], Project White [3], Selenium [4] and QFTest [5] 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[6] 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 [7] applications. Vimov [8] 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.

EXPLORATORY STUDY

We conducted a set of interviews to explore the common testing and debugging challenges that developers face while developing tabletop applications.

Study Participants

We interviewed 3 participants who developed tabletop applications in a university lab environment. All the 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. In this paper we refer to those participants as P1, P2 and P3. Table provides a summary of their experience levels.

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.

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:

1. Please tell me how you tested your application.
2. Is there anything that was difficult to test?
3. 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 [20]. In the second stage of our analysis, we grouped the coding into five categories that capture the main challenges our participants faced.

Findings

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

Tabletop Application Testing Workflow. Figure 1 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".

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.

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

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

Table 1: Participants' Experience

[2]http://msdn.microsoft.com/en-us/library/dd286726.aspx
[3]http://white.codeplex.com/white
[4]http://seleniumhq.org/
[5]http://www.qfs.de
[6]http://www.froglogic.com/
[7]http://cocoatouchapps.com
[8]http://www.vimov.com/

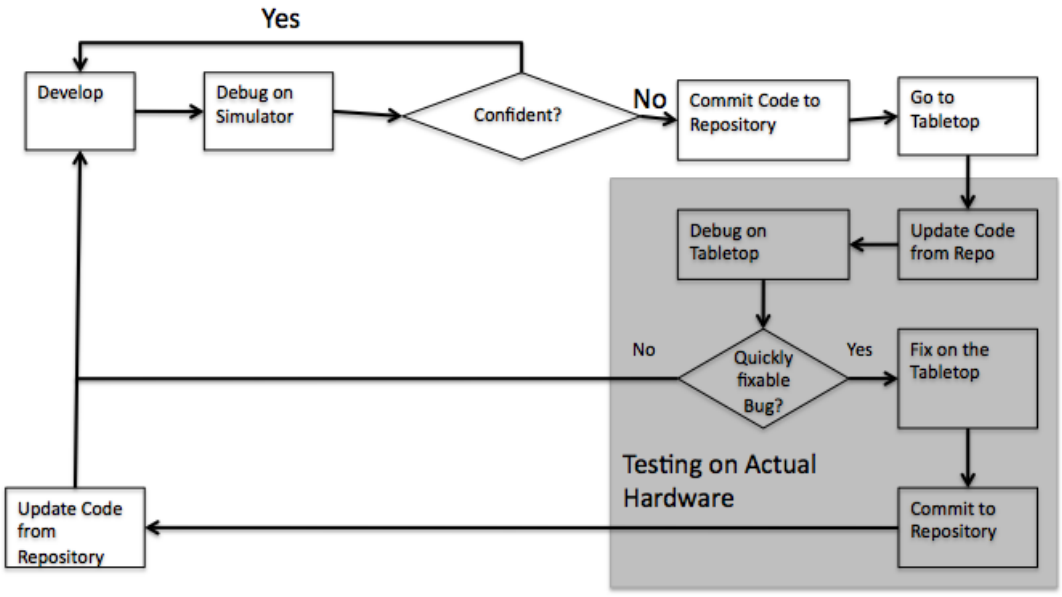

Figure 1: An example of a tabletop application testing workflow

tabletop devices with different physical sizes. So, P3 had to manually test on both tables whenever there was a significant change in the application.

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

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.

Feature	Simulator	Actual Table
# 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 2: Feature comparison between a Simulator and an Actual Tabletop

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

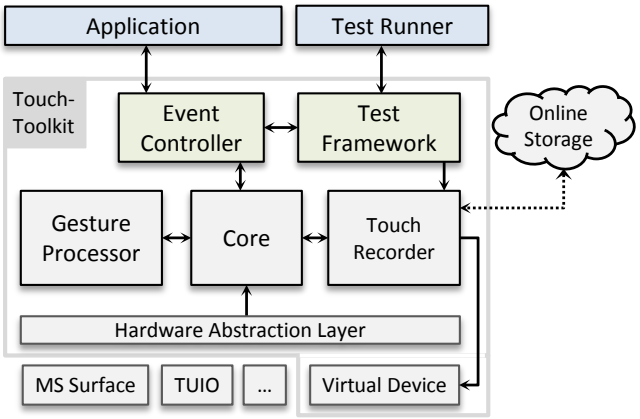

Figure 2: Components of the TouchToolkit Framework

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.

TOUCH TOOLKIT ARCHITECTURE

To address some of the challenges faced by the participants in our exploratory study, we developed TouchToolkit. Specifically, TouchToolkit aims to address four development and testing challenges: (1) automated unit testing, (2) debugging, (3) testing multi-user scenarios, and (4) device independence.

TouchToolkit provides a touch interaction record and playback system that helps simplify testing and debugging during development as well as automating tests to validate multi-touch interactions. It also provides a device independent API that can be used to add new device support and a gesture recognition system. The tool can be used in applications that use Windows Presentation Foundation (WPF) and also in Silverlight-based web applications. The entire source code and documentation for the tool is available on the project web site[9]. The video with this paper shows how the tool is used.

The component diagram of the TouchToolkit framework is shown in Figure 2. The four 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 touch recorder that stores the raw data from the hardware abstraction layer and (4) the core that acts as a bridge among the components. The test framework executes the automated test scripts. A test script validates the gesture recognition code against one or more recorded touch interactions. Gestures are defined using a gesture definition language [15] provided by the toolkit which allows to define multi-touch gestures that may include touch interactions in multiple steps. We describe the key components related to automated testing in the subsequent sections.

Hardware Abstraction Layer

Like other tools (e.g., [16]), TouchToolkit 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. This interface can be implemented for most multi-touch enabled hardware platforms and we currently have implementations for Microsoft Surface, SMART Tabletop, Windows 7, AnotoPen and the TUIO protocol[10]. TouchToolkit also has an implementation of this interface for a virtual hardware device that can be used to playback recorded interactions and run automated tests. Multiple devices can be active at the same time and the framework supports changing devices at runtime. This is useful in an environment where additional devices (e.g., AnotoPens) need to be connected while the application is running.

While all multi-touch devices provide a set of common inputs like coordinates for touch points, additional inputs are also available that are often unique to a particular device. For example, the Microsoft Surface provides finger direction data, Diamond Touch can identify users and so on. To maintain hardware independence, the hardware abstraction layer ensures that basic touch information is processed in a common format. However, hardware specific data can also be provided and that data is passed, through the core component, to other parts of the toolkit and to applications.

When an application uses a gesture that needs a device specific input, it should also provide an alternate option (i.e. another gesture) as a fall back. For example, an application can have a single finger "rotate" gesture. This gesture requires the "touch direction" data that is available in Microsoft Surface but not in many other multi-touch devices. As an alternate, the application may also provide the two finger rotate gesture that uses the basic inputs to comply with different devices. In most cases, the TouchToolkit framework can be used to determine whether an application that uses the Toolkit has any device specific dependencies.

Touch Interaction 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 cache it locally to improve performance. This allows automatic synchronization of data between developer machines and actual devices. The data is stored in an XML format (see Figure 3). The recorder can record and store interactions from any device that is supported by the hardware abstraction layer, including basic touch information (i.e., co-

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

```
<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 3: An XML code fragment representing a part of a touch interaction. Some details are omitted for clarity. Each interaction is recorded as a *frame* which contains one or more *touches*.

ordinates, touch ID) and any additional device specific data provided by the hardware.

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.

Test Framework

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 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 can be defined in the gesture definition language, including complex multi-touch gestures that involve touch interactions with multiple steps. More details about the gesture processor and the gesture definition language can be found in [15].

Figure 4 shows the work flow of an automated test in TouchToolkit. 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

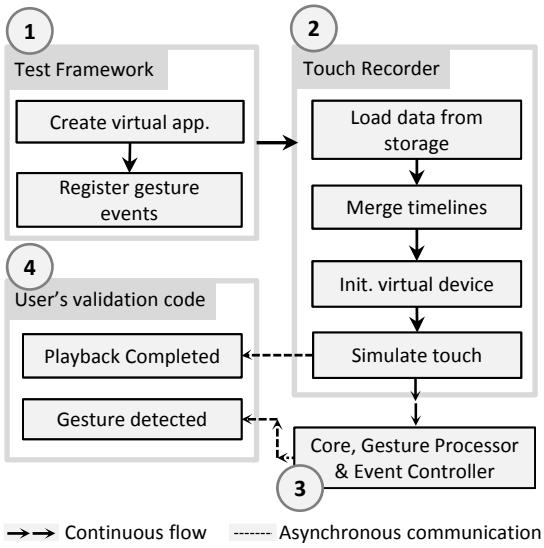

Figure 4: Work flow of automated test framework

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. An example test using this framework is given later in this paper.

TESTING USING TOUCH TOOLKIT

Automated unit testing is a well known way to increase the effectiveness, efficiency and coverage of software testing [14]. 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.

Writing Unit Test for Gestures

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 TouchToolkit 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 TouchToolkit 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:

1. **expectedGestureName:** The name of the gesture to detect (e.g., Zoom).

```
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 5: Example unit test code using Touch Toolkit

2. **savedInteraction:** The identifier or the recorded interaction that should produce the expected gesture.

3. **gestureDetectedCallback:** The method to call when the gesture is detected. Developers can write custom validation code inside this method.

Figure 5 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 6: User defined gesture validation code

Figure 6 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 TouchToolkit allows developers to write unit test code for validating multi-touch interactions.

Step by Step Debugging.

In addition to unit testing, the record and playback feature can also be used to debug complex touch interactions. Using TouchToolkit one can set breakpoints in an IDE to debug touch interaction related code. Also TouchToolkit provides a speed control that developers can use to speed up the debugging process. As a result, a developer can quickly move to a point of interest during debugging as well as investigate an interaction in slow motion. Currently developers can use the remote debugging feature of Microsoft Visual Studio to debug the application on the actual hardware from their workstation. However, this requires developers to move between the actual hardware and their workstations to interact with the application and to see the live debug outputs. TouchToolkit enables the developers to achieve the same goal without using the actual hardware.

Testing Multi-User Scenarios.

In our exploratory study we found that a key challenge for developers is testing and debugging interactions or scenarios involving multi-user interactions. To mitigate these challenges, TouchToolkit supports the execution of different interactions in parallel, even if those interactions were recorded separately. The result is that developers do not need to involve multiple testers every time they need to test concurrent multi-user interactions. At present, we do not have a visual tool for editing the time lines of recorded touch interactions but a knowledgable programmer can use the API methods to achieve this effect.

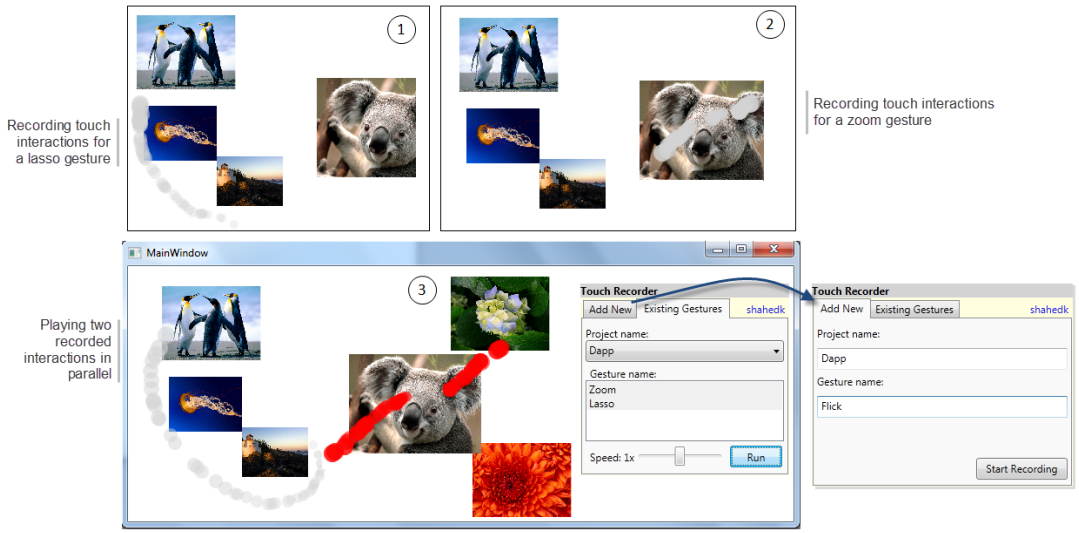

Figure 7: TouchToolkit: Simulating multi-user touch interactions

Figure 7 shows a screenshot of the TouchToolkit record and playback window for a multi-touch photo viewer application. The "Add New" tab in the debug window allows developers to record new touch interactions. The "Existing Gestures" tab allows developers to playback recorded touch interactions individually or in parallel. In step 1, a "Lasso" gesture is recorded. Next, in step 2 the "Zoom" gesture is recorded. Once these two are recorded separately, as shown in step 3, TouchToolkit allows one to run both of them in parallel to simulate multi-user scenario. To differentiate the constituent interactions of a combined playback, TouchToolkit applies different color codes to the individual interactions as shown in Step 3. A speed controller on the playback panel allows the playback speed to be adjusted.

```
string[] expectedGestures = {"Zoom", "Lasso"};
string[] savedInteractions = {"Interaction01", "Interaction02"};

GestureTestFramework.Validate(expectedGestures, savedInteractions,
    ...
```

Figure 8: Using multiple recorded touch interactions in a single Unit Test

This same combined playback can be used inside unit test code to test for multi-user multi-touch interactions. To achieve this, a developer needs to specify all the desired gestures while calling the Validate method. Figure 8 shows an example test code fragment. This code tests if "Zoom" and "Lasso" gestures are detected after executing two saved interactions named "interaction01" and "interaction02" in parallel. As shown in this example, the test framework hides the complexity of dealing with multi-user, multi-touch scenarios and provides a high level API for developers.

EVALUATION

We performed a 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.

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:

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

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

Findings

At a high level, our preliminary evaluation has suggested that the TouchToolkit 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.

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

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 TouchToolkit'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.

Findings from Task 2: Record the Resize Touch Interaction. This task requires one to do the following:

1. Write the appropriate code to show the recording panel in the debugger.
2. 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.

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, TouchToolkit 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 which 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 simplified 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 TouchToolkit, could write an automated test for the interaction. We also found that templates can help reduce the learning curve for new developers.

Findings from Interviews. 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 TouchToolkit allowed him to test and debug without moving between the tabletop device and his workstation, which he believes will help him 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 that 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 TouchToolkit to be the most useful for her development work.

LIMITATIONS of TOUCH TOOLKIT and STUDY

The goal of our work has been to develop tool support for Microsoft .NET based tabletop applications. The tool is well integrated with the Microsoft Visual Studio IDE and test framework. This makes it easy to use for any application that runs on the same platform. On the other hand, it does not support application development with non Microsoft languages and touch frameworks.

The purpose of the preliminary user study was to generally evaluate the approach taken in TouchToolkit. 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.

TouchToolkit was intended to be used for automated testing of multi-user multi-touch based interaction. However, some other tabletop input techniques such as Smart Tags or physical object based interactions are not yet supported by this tool which limits its usefulness in testing some kinds of applications.

FUTURE WORK

In future, we plan to extend the TouchToolkit in the following ways:

- To evolve TouchToolkit as a comprehensive multi-touch application test framework we need to include support for other devices and input techniques.
- In addition to the record/replay based testing we plan to provide support for programmable user interface automation testing for multi-touch applications.

CONCLUSION

We have introduced TouchToolkit which provides tool support for debugging and testing multi-user, multi-touch tabletop applications. This tool offers a device independent record and replay based framework. Using this framework, developers can write automated tests for complex gestures that run on their development machine. Also, these automated tests can be used as a part of regression testing in a continuous integration process. To reduce the learning curve,

we present an event based API that most UI developers are already familiar with. Also, we provide IDE integration for Microsoft Visual Studio including test code templates and test execution as a part of the test process. We believe this automated test framework can help to produce real-world multi-touch tabletop applications with good quality. The results of our evaluation of the toolkit are preliminary, but encouraging.

ACKNOWLEDGMENTS

We would like to thank Teddy Seyed and Andy Phan for their contribution on developing providers for different hardware platforms.

REFERENCES

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

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

3. 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.

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

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

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

7. 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.

8. 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.

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

10. Kaltenbrunner, M., Bovermann, T., Bencina, R., and Costanza, E. TUIO: A protocol for table-top tangible user interfaces. In Proceedings of Gesture Workshop 2005, 2005.

11. 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.

12. 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.

13. Bastea-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

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

15. Khandkar, S. and Maurer, F. 2010. A Domain Specific Language to Define Gestures for Multi-Touch Applications. In Proc. of the 10th SPLASH Workshop on Domain-Specific Modeling, Reno/Tahoe.

16. Echtler, F. and Klinker, G. 2008. A multitouch software architecture. In Proceedings of the 5th Nordic Conference on Human-Computer interaction: Building Bridges (Lund, Sweden, October 20 - 22, 2008). NordiCHI '08, vol. 358. ACM, New York, NY, 463-466.

17. Hansen, T. E., Hourcade, J. P., Virbel, M., Patali, S., and Serra, T. 2009. PyMT: a post-WIMP multi-touch user interface toolkit. In Proceedings of the ACM international Conference on interactive Tabletops and Surfaces (Banff, Alberta, Canada, November 23 - 25, 2009). ITS '09. ACM, New York, NY, 17-24.

18. Lee, J.C. Hacking the Nintendo Wii remote, IEEE Pervasive Computing, vol. 7, no. 3, July-Sept 2008, 39-45.

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

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

21. Villar, N., Izadi, S., Rosenfeld, D., Benko, H., Helmes, J., Westhues, J., Hodges, S., Ofek, E., Butler, A., Cao, X., and Chen, B. 2009. Mouse 2.0: multi-touch meets the mouse. In Proceedings of the 22nd Annual ACM Symposium on User interface Software and Technology (Victoria, BC, Canada, October 04 - 07, 2009). UIST '09. ACM, New York, NY, 33-42.