Poster: Real-Time Gesture Detection for Multi-Touch Devices

Francisco R. Ortega[*] Florida International University
Armando Barreto[†] Florida International University
Naphtali Rishe[‡] Florida International University
Malek Adjouadi[§] Florida International University
Fetemeh Abyarjoo[¶] Florida International University

Abstract

We are motivated to seek a fast and accurate multi-touch gesture detection algorithm that can be utilized for 3D navigation. Our current approach tries to solve the online gesture detection for multi-touch devices for translation, rotation and zooming gestures.

Index Terms: H.5.2 [Multi-Touch Gesture Detection]: Real-Time Gesture Detection Algorithm—Multi-Touch

1 Introduction

Our work is inspired by previous contributions such as the $1 algorithm[8] and the Rubine algorithm[5]. The contributions mentioned have shown that simple approaches can be quite effective and easier to implement in contrast to Hidden Markov Models[6] or neural networks[4]. The primary motivation is to find a fast algorithm for a high-demanding 3D navigation system using multi-touch devices. The Rubine algorithm[5] showed specific features that can be used for stroke recognition and Wobbrock et el.[8] demonstrated an efficient approach to detect different gestures. Our goals are also aligned with the problem statement written by Greg Hamerly[1] for interaction with a generalized multi-touch system. We believe that our work can contribute to the development of real-time solutions for the algorithm development of the post-WIMP era. Particularly, the use of real-time gesture detection for 3D synthetic worlds. For further review of the state of the art, please see[8] and for a in-depth study of post-WIMP devices and interfaces, see[1].

2 Exposition

We are currently developing our multi-touch system using C++11 with Microsoft Visual Studio 2012 and running on a Windows 7 64-bit machine. We are also using the Parallel Patterns Library from Microsoft to keep all data structures thread safe. For the touch data, we use the raw data provided by the windows touch drivers[3]. The raw touches they gives us the flexibility to test different multi-touch gesture recognition approaches and build new gestures.

We use raw touch data[3] (e.g., Windows, iOS) to capture data points stored in a set called **trace**. A **trace** starts when a user presses with one finger and continues until the user removes the finger from the device. Figure 1, shows a rotation gesture with two fingers in our test system. This constitutes two traces. Each **trace** has a a unique identifier (**id**) and contains a set of points with 2D Coordinates (**x**,**y**) and a timestamp **t**, for each point. The general events provided are the initial touch of a trace (TOUCHDOWN), the translation of the trace (TOUCHMOVE) and the end of the touch (TOUCHUP). A **trace point** structure contains coordinates **x** and **y**, timestamp $\mathbf{t_0}$, count **c** (indicating how many continuos points are found in the same location), boolean **p** (indicating if the touch-point was already processed) and the last timestamp $\mathbf{t_1}$. This we call TOUCHPOINT. We also keep an additional data structure with the name of TRACE. This contains **id** for the unique identifier. The initial timestamp $\mathbf{t_i}$ for the trace, the final timestamp $\mathbf{t_f}$ and boolean **d** to see if the trace is ready for deletion. For additional information on how Windows 7 handles the touch driver, please see [3].Our approach concentrates in the online gesture detection using multi-touch devices. However, we believe that our approach combined with finite state machine will yield a better result.

It is important to keep the touch events and gesture detection in different thread processes[7]. Therefore, all the active traces are stored in a concurrent hash map and a concurrent arraylist (vector) to keep the set of touch points of a given trace. Once data points are processed, they can be safely removed. The advantage to have them in different threads (other than speed) is to have a real-time approach to gesture detection. We define a buffer with a maximum size of **windowSize**. This means that when the buffer is full, it needs to perform the gesture detection while still allowing touch events to execute. In our experiment we have found that the windowSize works fine when it has 50 data points for each trace. This means, that a four-finger gesture will need a windowSize of 200.

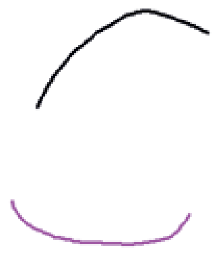
Figure 1: Rotation Gesture

TOUCHDOWN is the initial event that fires when a trace begins. This means that a user has pressed a finger in the multi-touch device. The event fires for each new trace. There is room to further improve the performance of the gesture detection system by creating additional threads for each trace. The first trace is stored in the vector **vtrace**, during this event. The vector is kept in a hash map **mtrace**, which contains a collection of all traces. The timestamp for t_0 and t_1 will be identical for the first trace.

As the user moves across the screen (without lifting the fingers) the event that fires is TOUCHMOVE (Listing 1). In line 3 of Listing 1, we invoke a method called removeNoise passing the given trace and the previous traces. This algorithm should be modified according to how the "noise" is defined. We consider that "noise" occurs if a new touch point is within $\pm\mathbf{d}$ of a previous point where **d** is a pre-calculated value depending on the size of the screen (e.g., 2). If the noise is true, then we just update the counter **c** and the timestamp $\mathbf{t_1}$ for this trace as shown in lines 5-6. Otherwise, we add this

[*]email: forte007@fiu.edu
[†]e-mail:barretoa@fiu.edu
[‡]email: ndr@acm.org
[§]email: adjouadi@fiu.edu
[¶]email: fabya001@fiu.edu

[1]`http://cs.baylor.edu/~hamerly/icpc/qualifier_2012/`

IEEE Symposium on 3D User Interfaces 2013
16-17 March, Orlando, FL, USA
978-1-4673-6098-2/13/$31.00 ©2013 IEEE

touch point to the vector. At the end of the procedure, in line 13, we update the map (Note that depending on the implementation and language, you may skip this last step).

Algorithm 1 TOUCHMOVE

```
 1: $trace \leftarrow TRACE(id)$
 2: $vtraces \leftarrow mtraces.find(id)$
 3: $noise \leftarrow removeNoise(trace, vtrace)$
 4: if $noise$ then
 5:     $trace.c += 1$
 6:     $trace.t_1 = trace.requestTimeStamp()$
 7: else
 8:     $trace.t_1 \leftarrow trace.requestTimeStamp()$
 9:     $trace.t_0 \leftarrow vtraces[length-1].t_0$
10:     $vtrace \leftarrow mtraces.getValue()$
11:     $vtrace.push\_back(traces)$
12: end if
13: $mtraces.insert(id, vtrace)$
```

The final event is when the user removes the finger. Since the gesture detection algorithm may still be running, the data is marked for deletion only. Once Algorithm 2 finishes, the process can safely delete all the data touch points that have been used.

Algorithm 2 detects translation(swipe), rotation, zooming (pinch). Once the buffer is full, we split a window of touch data into two lists named **top** and **bottom**. This creates an initial and a final snapshots to work with.

Algorithm 2 requires pre-computed values, grip points and average touch points(described below) before it can execute. The choices to pre-compute the values can be done in the TOUCHMOVE event or by firing a separate process before Algorithm 2 starts. If the latter is chosen, then a GPU approach is recommended. For now, we are using the touch events for the pre-computed values. We have stored the values for Algorithm 2 in the **top** and **bottom** structures respectively.

The features identified in each gesture are **grip**, **trace vector**, **spread** and **angle rotation**. A **grip** is the average of all points in each top and bottom lists. A **trace vector** is a **trace** minus the **grip**, as shown in Algorithm 2 lines 11 through 14. The **spread** is calculated in lines 15-18 of Algorithm 2 as the average distance between the **grip point** and **the touch vector**. The **angle rotation** is given by the average of the angles obtained by atan2[2], which is the angle between the final touch vector and the initial touch vector.

To select the correct gesture the algorithm finds the highest value from the three distance variables: swipeDistance, rotDistance, or zoomDistance. The definition of the swipe distance is the spread of the first trace and the grip. The rotate distance is calculated to be the arc length, which is given by the the radius of the swipe distance and the average angle shown in lines 19-20 of Algorithm 2.It is important to note that atan2[2] values range between $\pm\pi$. This is why there is a factor of 2 in line 26. Finally, the zoom distance is defined as the difference between the average final spread distance and the average initial spread distance.

3 Conclusion

We have shown, in agreement with previous contributions[5, 8], that gesture detection can be solved without complex methods like Hidden Markov Models or neural networks. Our approach finds, in real-time, rotation, translation and zooming gestures for multi-touch devices.

Our future work will attempt to develop mechanisms for detection of new gestures in real-time. We also expect to look into GPU computing and Cloud computing to find innovative ways to perform a big set of faster recognition of gesture from a larger set. Our final

Algorithm 2 GestureDetection

```
 1: $top \leftarrow traces.getTop(windowSize)$
 2: $bottom \leftarrow traces.getBottom(windowSize)$
 3: $tGrip.x \leftarrow top.getGrip.x$
 4: $tGrip.y \leftarrow top.getGrip.y$
 5: $bGrip.x \leftarrow bottom.getGrip.x$
 6: $bGrip.y \leftarrow bottom.getGrip.y$
 7: $spread.x \leftarrow iTrace[1].x - iGrip.x$
 8: $spread.y \leftarrow iTrace[1].y - iGrip.y$
 9: $swipeDistance \leftarrow sqrt(spread.x^2 + spread.y^2)$
10: for $t = 1$ to $traces.Count$ do
11:     $i.x \leftarrow tTrace[t].x - tGrip.x$
12:     $i.y \leftarrow tTrace[t].y - tGrip.y$
13:     $f.x \leftarrow bTrace[t].x - bGrip.x$
14:     $f.y \leftarrow bTrace[t].y - bGrip.y$
15:     $di \leftarrow sqrt(i.x^2 + i.y^2)$
16:     $df \leftarrow sqrt(f.x^2 + f.y^2)$
17:     $iSpread \leftarrow iSpread + di$
18:     $fSpread \leftarrow fSpread + df$
19:     $angle \leftarrow atan2(f.y - i.y, f.x - i.x)$
20:     $rotAngle \leftarrow rotAngle + angle$
21: end for
22: $iSpread \leftarrow iSpread/traces.Count$
23: $fSpread \leftarrow fSpread/traces.Count$
24: $rotAngle \leftarrow rotAngle/traces.Count$
25: $zoomDistance \leftarrow fSpread - iSpread$
26: $rotDistance \leftarrow rotAngle/360.0 * 2 * \pi * swipeDistance$
27: return Gesture With Highest Distance
```

goal is to use our gesture detection method with 3D worlds for navigation purposes. We will compare our method with other gesture detection algorithm.

Acknowledgements

This work was sponsored by NSF grants HRD-0833093, and CNS-0959985. Mr. Francisco Ortega is the recipient of a GAANN fellowship, from the US Department of Education, at Florida International University.

References

[1] D. A Bowman, E. Kruijff, J. J. LaViola, and I. Poupyrev. *3D user interfaces: theory and practice*. Jan. 2005.

[2] F. Dunn and I. Parberry. *3D Math Primer for Graphics and Game Development, 2nd Edition*. A K Peters/CRC Press, 2 edition, Nov. 2011.

[3] Y. Kiriaty, L. Moroney, S. Goldshtein, and A. Fliess. *Introducing Windows 7 for Developers*. Microsoft Pr, Sept. 2009.

[4] J. Pittman. Recognizing handwritten text. In *Human factors in computing systems: Reaching through technology (CHI '91)*, pages 271–275, New York,NY, 1991.

[5] D. Rubine. Specifying gestures by example. *ACM SIGGRAPH Computer Graphics*, 25(4):329–337, 1991.

[6] T. Sezgin and R. Davis. HMM-based efficient sketch recognition. *Proceedings of the 10th international conference on Intelligent user interfaces (IUI '05)*, 2005.

[7] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, 1 edition, Feb. 2012.

[8] J. Wobbrock and A. Wilson. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*, 2007.