HCIRA Project #2 Final Report

Yu-Peng Chen

Abstract—The goal of this project is to examine the performance of the $P recognizer on a new dataset called the Zhuyin Multistroke Gestures (ZMG) dataset. I also investigated whether familiarity with the Zhuyin symbols may affect the recognition accuracy. I implemented the $P recognizer and evaluated it with offline user-dependent recognition tests. The recognition tests were run on both the Mixed Multistroke Gestures (MMG) dataset and the ZMG dataset my classmate, Katarina Jurczyk, and I collected. Results from user-dependent random-100 tests on the two datasets are 99.27% (MMG) and 87.53% (ZMG). I also ran the ZMG dataset through the GHoST heatmap toolkit and extracted a few insights about users' gesture articulation. I found that familiarity with the Zhuyin symbols has no significant effect on recognition accuracy. Surprisingly, the $P dollar recognizer performed slightly better on the gestures drawn by participants who are not familiar with Zhuyin, compared to the gestures drawn by participants who are familiar with Zhuyin. I conclude that the inherent similarities between some of the Zhuyin symbols make it challenging for the $P recognizer to achieve an accuracy that is as high as the one (i.e., 99.27%) on the MMG dataset.

1 Introduction

THE purpose of this project is to implement the $P algorithm and examine the performance of the $P recognizer on a new dataset called the Zhuyin Multistroke Gestures (ZMG) dataset. Zhuyin is the phonetic system for learning how to speak Taiwanese Mandarin. This phonetic system is the first thing children in Taiwan learn about in elementary school. It is also used for entering Chinese on smartphones and computers, as shown in Fig. 1. I am interested in seeing if the $P recognizer can be used to recognize these symbols that I have learned as a child. The $P recognizer was implemented in Java, including four different applications: 1) an application for running offline tests on the Mixed Multistroke Gestures (MMG) dataset [1], 2) an application for running offline tests on the ZMG dataset, 3) a graphical user interface (GUI) for live demo, and 4) a GUI for data collection. I tested the $P recognizer on two datasets: 1) the MMG dataset [1], provided on the $P recognizer homepage, and 2) the ZMG dataset, collected from 12 users. For data collection, I collaborated with one of my classmates, Katarina Jurczyk. I collected data from 6 users who are all from Taiwan while Katarina collected data from 6 users who have no prior experience with any of the Zhuyin symbols. For this project, I only ran user-dependent tests on the datasets. The user-dependent recognition results for the two datasets are 99.27% on the MMG dataset and 87.53% on the ZMG dataset. I found that familiarity with the Zhuyin symbols has no significant effect on recognition accuracy. Surprisingly, the performance of $P on the data from 6 users who are not familiar with Zhuyin was slightly better than its performance on the data from 6 users who are familiar with Zhuyin. Additionally, I also used the GHoST heatmap toolkit [2] to analyze the ZMG dataset, and then examined the resulting heatmap for a few insights into how users articulated the gestures they made. Based on the results, I conclude that the inherent similarities between some of the Zhuyin symbols make it challenging for the $P recognizer to achieve an accuracy that is as high as the one (i.e., 99.27%) on the MMG dataset.

Fig. 1. The Zhuyin keyboard on an iPhone SE. Users can use these symbols to enter Chinese on the smartphone.

2 Related Work

The pervasiveness of mobile touchscreen devices presents opportunities for using pen or finger gestures in user interfaces. To enable rapid prototyping for user interface prototypers, Wobbrock et al. [3] presented a $1 recognizer that is easy to implement with only 100 lines of code. By comparing the $1 recognizer with Dynamic Time Warping (DTW) [4] and the Rubine classifier [5] in user-dependent scenarios, the authors found that the $1 recognizer achieves over 97% accuracy with only one training template and 99% accuracy with more than three training templates. They showed that the $1 recognizer achieved nearly identical results to DTW and performed better than the Rubine classifier. But $1 can only handle unistroke gestures. Anthony and Wobbrock [6] presented the $N recognizer that extends the $1 recognize to support multistroke recognition by considering all the possible combinations of stroke order and direction. The authors tested the $N recognizer with high school students writing algebra symbols in a math tutor prototype. They showed that $N achieved 96.6% accuracy with 15 training templates in a real-world scenario. However, $N needs to generate all possible permutations of a given multistroke, which can result in an explosion in both the required memory space and execution time. Vatavu et al. [7] developed the $P recognizer to address this problem by considering gestures as clouds of points. The authors showed that $P is more accurate than $N and requires

considerably less memory than $N. By discarding the chronological order of drawn points, the $P recognizer can ignore aspects such as the number of strokes, stroke order, and stroke direction. Such implementation allows $P to reduce the time and space complexity of $N while achieving a better accuracy on multistroke gestures.

Additionally, to design a better gesture set and develop a more accurate recognizer, it is important to understand users' gesture articulation patterns. Vatavu et al. [2] presented the GHoST toolkit for generating gesture heatmaps. By utilizing color maps, the toolkit enables the visualization of the variation of local features along the gesture path. The authors evaluated the gesture heatmaps on three public datasets and showed that the heatmaps can help understand gesture recognition errors, characterize users' gesture articulation patterns, and uncover user perception of execution difficulty. In this project, the GHoST toolkit is used to extract user articulation insights.

3 Dataset Details

3.1 Mixed Multistroke Gestures (MMG) dataset

The Mixed Multistroke Gestures (MMG) dataset [1] consists of a set of multistroke gesture samples in XML format. The gestures in the dataset were collected from 20 different users. Each user made 10 gestures repeatedly for each of the 16 gesture types shown in Fig. 2, using either their finger or a stylus on a Tablet PC, at three different speeds (i.e., slow, medium, and fast). A total of $20 \times 10 \times 16 \times 3 = 9600$ gesture XML files are included in this dataset. I only used the 3200 gestures that were made at the "medium" speed.

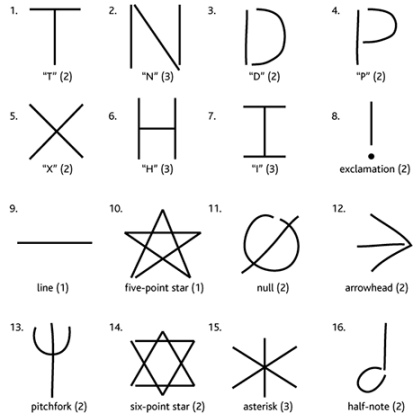

Fig. 2. A set of 16 gesture types. This gesture set was used to collect the Mixed Multistroke Gestures (MMG) dataset [1].

3.2 Zhuyin Multistroke Gestures (ZMG) dataset

In the Zhuyin phonetic system, there are 37 Zhuyin characters. I only extracted 16 symbols which are the ones that are easy to confuse, as shown in Fig. 3. These symbols include both unistroke gestures and multistroke gestures. Since Katarina and I are interested in investigating the effect of familiarity with Zhuyin symbols on the recognition accuracy, we decided to collect data from 6 users of different populations individually, and then combine the data into one dataset. More specifically, I recruited six Taiwanese people to provide gesture samples while Katarina recruited six users who have no prior experience with any of the Zhuyin symbols. The average age of the 12 users was 32.5 (SD=11.3). Among the six users I recruited, three were female and three had a technical degree in computer science. Among the six users Katarina recruited, three were female and three had a technical degree in computer science. All 12 users consented to participate by digitally signing an informed consent form. Users were asked to draw 10 gestures for each of the 16 gesture types (Fig. 3) at their own pace. Users practiced drawing each gesture type once before beginning the data collection process. The program displayed a prompt in blue bold text to tell the user which gesture to draw next. The type of the next gesture is randomly chosen to address the potential fatigue effect. Users were allowed to clear the canvas if they were not happy with the sample. A total of $12 \times 10 \times 16 = 1920$ gestures were collected. Details of the GUI for collecting gesture data are provided in section 4.2.

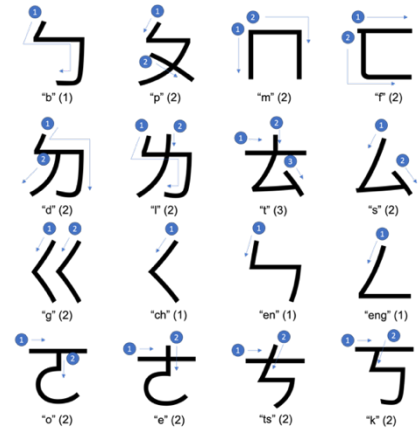

Fig. 3. A set of 16 Zhuyin symbols. Out of the 37 Zhuyin characters, these 16 symbols include some confusing pairs. For example, the ("b", "en") pair, the ("o", "e") pair, and the ("ts", "k") pair. The order and number of strokes are provided in this image for the user to follow.

4 Implementation Details

The implementation for this project consists of four different applications: 1) an application for running offline tests on the Mixed Multistroke Gestures (MMG) dataset [1], 2) an application for running offline tests on the ZMG dataset, 3) a graphical user interface (GUI) for live demo, and 4) a GUI for data collection.

4.1 Language and Code Structure

$P was implemented in Java for Project #2. The codebase consists of 12 Java files (.java) including "Main", "Constants", "User", "PointCloud", "Point", "Box", "Preprocessing", "PDollarRecognizer", "RecognitionResults", "NBestListItem", "Utils", and "GesturePanel". For Project #2, the *Gesture* class is replaced with the *PointCloud* class and the *OneDollarRecognizer* class is replaced with the *PDollarRecognizer* class. Fig. 4 shows the system architecture diagram.

Main. This class is the program's entry point. Changes made for Project #2 include replacing the *Gesture* class with the *PointCloud* class and replacing the *OneDollarRecognizer*

class with the *PDollarRecognizer* class. The rest of the implementation details have not been changed, including 1) reading in a gesture dataset, 2) parsing raw data into the *PointCloud* data structure, 3) preprocessing the raw *Points*, 4) looping through the *Users* and calling the recognize() method implemented in the *PDollarRecognizer* class for the recognition test on each candidate gesture, and then 5) generating testing logs and writing them to the output log file.

Constants. This class is implemented to store the constants (i.e., hyperparameters) for the application, including the number of users, number of gestures for each gesture type (e.g., 10), number of gesture types (e.g., 16), number of sample points for resampling (e.g., 32), and so on. Most of the numbers were decided based on the pseudocode provided in the $P paper [7]. More specifically, the number of sample points for resampling (i.e., 32) is different from the one (i.e., 64) used for the $1 recognizer.

User. This class is used to store user IDs and arrays of *PointClouds*. The gestures collected from each user are stored in a Java Map. The *key* of the Java Map is the type of the gesture, and the value is an array of *PointClouds*, with the length of the array being the number of gestures for each gesture type (e.g., 10).

PointCloud. This class stores the label and the index of a gesture, the raw data read from XML files in a list of *Points*, and the preprocessed data in another array of *Points*. The preprocessRawPoints() method declared in this class preprocesses the raw points by calling the following functions that are implemented in the *Preprocessing* class: resample(), scaleToSquare(), and translateToOrigin(). Note that the rotateToZero() preprocessing function is no longer needed because $P considers gestures as clouds of points.

Point. This class is used to store the x- and y-coordinates read from the XML files or the mouse events. Additionally, this class also stores the stroke ID, which is a new attribute added for multistroke gestures.

Box. This class is integrated into the scaleToSquare() preprocessing method in Project #2.

Preprocessing. This class contains the implementation for the three preprocessing methods called in the *PointCloud* class. 1) The resample() method resamples the array of *Points* from the raw user input. 2) The scaleToSquare() method scales the resampled *Points*. 3) The translateToOrigin() method translates the centroid of the scaled *Points* to the origin.

PDollarRecognizer. This class is implemented by modifying the *OneDollarRecognizer* class for the $1 recognizer. It stores a list of *PointClouds* as training templates and implements the recognize() method. The recognize() method takes a candidate *PointCloud* as input and calculates the distance between the candidate *PointCloud* and each training template using the greedyCloudMatch() method. Based on the resulting distances, the label of the template that results in the minimum distance is assigned to the candidate *PointCloud* as the recognized label. Lastly, the recognize() method outputs a *RecognitionResult* instance.

RecognitionResults and **NBestListItem**. These two classes were implemented to store the data needed for the output log file. No changes have been made for Project #2.

Utils. This class implements the utility methods, which were initially implemented following the pseudocode provided in the $1 paper [3] and then slightly modified based on the pseudocode provided in the $P paper [7] for Project #2. For example, when calculating the path length of a gesture, the length only includes the distance between points that belong to the same stroke (i.e., with the same stroke ID). Other utility functions were also implemented in this class, including the saveToXMLFile() method that saves a *PointCloud* to a file in XML format. Since an additional node is added in XML files for the stroke ID of a multistroke gesture, corresponding changes were made to the saveToXMLFile() method to create the additional node.

GesturePanel. JavaScript was chosen to implement the live demo part for Project #1. For Project #2, I implemented the live demo application in Java. The *GesturePanel* class is implemented to 1) show a canvas for users to draw gestures and display those gestures by adding mouse event listeners and overriding the paintComponent() method, 2) save the gestures to a file in XML format by calling the saveToXMLFile() method implemented in the *Utils* class, 3) display prompts to tell the user which gesture to draw next, 4) allow users to submit the gesture or clear the canvas by calling the getClearButton() and getSubmitButton() methods, and 5) provide the user with a *Recognize* button for live demo by calling the getRecognizeButton() method.

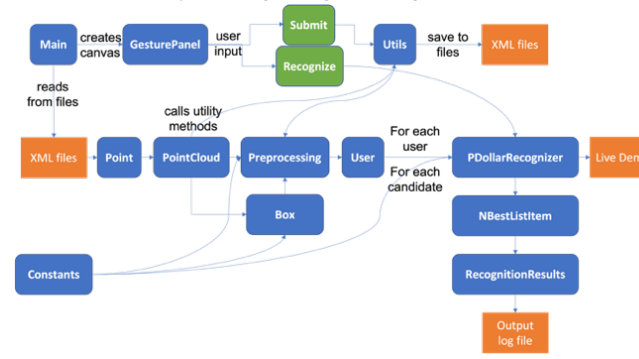

Fig. 4. The system architecture diagram. For Project #2, I integrated the live demo application into the Java implementation by enabling the recognize button on the canvas. The *Gesture* class was replaced with the *PointCloud* class and the *OneDollarRecognizer* class was replaced with the *PDollarRecognizer* class.

4.2 Runnable Components

I implemented two applications to generate logs for offline gesture recognition tests because of the different file structures between the MMG dataset and the ZMG dataset. However, the required formats of the XML files are compatible, and the interfaces are the same between the two applications. The window that displays the recognition progress is the same as the one implemented for Project #1, as shown in Fig. 5.

Fig. 5. The window for displaying recognition progress. This figure shows the last phase of the recognition process.

For the demonstration of online recognition with the $P recognizer, a GUI containing a canvas was implemented in Java, which is different from the one implemented in JavaScript for Project #1. A folder with loaded templates is required for the $P recognizer to perform the online recognition. As shown in Fig. 6, the user can draw gestures on the canvas with orange borders. Clicking the *Clear* button clears the canvas and allows the user to draw another gesture. Clicking the *Recognize* button displays the recognition result in blue bold text.

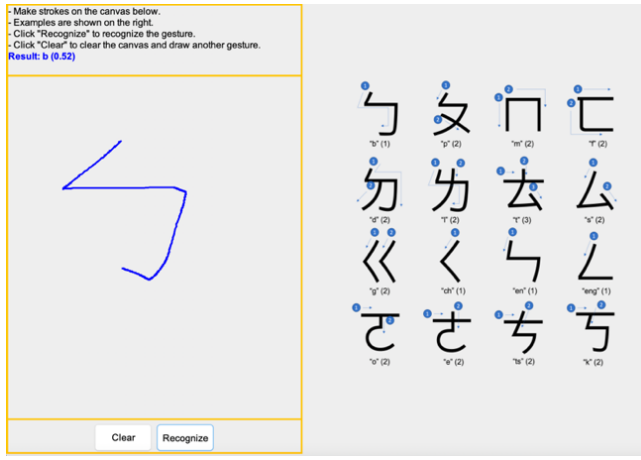

Fig. 6. The GUI for the live demo. The *Clear* button clears the canvas. The *Recognize* button displays the recognition result in blue text.

I implemented a GUI containing a canvas in Java for data collection. For Project #2, the only change I made is replacing the example image with the new dataset. As shown in Fig. 7, this GUI shares the same interface as the live demo interface. Instead of displaying the recognition results, the interface shows the user what to draw next and the count of gestures that have already been submitted. The *Clear* button clears the canvas and shows the required gesture again. The *Submit* button will save the drawn gesture, create an XML file, clear the canvas, and then show the next gesture.

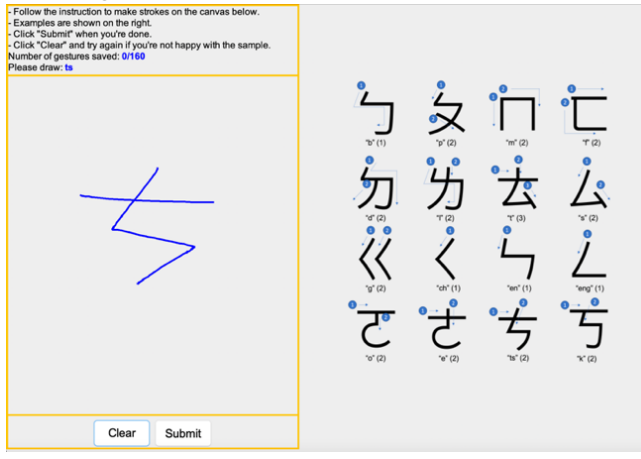

Fig. 7. The GUI for data collection. It contains a canvas and instructions for users to make their gestures. The *Clear* button clears the canvas. The *Submit* button saves the drawn gesture to an XML file.

4.3 Implementation Challenges and Solutions

To implement the live demo application, I had to understand how to implement interactive components in Java. Compared to implementing such components in JavaScript, the implementation process was less straightforward and involved more trial-and-error processes. Eventually, I was able to integrate the live demo functionality into the existing GUI (Fig. 7) that I have already implemented for data collection.

5 Offline Recognition Results

I ran user-dependent tests on the two datasets: the MMG dataset and the ZMG dataset. For each user, for each activity type, E samples were randomly selected as the training data and one additional sample was selected for testing. This process was repeated 100 times for each value of E (e.g., 1 to 9), and the average of the results was calculated for each user. The results from a total of 20 (users) × 9 (values for E) × 16 (gesture types) × 100 (random iterations) = 288,000 recognition calls for the MMG dataset were averaged into an overall recognition accuracy of 99.27%. The results from a total of 172,800 recognition calls were averaged into an overall recognition accuracy of 87.53% for the ZMG dataset. Results from running the $P recognizer on the MMG dataset were similar to the results reported in the $P paper [7]. Results also showed that it is challenging for the $P recognizer to achieve an accuracy that is higher than 90% on the ZMG dataset when the training samples are less than five for each gesture type.

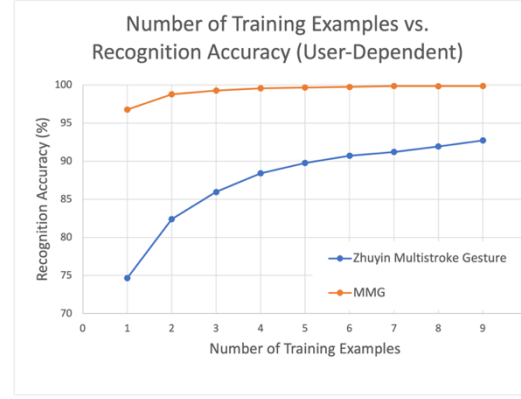

Fig. 8. Recognition results on the MMG dataset and the ZMG dataset in the user-dependent test. The accuracy of the $P recognizer increased as the number of training examples increased for the tests on both datasets.

I further compared the performance of $P on the data from 6 users who are familiar with Zhuyin and 6 users who are not. As shown in Fig. 9, there was no significant effect of the familiarity with Zhuyin symbols on the recognition accuracy. Surprisingly, the average recognition accuracy (88.49%) of the gestures drawn by participants who are not familiar with Zhuyin was slightly higher than the average recognition accuracy (86.57%) of the gestures drawn by participants who are familiar with Zhuyin. A possible reason is that users who are not familiar with the symbols might have been more careful when drawing the gestures.

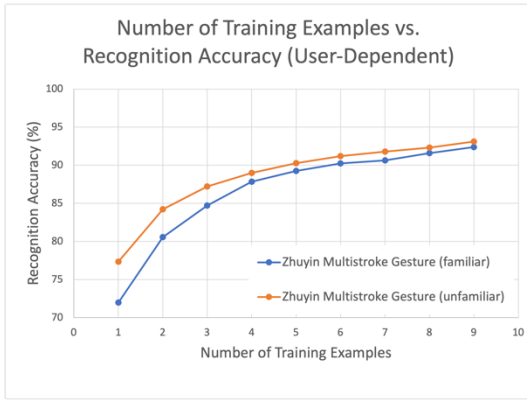

Fig. 9. Recognition results on different parts of the ZMG dataset. The performance of $P on the data from 6 users who are not familiar with Zhuyin was slightly better than its performance on the data from 6 users who are familiar with Zhuyin.

6 Understanding Data

To understand gesture recognition errors, I used the GHoST heatmap toolkit [2] to analyze the ZMG dataset and extracted user articulation insights. As shown in Fig. 10, for all 12 users, the "b" gesture and the "en" gesture were drawn in a very similar way. The important "hook" at the end of the "b" gesture is not obvious enough to help the recognizer distinguish these two gestures. Comparing the heatmap generated using gestures from 6 users who are familiar with Zhuyin and the heatmap generated using gestures from 6 users who are not familiar with Zhuyin, we can see that the latter shows higher variations across different users, as shown in Fig. 11. It was expected that there will be higher variations among gestures that were drawn by users who have no prior experience with Zhuyin. However, this variation did not cause the recognition accuracy to be lower, based on the results reported in section 5. Thus, the main reason for a lower accuracy of 87.53% may still be the inherent similarities between Zhuyin symbols in certain confusing pairs.

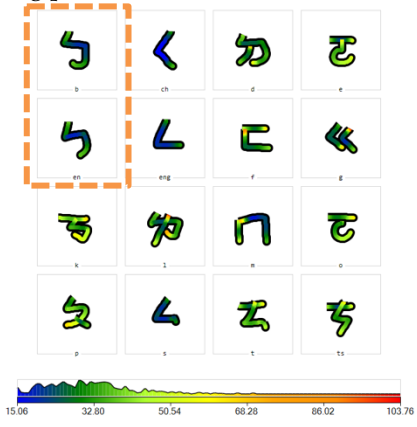

Fig. 10. The gesture heatmap generated using the entire ZMG dataset. The ("b", "en") pair is annotated to show a possible confusing pair of gestures that might cause recognition errors.

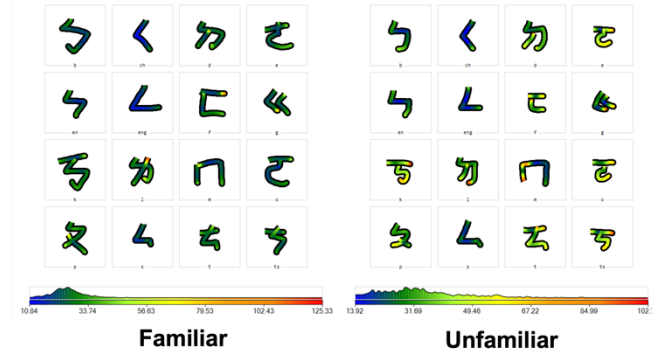

Fig. 11. The gesture heatmap generated using gestures from two different groups of users in the ZMG dataset. The heatmap of users who are familiar with Zhuyin (left) shows lower variation than the heatmap of users who are not familiar with Zhuyin (right).

7 Outcome and Conclusion

The outcome from Project #2 includes: 1) similar Zhuyin symbols can lower the recognition accuracy and 2) familiarity with the Zhuyin symbols has no significant effect on the recognition accuracy. The former is aligned with the prediction in my Project #2 proposal while the latter is not. The fact that the stroke order was provided (Fig. 7) might be a potential reason that users who are not familiar with the symbols did not perform worse.

After implementing both $1 and $P for the two projects, I gained deeper insights into how these two members of the $-family were designed and their limitations. It was also very interesting to collect data remotely from users by deploying the application instead of conducting the session in person. Lastly, the GHoST toolkit [2] was very useful for understanding users' gesture articulation patterns and recognition errors.

References

[1] L. Anthony and J. O. Wobbrock, "$N-Protractor: A Fast and Accurate Multistroke Recognizer," in *Proceedings of Graphics Interface 2012*, 2012, pp. 117–120.

[2] R.-D. Vatavu, L. Anthony, and J. O. Wobbrock, "Gesture Heatmaps: Understanding Gesture Performance with Colorful Visualizations," in *Proceedings of the 16th International Conference on Multimodal Interaction*, 2014, pp. 172–179, doi: 10.1145/2663204.2663256.

[3] 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, pp. 159–168, doi: 10.1145/1294211.1294238.

[4] C. S. Myers and L. R. Rabiner, "A comparative study of several dynamic time-warping algorithms for connected-word recognition," *Bell Syst. Tech. J.*, vol. 60, no. 7, pp. 1389–1409, 1981.

[5] D. Rubine, "Specifying Gestures by Example," in *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, 1991, pp. 329–337, doi: 10.1145/122718.122753.

[6] L. Anthony and J. O. Wobbrock, "A Lightweight Multistroke Recognizer for User Interface Prototypes," in *Proceedings of Graphics Interface 2010*, 2010, pp. 245–252.

[7] R.-D. Vatavu, L. Anthony, and J. O. Wobbrock, "Gestures as Point Clouds: A $P Recognizer for User Interface Prototypes," in *Proceedings of the 14th ACM International Conference on Multimodal Interaction*, 2012, pp. 273–280, doi: 10.1145/2388676.2388732.