POLYREC GESTURE DESIGN TOOL: A tool for fast prototyping of gesture-based mobile applications

Roberto Bufano | Gennaro Costagliola | Mattia De Rosa | Vittorio Fuccella

Department of Informatics, University of Salerno, Fisciano, Italy

Correspondence
Mattia De Rosa, Department of Informatics, University of Salerno, Via Giovanni Paolo II, 84084 Fisciano (SA), Italy.
Email: matderosa@unisa.it

Abstract
In this article, we present POLYREC GESTURE DESIGN TOOL (POLYREC GDT), a tool that allows the definition of gesture sets for mobile applications. It supports the entire process of designing gesture sets, from collecting gestures to analyzing and exporting the set created for use in a mobile application. It is equipped with the POLYREC gesture recognizer, a unistroke recognizer based on template matching with good accuracy and performances while still allowing support for other recognizers. The main features of POLYREC GDT include the ability to collect gestures directly on mobile devices, to detect similarities between gestures to help the designer to detect ambiguities in the design phase, and the ability to automatically select the most representative gestures for each gesture class by using a clustering technique to increase recognition speed without affecting accuracy. Experimental results show that this latter feature allows an increase in accuracy compared with when the same operation is performed manually by developers or randomly by software. Finally, a user study was carried out to evaluate POLYREC GDT, in which participants were asked to use it to add gesture functionality to a mobile application. The results showed that with the support of POLYREC GDT this operation took only a few minutes for participants with mobile development experience, whereas it would have taken much longer without the support of POLYREC GDT.

KEYWORDS
gesture recognition, gesture-based interaction, PolyRec, unistroke gestures

1 | INTRODUCTION

In recent years there has been a huge spread of electronic devices equipped with touch inputs, such as smartphones, tablets, smartwatches, and so on. On these devices, most applications are not limited to using classic touch-based interaction methods but also support some kind of gesture-based interaction (e.g., gestures to execute commands, text entry/editing). The usability of a gesture-based UI depends essentially on three factors: the hardware used for the gesture input, the gesture recognition algorithm, and the designed gesture set.
The characteristics for which the different hardware differ are essentially accuracy, pressure detection, and the number of simultaneously detectable points (multitouch). These characteristics define the types of gestures that the device is able to accept and therefore must be taken into account during the design of the gesture set.

Mobile operating systems such as iOS and Android include features for recognizing the most common gestures, and the HCI community has produced different solutions to the problem of recognizing gestures belonging to a specific gesture sets, both multistroke and unistroke. In particular, the latter is often preferred for their usability and for the ease with which they can be interpreted by the computer.

Designing a good set of gestures can be complicated, especially when the number of gestures increases. A good set of gestures must balance two often conflicting features: be composed of gestures easily classifiable by an algorithm but at the same time easy to perform and remember for human beings. Moreover, for a software designer, it can be difficult to evaluate the characteristics of a gesture set with respect to these features.

A good tool for the gesture design, in addition to allowing to collect and organize the gesture set, must, therefore, provide the developer feedback about these features. Designing good gesture sets can be essential to take full advantage of gesture-based interfaces.

The goal of this work is therefore to present Polygon Gesture Design Tool* (or Polygon GDT), a tool for the design and definition of gesture sets for mobile applications (touch devices), with particular attention to the definition of gesture sets that allow a high recognition rate.

Polygon GDT is equipped with the Polygon 8 unistroke gesture recognizer based on template matching and supports the whole process of gesture set design, from the collection of gestures, to the analysis, to the export of the created set for use in a mobile application together with the Polygon recognizer. Moreover, the tool is fairly generic and thus allows supporting additional gesture recognizers.

To simplify the analysis of the gesture set, the tool provides functionalities for the detection of similarity between gestures, in order to help the designer in identifying problems that could generate ambiguity in the recognition phase, so that he can redesign them to limit conflicts. In addition, the tool allows the developer to automatically select the most representative templates of a gesture class, so as to reduce the computational cost while reducing the impact on recognition accuracy. These are the key distinguishing features of Polygon GDT with respect to other gesture management tools.

The latter features and the ability to export easy-to-use gesture recognition source code are the main design goals of the tool.

Another issue in the area of gesture recognition is how to find the right compromise between execution speed and accuracy of recognition. To this end, Polygon GDT uses a clustering technique to automatically select the most representative templates for each gesture class. The usefulness of this feature was evaluated both on datasets of standard gestures and in the real case of a mobile application in which the developers, not having Polygon GDT, had manually chosen the gestures for the recognizer. The results show that the algorithm selection allows to increase the accuracy of the recognition, especially when only one sample per class is available.

Finally, to evaluate Polygon GDT, an experiment has been performed in which participants were asked to use it to support them during the development of a mobile application. Results show that adding gesture functionalities to an existing application took only a few minutes for a participant with mobile development experience, which would have taken a much longer time without Polygon GDT.

This article is organized as follows: Section 2 describes some works related to ours; Section 3 describes the Polygon gesture recognizer and analyzes its accuracy; Section 4 describes the Polygon GDT user interface and features; Section 5 describes an empirical case study on the automatic templates selection functionality; Section 6 describes the Polygon GDT experimental evaluation. A discussion on the results of this study and some comments on future work concludes the article.

2 RELATED WORK

In this section, we discuss previous researches related to our work. It is divided into two sections: the first one describes works on gesture design and supporting tools and techniques, while the second one describes algorithms for gesture recognition on touch interfaces.

https://github.com/cluelab/polygon-gdt.
2.1 Gesture design

Over the years, some tools have been proposed to help in the development of gesture-based applications. Some of them are complete environments that support the developer both in creating the gesture set and in developing the applications that use (recognize) them.

Quill\textsuperscript{9,10} is a Java software whose goal is to help designers of pen-based user interfaces to design sets of gestures, with the aim, for example, to allow the execution of application-specific operations through the input of gestures. The tool precedes the diffusion of finger-based touch devices and is therefore dedicated to pen-based devices. The features supported by Quill include the ability to add gestures, recognize gestures, edit gesture sets, organize gestures, and try out the gesture recognition. The tool also gives designers advice on gestures that may be difficult to remember or recognize, and provides suggestions on how to improve them.

RATA\textsuperscript{11} is a tool to support developers in the generation of gesture recognizers. Based on the Rata.SSR\textsuperscript{12} gesture recognizer, it offers a wizard that without requiring coding experience guides the developer through the processes of gesture set definition, sample collection, labeling (with support for automatic labeling), and recognizer generation. Another tool is Gesture Script,\textsuperscript{13} capable of creating unistroke gesture recognizers. It allows to enhance classic example-based learning by using rendering scripts and interactively trained parts. Rendering scripts with their structural information allow the generation of a more accurate recognizer that can also automatically extract gesture attributes that can be useful for the application that uses the recognizer.

iGesture\textsuperscript{14} is a framework for gesture recognition in Java, focused on cross-application extensibility and reusability, with the aim of providing an integrated solution that includes a recognition framework and a tool for creating custom gesture sets. iGesture supports both touchscreen and digital pen-and-paper-based interfaces and allows to use all available recognition algorithms with a single API. Another framework is Proton,\textsuperscript{15} which focuses on multitouch gestures. Proton allows the developer to define a gesture as a regular expression on a touch event stream, it is capable of performing static analysis in order to detect conflicts, and it can automatically generate a recognizer for the whole set of gestures.

Gesture coder\textsuperscript{16} is an Eclipse plugin that allows developers to automatically generate code that recognizes multitouch gestures. It allows developers to create sets of multitouch gestures by providing examples using a multitouch device (e.g., a tablet). The developer can use the tool to test the gesture recognition, adjust the set by adding further examples, and when satisfied, generate the code to be included in its application. Experimental results show that the recognition accuracy is sufficient for rapid prototyping and that it can be improved with more, and higher quality, training data. An experiment with professional programmers showed that the tool significantly reduces the difficulty of programming multitouch gestures. The same authors later proposed GestureStudio,\textsuperscript{17} which adds a declaration-based approach that allows developers to explicitly inform the system of high-level behaviors by composing different gestures on a timeline and adding callback actions. An experiment showed that the performance of their touch motion recognizer exceeded that of their previous work. TouchToolkit\textsuperscript{18} is instead a tool that focuses on the automatic testing of multitouch gestures. It allows developers to record gestures and incorporate them into unit tests, allowing developers to avoid the repetitive task of testing their application manually on the target device.

Dealing with gesture recognition can also be a barrier to rapid prototyping. GestureWiz\textsuperscript{19} is a prototyping system for applications using different types of gestures, for example, touch or midair. The system provides an integrated solution for gesture definitions, conflict checking, and gesture recognition performed by humans (including crowd workers) in a Wizard of Oz manner.

Some tools also have functionality to support gesture elicitation. GestMan\textsuperscript{20} is a web-based tool for the management and acquisition of 2D/3D gestures. In addition to allowing users to collect gestures to create their own gesture sets, it supports gesture sharing and the management of collaborative experiments for gesture elicitation. It also provides processing (e.g., rotation, translation, scaling), analysis (e.g., viewing ruby features), and recognition (with $1,\textsuperscript{21} P,\textsuperscript{22}$ or $!FTL,\textsuperscript{23}$) functions. The gesture sets can be exported in JSON format. Some of the same authors have also created GELICIT,\textsuperscript{24} a cloud computing platform that supports gesture elicitation studies distributed across time and space, so that researchers are not limited to a single, physical, synchronous setup, and that supports them at the different stages of such studies.

Instead of targeting developers, Gestify\textsuperscript{1} is an Android app that allows the user to define custom gestures and the functions to execute when they are performed. Created in order to investigate which functions users want to access on smartphones using touch gestures as shortcuts, it uses the $1,\textsuperscript{21}$ recognizer and allows the user to perform the gestures either in the lock screen, by invoking the feature from the notification bar, or through a live wallpaper.

Other tools allow operations on already existing gesture sets and are designed to help analyze, revise, and reorganize the gesture set. The Gesture RElative Accuracy Toolkit (GREAT)\textsuperscript{25} is a tool aimed at developers that need to tweak gesture
sets and recognizers, both for single and multistroke gestures. Its main feature is that it allows the developer to analyze 12 new relative accuracy measures for stroke gesture articulation that characterize geometric, kinematic, and articulation accuracy of gestures. The measures have been evaluated on public gesture datasets in order to better understand users’ stroke gesture articulation behavior and to obtain useful information for developers on this topic. The GESture Clustering toolKit (GECKo)\textsuperscript{26} is a tool whose aim is to facilitate the study of how users articulate stroke gestures (both unistroke and multistroke) and help to better understand how users make gestures. It allows to visualize and playback gestures, and, in order to analyze user consistency in gesture production, to cluster them based on stroke number, order, and direction. It allows to audit the clustering results and to see within- and between-subject agreement rates after clustering. The methodology has been used on four gesture datasets and implications from results can be used to help designers create better gesture sets.

So far we reported tools that support touchscreen operations; however, there are also systems that deal with different types of gestures, such as midair or hand-based gestures. We report them here only for completeness since these types of gestures are out of the scope of our current proposal. Among these systems we can mention MAGIC (Multiple Action Gesture Interface Creation),\textsuperscript{27} a tool that allows to design motion gestures without specific knowledge of pattern matching, and also test them in order to ensure that they are not triggered unintentionally by the user (false positives); GRDT (Gesture Recognition Design Toolkit),\textsuperscript{28} a set of tools to support gesture creations; Mogeste,\textsuperscript{29} a tool that allows designers to define, train, and test new motion gestures just captured through the inertial sensors in a commodity device (e.g., smartphone); EventHurdle,\textsuperscript{30} a visual design tool that support designers in exploratory prototyping, provides functionality to integrate the defined gestures into a prototype for their automatic recognition, and supports handled gestures through physical sensors, remote gestures through a camera, and also touchscreen gestures; ProGesture,\textsuperscript{31} that supports rapid prototyping of full body gestures by combining three design matters: gesture, presentation, and dialog; KIND-DAMA,\textsuperscript{32} a modular middleware for developing interactive applications based on gestures on Kinect-like devices; KinectAnalysis,\textsuperscript{33} a system for elicitation studies on the Microsoft Kinect, with support for recording, analysis and sharing; GestureAnalyzer,\textsuperscript{34} a tool that allows gesture analysis by applying clustering and visualization techniques to the gesture data captured by motion tracking; CUBOD,\textsuperscript{35} which allows users to design and customize their own gestures, rather than relying on those defined by designers, and offers feedback to guide users through the design process in order to avoid gestures that are difficult to distinguish from others, difficult to execute with consistency, or too similar to unintended movements; and HotSpotizer,\textsuperscript{36} another system that allows users to define their own custom gestures, in this case to map them into keyboard commands, in order to use them in arbitrary applications.

In addition to tools, techniques, and methods to aid in the design of gesture sets have also been presented in the literature. Henrique et al.\textsuperscript{37} proposed a methodology, with focus on pen-gesture interfaces, to automate the design of a gesture vocabulary. Using a clustering algorithm, the gesture space can be explored for a set of gestures easily recognizable by the recognizer, and then allowing the gesture vocabulary designer to select the most appropriate gestures from the self-organized gesture lists. When gestures are designed for a specific system or platform, it is often difficult, if not impossible, to use the same gestures on systems with different characteristics. Lao et al.\textsuperscript{38} attempted to make a complete understanding of all possible touch types for touch-sensitive devices, and then create a design model for touch interaction. The model consists of three levels: action level, motivation level and computing level. This allows reusing the design work performed in one level, for example, when designing an interactive interface it is possible to design the touch at the action level once and reuse it for other applications and platforms. In Reference 39 the authors propose a technique for gesture analysis called gesture heatmaps. This technique allows showing, through a colored visualization, how the values of some descriptors (e.g., speed) change along the gesture path. The authors also introduce chromatic confusion matrices, which extend the expressiveness of classical confusion matrices in the analysis of gesture classification performance. These visualizations can be helpful for people doing gesture analysis, for example, to understand in a simpler way the causes of recognition errors.

Table 1 summarizes the main attributes of the tools and techniques for gesture design that we have discussed in this section. It is worth noting that most of the work focuses on using a single gesture recognizer. PolyRec GDT supports multiple recognizers and is easily extensible in that respect. Various tools, on the other hand, offer functionality to help the user design the gesture set, but none of them offer a technique that is exactly equal to the one of PolyRec GDT. The one that comes closest is the one proposed in Reference 37. Although various systems offer functionality to simply generate a gesture recognizer, none offer a dedicated feature for Android developers like that of PolyRec GDT, which allows to have a working recognizer in an Android touch application by just pasting the source code generated by the tool.
TABLE 1  Gesture design software and techniques discussed in the related work section

| Gesture design software (touch/pen gestures) | 1,9,11,13-20,24,25,30,38 |
| Gesture design software (nontouch gestures) | 19,20,24,27-32,34-36 |
| Gesture design technique | 37,38 |
| Gesture analysis technique | 38,39 |
| Gesture analysis software | 25,26,34 |
| End-user gesture software | 1,35,36 |
| Gesture elicitation software | 20,24,33 |
| Multiple recognizers support | 20,24 |
| Unit testing tool | 18 |

2.2  Gesture recognition methods

Most of the applications for the latest mobile devices support a gesture-based interaction. In the last years, the HCI scientific has proposed many gesture recognizers (in this context we refer to a “gesture” as the path traced on a touch-sensitive surface by a finger or a stylus).

Among the first works, we can remember those based on dynamic time warping (DTW)\textsuperscript{40,41} and Rubine,\textsuperscript{42} the first widely used gesture recognizer, a feature-based linear classifier. The various techniques used for gesture recognition include those based on Hidden Markov Models,\textsuperscript{43} neural networks,\textsuperscript{44-46} feature-based statistical classifiers,\textsuperscript{47} or a combination of them.\textsuperscript{12}

More recently, among the template-based matching algorithms, the “$-$family” of gesture recognition algorithm has emerged. It started with $1,\textsuperscript{21}$ an algorithm for unistroke gesture recognition, with a focus on simplicity and fast implementation, which compares points in the sampled strokes for each gesture and is scale and (optionally) rotation invariant. Although more accurate than Rubine, $1$ has some problems, for example, a problem with uniform sampling has been reported: “misalignment” in one-to-one correspondence and “unwanted flexibility” in elastic matching.

$1$ has since been extended several times, for example, for multistroke gesture recognition ($SN\textsuperscript{48}$), for increased speed and accuracy as in Protractor\textsuperscript{49} and $SN$-Protractor,\textsuperscript{50} for articulation-invariant recognition ($SP\textsuperscript{22}$ and the faster and more accurate $SQ\textsuperscript{51}$), and for people with low vision ($SP+\textsuperscript{52}$).

Methods of other authors inspired by the same philosophy include: Quick$,\textsuperscript{53}$ which uses hierarchical clustering and branch and bound to increase efficiency and avoid making comparisons with all training templates; $1\textsuperscript{E}\textsuperscript{54}$ which improves speed compared with $1$; Penny Pincher\textsuperscript{55} which uses only addition and multiplication for recognition and improves the recognition rate and speed of previous $-$family recognizers; Jackknife,\textsuperscript{56} a set of techniques for pen and touch, Wii Remote, Kinect, Leap Motion, and sound-sensed gesture recognition; $3,\textsuperscript{57}$ Protractor3D\textsuperscript{57} and 3 cent\textsuperscript{58} which allow 3D gesture recognition. $1\textsuperscript{59}$ and $1\textsuperscript{F}\textsuperscript{60}$ are filters to improve the performance of recognizers in specific conditions.

Other recognizers based on geometric\textsuperscript{3,23} functions have also been proposed.

3  THE POLYREC RECOGNIZER

POLYRec\textsuperscript{8} is scale and (optionally) rotation invariant and uses a nearest neighbor approach to recognize an unknown gesture, that is, the distance between the unknown gesture and each template is calculated, then the gesture is classified into the class of the template at shortest distance. The main feature of POLYRec is to represent a gesture through a polyline identifying its main movements. An advantage of this representation is to only use a small sequence of dominant points to calculate the distance from another gesture, improving recognition efficiency.

Summarizing, the distance between two gestures is calculated through the following three steps:

1. Preprocessing: the two gestures are approximated to a polyline, in order to extract their dominant points. A polyline can be composed of one or more segments;
2. **Alignment**: the two polylines are aligned, that is, their segments are possibly split until the polylines are composed of an equal number of segments;

3. **Calculation**: the distance is found by summing the contribution of the distances between each pair of corresponding segments.

To find the distance between two gestures, the segments of each polyline are treated as vectors, characterized by magnitude and direction, and compared in pairs with those of the other polyline. The distance is obtained by summing the contribution of each pair of segments.

### 3.1 Step 1: Preprocessing

An input gesture contains a variable number of points sampled from the device. Our preprocessing uses the Douglas-Peucker reducer\(^6^1\) (DP, in the following) to extract the dominant points, where the gesture shows a greater curvature. The result is a broken line, which we call a polyline.

DP recursively works as follows: initially the polyline is formed by the segment connecting the endpoints of the gesture; then the furthest point of the gesture is added to the polyline, and DP is recursively executed on the two parts of the gesture. The point is added only if it is at a distance greater than a tolerance threshold.

In order to obtain a smoother polyline, with segments representing the main movements intended by the drawer, an iterative LineFusion procedure (see Reference \(6^2\)) is executed. At each iteration, the point between the segments having the smallest direction change is removed. The process ends when the smallest direction change exceeds a minimum threshold \(\theta\).

Figure 1 shows the result of preprocessing on an arrow sample gesture. As shown in the figure, after the DP step, LineFusion refines the polyline in order to better identify the main movements of the gesture.

The complexity of preprocessing is \(O(N^2)\), with \(N\) being the number of points in the input gesture. It is worth noting that preprocessing is executed only once on each template.

### 3.2 Step 2: Alignment

After preprocessing, a polyline is composed of a variable number of points. In step 2, we equalize the number of points of the polylines to compare. To do this, we firstly establish a mapping between the existing points. If a point is unmapped, a corresponding point is added to the other polyline. The mapping is based on the similarity between the points.

To perform the mapping we use a modified version of Needleman-Wunsch\(^6^3\) (NW, in the sequel), an algorithm used for the alignment of strings. NW calculates an optimal alignment between two strings based on a scoring scheme. Here we use the same idea to map points. In particular, the effect is to match two points of the polylines only when they are quite similar. Similarity is determined using two features:

- **pos**\((p)\): the position of the point within the gesture, calculated as the ratio between the length up to \(p\) and the total length;
- **angle**\((p)\): the difference of direction between the two segments of the polyline that meet at \(p\).

Given two polylines \(P\) with points \(p_1, \ldots, p_n\) and \(Q\) with points \(q_1, \ldots, q_m\), the alignment can result in:

![FIGURE 1 Points (highlighted in red) sampled on an arrow sample gesture](image-url)
Before alignment

After alignment

FIGURE 2  Alignment of two polylines. After alignment, two new vertices (marked through a circle) are sampled from P and one from Q.

- **Match** \((p_i, q_j)\): points \(p_i\) from \(P\) and \(q_j\) from \(Q\) are matched, that is, the pair \((p_i, q_j)\) is added to the alignment;
- **Insert** \((P, q_j)\): a new point mapped to \(q_j\) is added in \(P\), that is, the pair \((- q_j)\) is added to the alignment;
- **Insert** \((Q, p_i)\): a new point mapped to \(p_i\) is added in \(Q\), that is, the pair \((p_i, -)\) is added to the alignment.

The insertion of a point in a polyline occurs between the closest points having a correspondence in the other polyline. For instance, from the alignment \(\ldots (p_{m'}, q_{m'})(- q_j)(p_{m''}, q_{m''}) \ldots\), we determine that a new point \(p\) mapped to \(q_j\) must be added in \(P\) between \(p_{m'}\) and \(p_{m''}\). The ideal position of the new sampled point \(p\) in the subgesture delimited by \(p_{m'}\) and \(p_{m''}\) is established proportionally to the position of \(q_j\) in the subgesture delimited by \(q_{m'}\) and \(q_{m''}\).

An example of polyline alignment is shown in Figure 2. The figure shows the result of the application of alignment on two gestures reduced to two polylines \(P\) and \(Q\). Before alignment (see Figure 2A), \(P\) and \(Q\) contain 8 and 9 points, respectively. The alignment returned by NW is the following: \((p_1, q_1)(p_2, -)(p_3, q_2)(- q_3)(p_4, q_4)(p_5, q_5)(p_6, q_6)(- q_7)(p_7, q_8)(p_8, q_9)\). As a result, two vertices corresponding to \(q_3\) and \(q_7\) are inserted in \(P\) and one corresponding to \(p_2\) in \(Q\) (see Figure 2B), thus obtaining two polylines composed of 10 points each.

3.3  |  Step 3: Distance

In this step, a polyline is considered as a sequence of vectors. Each segment of the polyline is transformed into a vector having magnitude equal to its relative length and direction equal to its slope. The distance between two polylines is calculated by summing the magnitude of the vectors obtained from the difference of each pair of vectors in the corresponding sequences of vectors. A penalty increasing the distance proportional to the number of mismatches is also used.

Our recognizer can be used both in a rotation sensitive and in a rotation invariant context. In the latter case, the indicative angle\(^{21}\) of the gesture is determined. In both cases, we adopt golden section search (GSS) to determine the distance at the best angle.

The three steps require the setting of parameters. We performed a tuning on a dataset composed of 400 curves.\(^{64}\)

3.4  |  Recognition accuracy

We evaluated our recognizer in a rotation-invariant context by testing it on two different unistroke datasets. In addition to Protractor\(^ {49}\) as in Reference 8, we compared its performance also with \(\$P, ^{22}\) \(\$P+, ^{52}\) \(\$Q, ^{51}\) and Jackknife.\(^{56}\) The main features of the datasets are summarized in Table 2. The first dataset is the one introduced to evaluate the \$1 recognizer.\(^{21}\) The second dataset is the one presented in Reference 65.
TABLE 2 Features of the two datasets used in our evaluation

| Dataset | Classes | Articulation speeds | Symbols | Writers | Source |
|---------|---------|---------------------|---------|---------|--------|
| $1$     | 16      | 3                   | 4800    | 10      | 21     |
| ILGDB   | 21      | –                   | 1923    | 11      | 65     |

3.4.1 Methods

We performed both a writer-dependent test and a writer-independent test similar to the ones used in Reference 21 and 22. For the writer-dependent test, the recognition rates were computed individually for each participant and each articulation speed. For each gesture class and speed, a random sample was selected for testing, and $T$ different samples were randomly selected for training. For each value of $T$, the recognition was performed 100 times for each method, and results were averaged into a recognition rate per participant.

For the $1$ dataset the number of training samples per gesture class $T$ varied from 1 to 9, this resulted in a total of 10 participants $\times$ 3 speeds $\times$ 9 values for $T \times 100$ repetitions for each $T \times 16$ gesture classes $\times$ 6 recognizers = 2,592,000 recognition tests.

For the ILGDB dataset the number of training samples per gesture class $T$ varied from 1 to 4 (since there are writer/class combination with only five samples), this resulted in a total of 11 participants $\times$ 1 speed $\times$ 4 values for $T \times 100$ repetitions $\times$ 21 gesture classes $\times$ 6 recognizers = 554,400 recognition tests.

In the writer-independent test, for each articulation speed, a random participant was selected for testing and $P$ different participants for training. For each class, a random gesture was selected for the testing participant, while $T$ samples were randomly selected for each gesture class from each training participant. For each value of $P$ and $T$ the recognition was performed 100 times for each method and results were averaged into a recognition rate for each $P$ value.

For the $1$ dataset the value of $P$ varied from 1 to 9, and the value of $T$ varied from 1 to 9, this resulted in a total of 9 values of $P \times 3$ speeds $\times$ 9 values for $T \times 100$ repetitions $\times$ 16 gesture classes $\times$ 6 recognizers = 2,332,800 recognition tests.

For the ILGDB dataset the value of $P$ varied from 1 to 5, and the value of $T$ varied from 1 to 9, this resulted in a total of 5 values of $P \times 1$ speed $\times$ 9 values for $T \times 100$ repetitions $\times$ 21 gesture classes $\times$ 6 recognizers = 567,000 recognition tests.

Furthermore, as a comparison, we have also measured the accuracy of POLYREC when the samples, rather than randomly, were selected using the k-medoids clustering algorithm (see Section 4.3.3 for more details on the corresponding functionality of POLYREC-GDT).

For all recognizers, we used the default parameters defined by the authors in their published implementations. The experiment was conducted on a HPE server equipped with an Intel Xeon Gold 5118 CPU at 2.30-GHz running Ubuntu 20.04 operating system and the Java Run-Time Environment.

3.4.2 Results

The results of the recognition tests are shown in Figures 3 and 4, for the two datasets, $1$ and ILGDB, respectively.

In the case of the $1$ dataset, POLYREC is the method with the highest accuracy in the writer-independent test, while it is the second in the writer-dependent case, since $P+$ has higher performance, although at the cost of more than doubled recognition times. In the case of the ILGDB dataset POLYREC is also the method with the highest accuracy in the writer-independent test, while in writer-dependent case Jackknife is the method with the best accuracy when only one training sample is used, whereas POLYREC is the most accurate when two or more training samples are used.

As regards the recognition times, it can be noted that Protractor, although the least accurate, is clearly the fastest method, while $P$ and $P+$ are the slowest methods. POLYREC and Jackknife have very similar performance and are most often the fastest methods (excluding Protractor), while $Q$ is faster when there are many training samples. This is because the optimization of $Q$ has a high initial cost which, however, allows an almost linear trend with increasing training samples.

†https://github.com/cluelab/dollar-recognizers-java, https://github.com/ISUE/Jackknife.
Furthermore, the results show that when, instead of using a random selection, the training samples are selected with the k-medoids algorithm (see Section 4.3.3), the accuracy is much higher for both datasets in the case of a single training sample. When the number of training samples increases this difference decreases until it approaches zero.

4 | POLYREC GESTURE DESIGN TOOL

POLYREC GESTURE DESIGN TOOL (or POLYRec GDT in short) is a software designed to help mobile application developers in the creation of the gesture sets used in their applications. The goal is to allow designers, even those who have no experience with the POLYRec recognition technology (or other recognizers), to create good sets of gestures.

The system allows the developer to define the gesture classes that he intends to use and to add the corresponding gestures, both directly on the PC or by drawing them on a smartphone. In the following, we will call template a gesture that belongs to a class.

In addition to this basic functionality, POLYRec GDT provides functionality to help to understand if there are critical points in the gesture set that is being built. A study on the gesture design process\(^7\) has revealed that designers find it difficult to discover and solve problems in the gesture sets they create: POLYRec GDT tries to help in this by providing feedback on gesture set quality and by showing possible problems.

The tool provides warnings and statistics to inform the designer and allows him to make changes if he deems it necessary. Some analysis on the gesture set is made when a template is added, while others can be launched by the developer when he deems appropriate.
An issue in designing a gesture design tool is whether it should be dependent (or independent) on the gesture recognizer. The advantages of recognizing independence are obvious: the tool can be used with any recognition technology during both the design phase and in the deployment phase. On the other hand, by using features that are dependent on the recognizer, the tool can offer better advice in the design phase, but at the cost of nonportability. POLYREC GDT is equipped with POLYREC, but in the settings it is possible to choose whether to use POLYREC, $P$, $P+$ or $Q$, and it is relatively simple to add support for other recognizers. However, the provided suggestions are dependent on the selected recognizer, so in order to exploit them correctly, it is necessary to use the same recognizer both in the design and in the application performing recognition. In addition, the effectiveness of these suggestions may vary depending on the characteristics of the recognizer and its calculated scores, and some features may not be supported, for instance, the choice per template of recognition mode (rotation sensitive or invariant). Moreover, POLYREC GDT offers some functionalities, such as the possibility to create and modify training templates and to display tables with the geometric features of the gesture, which can be used with any recognizer (even those not integrated into the tool).

Even though POLYREC is a unistroke recognizer, it allows simple multifinger gestures. In this case, the number of fingers used is recorded and the comparison is made, on templates with the same number, by using the track of a single finger (so this is only feasible for gestures in which the fingers remain united).

The tool has been implemented in Java and allows easy integration of the recognizer in Android applications.

POLYREC GDT consists of two main display modes: the dashboard view and the template view. The dashboard mode represents the main screen (see Section 4.3) that gives a complete view of the gesture classes and contains the main commands to operate on it. The template view (see Section 4.4), that can be accessed from the dashboard, is the one in which it is possible to add or perform operations on a single template, also through companion apps for android devices.
(see Section 4.1). In both displays modes a menu to access common functionalities (e.g., save, load) and a title bar that provides information on the current location and allows to return to the dashboard are shown (see the top of Figure 7). The following sections describe these components in detail.

4.1 | POLYREC companion apps

In addition to allowing gestures to be entered via the pc cursor that is typically controlled by mouse or touchpad, POLYREC GDT provides two Android apps that allow the user to enter gestures directly on a smartphone touchscreen, in order to allow the designer to enter gestures performed in a natural way. Both apps have similar functionalities and can be paired and used from the template view. Figure 5 shows the drawing interface of the Bluetooth app. The WiFi app has a similar interface.

The two apps differ for the method with which they communicate with POLYREC GDT: one uses Bluetooth and the other one uses WiFi (TCP/IP). POLYREC GDT can also send the Bluetooth app installation file directly to the Android device, while the WiFi app can be only downloaded from the POLYREC GDT site.

4.2 | Menu

The menu, shown in Figure 6, can be called independently of the current application view. From top to bottom it allows sending the companion app via Bluetooth to an Android smartphone, to export the Java code that allows to use the recognizer, to import sample gestures from predefined datasets, to create a new gesture set, to open an existing gesture set (also by adding its gestures to the current set), to save the gesture set, to change application options (used gesture recognizer and warning thresholds) or to exit it.

4.2.1 | Export recognizer

This feature allows the user to export the Java code that allows the gesture recognition, using the selected gesture recognizer, for the current gesture set. The user can simply copy the generated code to the clipboard and paste it into the Java class where he needs to perform gesture recognition. In order to simplify the recognizer usage and sped-up operations, the generated Java code directly contains the templates (codified as strings) and a handleGesture() method stub.

![Figure 5](image-url) POLYREC companion app (Bluetooth version). Drawing screen (after connection between smartphone and pc) before and after the execution of a gesture.
4.3 Dashboard view

The dashboard view is composed of several components, as shown in Figure 7. In particular from top to bottom:

- **Message bar**: this bar shows messages about the last operation performed on the gesture set. For some operation, the system also uses popup windows to display longer messages.
- **Operations bar**: this bar contains buttons that allow the execution of commands on the gesture set. In particular, it allows, from left to right, to add a new gesture class, to merge two or more gesture classes, to delete all classes, to check the similarity between them (see Section 4.3.1), to display the gesture set features table (see Section 4.3.2) or to test the recognizer (see Section 4.4.1).
- **Gesture set table**: each row of this table shows a class of the gesture set and its templates.
  - The first column shows the class name and the number of templates that compose them, and also contains buttons that allow to perform operations such as: change the name of the class, remove it, set the class templates as rotation invariant or rotation sensitive, show the features table for the class templates (see Section 4.3.2), to call the automatic templates selection functionality (see Section 4.3.3).
- In each of the following columns, one of the class templates is shown, and by clicking on it, it is possible to switch to the template view. There are also buttons that allow to: move a template to another class, delete the template, display if the template is rotation invariant or rotation sensitive and possibly change this property.
- The last column contains a $\oplus$ button that allows adding a new template to the class (by opening the template view).
- Status bar: this bar contains summary information on the number of classes, the number of templates, and the average number of templates per class.

4.3.1 | Gesture set classes similarity analysis

Long et al.\(^6^7\) carried out an experiment in which the participants were asked to design a set of gestures and try to get the best possible recognition accuracy for the Rubine\(^4^2\) recognition algorithm. No participant managed to exceed the recognition rate of 95.4%. A typical problem that was observed was that the participants tended to add strokes that were too similar to those already defined.

If two gestures are similar to the recognizer, they are more likely to be confused. In order to help designers find this potential problem, POLYREC GDT provides a feature that allows the user to check for similarity issues between the gesture set classes.

By pressing the “verify similarity” button in the dashboard, a window is shown (see Figure 8) that displays the list of “conflicts” between templates belonging to different classes: a conflict is found if two templates have a similarity that exceeds a certain threshold. The system compares each template with all the templates belonging to the other classes, if the comparison score exceeds the threshold (by default 88%) then a warning is displayed, which indicates that there is an anomaly that may require actions.

Within the window, it is possible to change the similarity threshold and immediately recheck the conflicts with the new threshold.

Besides that, it is also shown what in POLYREC GDT has been defined as the similarity set score of the gesture set. It is equal to the maximum similarity score found when performing the comparisons. As it is defined, the goal is to keep this score as low as possible: having a low similarity score indicates that the classes are properly distinct. Therefore, if the similarity score of the set exceeds the default threshold, the system shows a warning.

4.3.2 | Gesture set features table

The system allows the user to view the values of 13 (mainly geometric) features for each of the templates of a class or of the whole gesture set. As shown in Figure 9 the values are presented in a table that contains a row for each template and that allows to export the data as a .csv file or to show it in a chart.

![Figure 8](image-url) Verification of similarity: negative outcome (left) and positive outcome (right)
These 13 features can be useful for understanding the complexity and differences of classes and templates and are calculated after normalizing the gestures (i.e., scaling them to the same size).

Some of the features (indicated with fn in the list) are those proposed by Rubine\cite{rubine42} and are commonly used in the study of gestures.

1. number of segments of the polyline generated by the POLYREC recognizer;
2. total sum of angles or total curvature (f10);
3. length of the gesture path (f8);
4. distance between the first and last point (f5);
5. area of the bounding box;
6. length of the bounding box diagonal (f3);
7. angle of the bounding box diagonal (f4)
8. cosine of the initial angle (f1);
9. sine of the initial angle (f2);
10. cosine of the final angle;
11. angle of the line that joins the first and the last point of the gesture;
12. cosine of the angle of the line that joins the first and the last point of the gesture (f6);
13. sine of the angle of the line that joins the first and the last point of the gesture (f7).

It is also possible to show these values only for a single template by calling this feature from the template view.

4.3.3 Automatic templates selection

The performance of a recognizer is measured primarily by the recognition rate, which is an estimate of the percentage of gestures that are expected to be correctly recognized. In addition to the recognition rate, there are other variables, such as the time required for classification and the amount of system memory used to store the gesture set. Their importance is particularly relevant for mobile computing where resources are limited. Some studies have demonstrated the benefits of clustering for restructuring the gesture set and improving recognition accuracy and performance. For example, Ouyang and Li\cite{ouyang68} used clustering to merge similar gesture patterns provided by many contributors in an attempt to build a large, constantly evolving dictionary of gestures for touchscreen mobile devices. Keskin et al.\cite{keskin69} used it as a preprocessing step to reconfigure the number and structure of gesture classes according to the similarity between templates in the training set. Inspired by the success of these approaches that use clustering algorithms to improve recognition performance, we decided to use clustering techniques also in POLYREC GDT. If a recognition algorithm based on template matching is used for real-time applications, it is often necessary to limit the number of templates with which the user input is compared in
order to limit the required computational resources. It is then necessary to choose how many and which templates to select for each class. This is because different classes require a different number of templates: for example, some handwritten digits have more variations than others (typically more variations in 4 than in 0).

For this reason, PolyRec GDT provides an automatic templates selection feature which allows the user to reduce the number of templates in a gesture class by automatically selecting the most representative templates. Given the desired number of templates $k$, the system calculates, using the selected gesture recognizer, the distance between each template and the others within the class, then applies the k-medoids$^{66}$ clustering algorithm to split the templates into $k$ clusters and for each of them selects a template (medoid) to be kept. Alternatively, the system also offers the possibility of dividing the existing class into new $k$ classes corresponding to the found clusters.

The desired number of templates $k$ can be selected by the user. Also, since a developer often does not know which $k$ number is the most suitable, the system proposes to the user a $k$ value. This $k$ is calculated by using the silhouette$^{70}$ analysis on different $k$ values in order to select the “better” number of clusters.

The issue of selecting the optimal subset of gestures has already been discussed in the literature. In Reference 71 the authors present an intelligent template selection for the $S$-family recognizers that makes use of a random mutation hill climb algorithm and a genetic algorithm to search for a reduced sets of templates that minimize recognition error and shows that the error rate can be reduced compared with random template selection. In our opinion, however, techniques based on error rate reduction are reliable when applied to a dataset with a substantial number of gestures (by different users). Since PolyRec GDT is aimed directly at application developers (rather than at gesture recognizer designers or gesture dataset creators), we think it is likely that it will deal with a small number of templates drawn directly by the developer. In this case techniques based on error rate reduction could cause undesired side effects, for example, a developer may intentionally add to a gesture class a single template very different from the others in order to support a different drawing style, but a methodology based on error rate reduction could remove it, since it does not help in recognizing the other (many) gestures in the set. The technique we used, instead, being based on k-medoids, has a better behavior in this case.

The usefulness of this function for PolyRec has been analyzed in Section 5.

4.4 Template view

The template view can be accessed from the dashboard to:

1. add a new template;
2. perform recognition tests;
3. view an existing template and its details.

The central part of the view is always used to show a gesture trace, either a real-time one drawn by the user (1, 2) or a template (3).

4.4.1 Add a new template and recognition tests

In this mode, the user can add a new template to a class, either by drawing it with the system cursor or through the companion app (by clicking the corresponding buttons at the top of the view). After the gesture has been performed (see Figure 10) the system compares it with the templates that are already part of the gesture set and displays the result in the message bar at the top, and shows a warning if the recognized class is different from the class in which the gesture would be added. Moreover, on the right side, there is a button that allows the user to view the score table. This table shows the distance and the score obtained by comparing the just inputted gesture with those already present in the gesture set, limiting it to the most similar one for each class. This allows the user to detect, for example, possible recognition problems. The table can also be used to generate a chart as shown in Figure 11.

If the user is satisfied, he can add the gesture to the current class using the corresponding button at the bottom of the view. He can also change the number of fingers/pointers associated with the gesture and whether it should be rotation invariant/sensitive by using the corresponding buttons. Moreover, on the left side, the previews of the templates belonging to the same class are displayed, and it is possible to click on them to view them in detail (see next section).
When the user chooses to perform a recognition test from the dashboard, an interface similar to the one described above is shown, with the only difference that the left and bottom bars are missing.

### 4.4.2 Existing template view

The application enters this mode when the user saves a new template or when the user clicks on the preview of a template.
As shown in Figure 12, the gesture is displayed at the center in the dimensions in which it was drawn. At the top of the view, buttons show and allow to modify the number of fingers/pointers of the gesture and whether the gesture is rotation invariant. By using the toolbar at the bottom the user can change how the gesture is visualized by changing zoom, rotation, translation. On the left side, the other templates belonging to the same class of the current template are displayed. On the right side, there are buttons to:

- enable/disable the visualization of the gesture path;
- enable/disable the visualization of the polyline created, from the original gesture, by the POLYREC recognizer;
- enable/disable the visualization of the vertices of the polyline created by the POLYREC recognizer;
- visualize a timed animation of the gesture, shown at the same speed with which it was entered;
- show the gesture features table, as described in Section 4.3.2;
- show the score table, as described in the next section.

5 | EVALUATION OF THE AUTOMATIC TEMPLATES SELECTION

In order to evaluate the usefulness of the automatic templates selection method in a real-life case, the work done by the developers of a gestural keyboard for entering programming code on mobile devices\(^72\) has been analyzed.

The developers have defined a set of gestures consisting of 14 classes (each one represents a programming construct), and have then collected the gestures from five people. Each of them performed the gesture of each class four times, for a total of 280 samples. The developers, not having a tool like POLYREC GDT, based the choice of templates to be used in their application on their perception (3 templates for each class).

We have therefore decided to evaluate the quality of the templates selected with the POLYREC GDT automatic templates selection method by comparing them to the ones selected by the developers.
The accuracy of the recognition on the gesture set collected by the developers was therefore measured for three different conditions regarding the choice of the templates: three random templates for each class, the templates chosen by the developers, three templates for each class selected using automatic templates selection method. The accuracy was calculated by averaging the one obtained individually for each of the five people that performed the gestures in the dataset. For each person, the recognition test was repeated 100 times by selecting a random gesture and performing the recognition for the three conditions (selecting the templates from those of the other four participants).

The results are shown in Figure 13. In the case of the randomly selected templates, the error rate is 2.69%, while it is 2.50% in the case of the templates selected by the developers, with only a small accuracy increase. In the case of the templates selected with the automatic templates selection method, instead, the error rate dropped to 1.66%, thus showing that the use of POLYREC GDT may provide useful help to developers.

6 | USABILITY EVALUATION

In order to evaluate POLYREC GDT, a user study has been performed in which participants were asked to use it to support them in the addition of gesture-based functionalities to an existing mobile application.

6.1 | Participants

For the experiment, 12 participants (four females) between 21 and 29 years old (mean 24.25, standard deviation 2.53) were recruited. All of them were university students in computer science who agreed to participate for free. All of them already had experience with touchscreen devices, using their smartphones every day. They were divided into two groups, the first one consisting of the six participants who declared they had a good knowledge of Android programming, while the other composed of the other six participants who declared no/low knowledge of Android programming (but a good knowledge of Java).

6.2 | Apparatus

Each participant performed the experiment on their notebook, appropriately configured for the development of Android applications. They also used their Android smartphone to draw gestures with POLYREC GDT and as a development device. An alternative device was provided to the few participants without a personal Android smartphone.

6.3 | Procedure

In the experiment invitation mail, the participants were asked to appropriately configure their Notebook for Android applications development by installing Android Studio and by compiling/running two stub example applications, in order to download the required dependencies and check the proper functioning of the development environment.
Moreover, they were also asked to read and try to comprehend the code that handles the GestureOverlayView component in the first example.

Before starting the experiment the participants were asked to fill out a form with the following information: age, gender, English knowledge level, experience level with smartphones, Java programming experience, and Android programming experience.

Then, they received a brief tutorial showing the functionalities POLYREC GDT. They were also asked to install POLYREC GDT on their notebook and the POLYREC GDT WiFi app on their smartphone and try POLYREC GDT functionalities. They were then asked to download and run the POLYREC Android Example from GitHub, open it in Android studio, run it on their device, and read the main parts of its source code. They had to perform the same operations for the “Pairs Game” Android application, a simple game in which the user has to match two pairs of cards depicting geometric figures.

Then the objectives of the experiment and its procedure were described to them. They were also given an instruction sheet, containing the description of the three tasks to perform to complete the experiment. The experiment started after the experimenter ascertained that the participant had well understood the procedure.

Task 1 consisted of using the POLYREC GDT tool to create a gesture set for the videogame “Pairs Game.” The gestures must be used to perform the following actions:

- **up**: moves the cursor up;
- **down**: moves the cursor down;
- **left**: moves the cursor left;
- **right**: moves the cursor right;
- **guess circle**: bet that in the selected position there is a circle;
- **guess square**: bet that in the selected position there is a square;
- **guess triangle**: bet that in the selected position there is a triangle;
- **guess check**: bet that in the selected position there is a check symbol.

When the participant is satisfied with the gesture design he must save the gesture set in a file called SurnameName.pgs. The time limit for this task is 15 min.

Task 2 consisted of modifying the “Pairs Game” videogame by: hiding the buttons for the eight operations (up, down, left, right, guess circle, guess square, guess triangle, guess check), adding a GestureOverlayView to the layout of the videogame to allow the user to perform a gesture in the application view area and see its trace, make the application log/print, when the user lifts the finger, the coordinates of all the points of the performed gesture. When the modification is completed without errors the participant has to commit its changes to the application. The time limit for this task is 25 min. Since task 3 depends on this task, in case the participant was not able to complete it successfully, an operator helped him complete it before starting task 3.

Finally, task 3 consisted of continuing the editing of “Pairs Game” by recognizing the gesture performed by the user using the gesture set defined in task 1 and performing the corresponding game operation (up, down, left, right, guess circle, guess square, guess triangle, guess check). If need the previously defined gesture set may be changed. When the modification is completed without errors the participant has to commit its changes to the application (and the gesture set, if modified). The time limit for this task is 25 min.

After the end of the experiment, participants were asked to fill out a System Usability Scale (SUS) questionnaire. The SUS questionnaire consists of 10 statements to which the user assigns a score on a scale from 1 (strongly disagree) to 5 (strongly agree). The final score of the SUS ranges from 0 to 100. A higher score indicates greater user usability. Moreover, after the experiment, user free-form comments and suggestions were also collected. In particular, at the end of the questionnaire, there was a blank space where each participant could write his/her comments and suggestions about POLYREC GDT.

### 6.4 Design

The experiment was a one-way with one between-subjects factor design. The one factor was the group (nonexpert and expert). For each task, the experiment-dependent variables were the task completion time and the task success/failures. To evaluate user satisfaction the SUS was used.
All participants completed the experiment. The experiment took each of them, excluding the introductory phase, up to two hours. We tested for significance using a repeated measures analysis of variance (ANOVA). The overall task completion times per task and group are shown in Figure 14. As it can be seen, all participants completed task 1 successfully and the two groups used similar times, 7:54 min (SD = 2:44) and 7:50 min (SD = 1:06) respectively for the nonexpert and expert groups. From the ANOVA resulted that the difference between the groups on completion time was not statistically significant ($F_{1,10} = 0.004, p > .05$). This provides a good sign on the usability of PolyRec GDT.

As for task 2, in the nonexpert group, two-thirds of the participants failed to complete the two tasks in the 25 min allocated time, with an average time of 22:35 min (SD = 3:47), while in the expert group all participants completed the task with an average time of 11:43 min (SD = 2:00). From the ANOVA resulted that the difference between the groups on completion time was statistically significant ($F_{1,10} = 38.817, p = .0001$). The difference on task failures was also statistically significant ($F_{1,10} = 10.000, p = .0101$).

As for task 3, in the nonexpert group, two-thirds of the participants failed to complete the two tasks in the 25 min allocated time, with an average time of 21:45 min (SD = 6:13), while in the expert group all participants completed the task with an average time of 8:54 min (SD = 5:06). From the ANOVA resulted that the difference between the groups on completion time was statistically significant ($F_{1,10} = 15.345, p = .0029$). The difference on task failures was also statistically significant ($F_{1,10} = 10.000, p = .0101$).

As would be widely expected, there was a significant difference between expert and nonexpert participants, although the results of task 1 show that all participants were able to define their set of gestures. The results on tasks 2 and 3 show that despite the initial tutorial, modifying an Android application was too difficult for participants with little experience in the field, even though one third of them succeeded in the available time.

On the other hand, for experienced participants, adding gesture functionalities took only a few minutes, which would have been instead impossible in the allocated time without the support of PolyRec GDT.

### 6.6 User satisfaction and free-form comments

The average SUS score was 65.83 for the nonexpert group (SD = 25.33) and 76.25 (SD = 6.85) for the expert group with a grand mean of 71.04 (SD = 18.51). This can be considered a good result.

Regarding the free-form comments, most nonexperts participants highlighted the difficulty of completing tasks 2 and 3 due to their lack of experience with Android programming. Comments about PolyRec GDT were mostly positive, such as one participant who said “I found PolyRec GDT software extremely useful if you want to implement an application that uses gestures. If I were to implement such an application, I would use this software.” Most of them liked the possibility to generate the Java code to recognize the gestures, stating that it greatly sped up the implementation. Some participants appreciated the possibility of making gestures, instead of using the mouse, directly on a smartphone thanks to the companion app. In this regard, however, some participants noted an instability on their hardware of the WiFi connection with the app, which did not compromise its use but made it necessary to repeat the connection multiple times. One user suggested offering the option of defining a gesture as “bidirectional,” similar to how it is possible to define one
as rotation invariant, so that when desired it will not be necessary to add a second gesture with the same shape but the opposite direction. Finally, one user suggested giving the possibility to sort the classes in the dashboard in a custom way (they are now in alphabetical order), while another one complained about the ambiguity of the $\oplus$ button when adding a new template.

When asked how they would perform the same tasks without having POLYREC GDT available, they gave two types of answers: a minority who stated that they would not know how to do it, and those who stated that they would instead modify an existing app (or create a new one) in order to capture their gestures. Most of them said they would modify an existing app, either the first stub app (which already has a GestureOverlayView) or the PolyRec Android Example available on GitHub, in order to be able to capture gestures made on the smartphone. In this respect, most said they would simply print on the log console (Logcat) the $x$ and $y$ coordinates of the points composing the gesture, and then copy and paste that data into Java code. Instead, another party stated that they would offer the ability to save the gestures performed as files on the smartphone, which they would then copy to the computer. When asked, they all answered that, even after learning about it thanks to POLYREC GDT, they would not perform any particular operations to choose the best gestures for the recognition, considering it too difficult to perform it manually. This could also be due to their low experience about gesture recognition, as well as the difficulty to perform them without a support tool. In fact, on this subject, none of them answered that they would search online for a tool for capturing/managing gestures.

6.7 | Control experiment

In order to assess the difficulty of performing the above tasks without POLYREC GDT, we programmed a control experiment. For this purpose we started a pilot study with three “expert” participants (two females) between 21 and 28 years old, university students in computer science who had taken the course on Android programming and who had not already participated in the previous study.

They followed a procedure similar to the one for the experiment described above, with the difference that they were not instructed on how to use POLYREC GDT (but just POLYREC, using the examples and documentation on GitHub) and were asked to perform the tasks using the methodology they thought appropriate. We increased the time available to complete the tasks, setting it at 50 min for all of them. The time taken for the three tasks by the participants was respectively 47:40 (SD = 2.05), 15:20 (SD = 4:44), and 45:20 (SD = 8.05) min. One participant was unable to successfully complete task 1 and 3, while all successfully completed task 2.

As was to be expected, not having POLYREC GDT available caused a significant increase in the time required to complete tasks 1 and 3, in which in the previous experiment POLYREC GDT was respectively used to design the gesture set and to generate the Java code for gesture recognition, with an average time of about six or five times higher than in the expert group of the previous experiment. In task 2, which does not benefit from the use of POLYREC GDT, the difference is small. Even for those who successfully completed the tasks, we saw less attention paid to the quality of the created gesture set, with typically only one sample used for each gesture class.

Regarding the used methodology, the two participants who successfully completed all the tasks modified the PolyRec Android Example in order to capture their own gestures and print their points on the log console (Logcat). The participant who failed to complete tasks 1 and 3 instead tried to create a new app to capture the gestures, but failed to capture the points correctly (resulting in task failure).

In comments after the experiment, participants stated that since they had never built Android apps with gesture recognition, they had to spend time figuring out how to operate and searching for documentation/answers online. When asked, one of them stated that while searching online he also tried to see if he could find an app or website to capture the gestures on his smartphone, but desisted after a few attempts, while the other two participants stated that they did not think about it.

Considered the poor results of the above described study, we felt it was useless to continue with a full-scale experiment.

6.8 | Discussion

From the results and comments of the participants, it can be seen that POLYREC GDT was appreciated and it allows to define the gesture set for a simple smartphone application in a short time. The functionality offered by POLYREC GDT to export the Java code needed to use the recognizer also allows users familiar with Android programming to add gesture
capability to an existing application in a few minutes, while without a similar tool the time required would have been much longer. As expected, however, it was difficult for participants not familiar with Android programming to integrate gesture recognition into existing software. Nevertheless, the experiment showed that they too were able to define their gesture set. In addition, the control experiment confirmed that the use of POLYREC GDT allows for a better and faster workflow compared with a development without a gesture management tool. It would also be useful to increase developer awareness of such tools, for example, it might be useful to have a reference gesture management tool mentioned in the documentation (and/or code comments) of gesture recognizers.

7 | CONCLUSIONS AND FURTHER WORKS

We presented POLYREC GESTURE DESIGN TOOL (POLYREC GDT), a tool that allows the definition of gesture sets for mobile applications that uses the POLYREC gesture recognizer. POLYREC GDT supports the entire gesture set design process, from collecting gestures, to analyzing and exporting the set created for use in a mobile application together with the POLYREC recognizer.

POLYREC GDT also allows to detect similarities between gestures in order to help the designer in detecting possible ambiguities, and thanks to a clustering technique it can automatically select the most representative gestures for each class of gestures, in order to allow the right compromise between execution speed and recognition accuracy. The usefulness of this function is supported by experimental results that have shown greater accuracy than the same operation done manually by developers or randomly by software.

In addition, an experiment was carried out to evaluate POLYREC GDT, in which participants were asked to use it to support them during the development of a mobile application. The results showed that adding gesture functionality to an existing application only took a few minutes for developers with mobile development experience, which would have taken much longer without POLYREC GDT.

Future work will include small improvements to user interaction with POLYREC GDT following feedback from its users, and support for the iOS system (in addition to Android) to support apps for both major mobile operating systems.

ACKNOWLEDGMENT

This work was partially supported by two grants from the University of Salerno: 300392FRB19COSTA and 300392FRB20COSTA.

ORCID

Gennaro Costagliola https://orcid.org/0000-0003-3816-7765
Mattia De Rosa https://orcid.org/0000-0002-6922-5529
Vittorio Fuccella https://orcid.org/0000-0003-3244-3650

REFERENCES

1. Poppinga B, Sahami Shirazi A, Henze N, Heuten W, Boll S. Understanding shortcut gestures on mobile touch devices. Proceedings of the 16th International Conference on Human-Computer Interaction with Mobile Devices & Service; 2014:173-182.
2. Blickenstorfer CH. Graffiti: Wow!!!! Pen Comput Mag. 1995;1:30-31.
3. Kristensson PO, Zhai S. SHARK2: a large vocabulary shorthand writing system for pen-based computers. Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology, UIST ’04. ACM; 2004:43-52.
4. Fuccella V, De Rosa M, Costagliola G. Novice and expert performance of KeyScretch: a gesture-based text entry method for touch-screens. IEEE Trans Human-Mach Syst. 2014;44(4):511-523. https://doi.org/10.1109/THMS.2014.2314447
5. Costagliola G, De Rosa M, Fuccella V. Handwriting on smartwatches: an empirical investigation. IEEE Trans Human-Mach Syst. 2017;47(6):1100-1109. https://doi.org/10.1109/THMS.2017.2754938
6. Costagliola G, De Rosa M, D’Arco R, De Gregorio S, Fuccella V, Lupo D. C-QWERTY: a text entry method for circular smartwatches. Proceedings of the 25th International DMS Conference on Visualization and Visual Languages; 2019:51-57. C-QWERTY: a text entry method for circular smartwatches; 2019:51-57. https://doi.org/10.18293/DMSVIVA2019-014
7. Costagliola G, De Rosa M, Fuccella V. A technique for improving text editing on touchscreen devices. J Vis Lang Comput. 2018;47:1-8. https://doi.org/10.1016/j.jvlc.2018.04.002
8. Fuccella V, Costagliola G. Unistroke gesture recognition through polyline approximation and alignment. Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI ’15. ACM; 2015:3351-3354. https://doi.acm.org/10.1145/2702123.2702505
9. Long AC, Landay JA, Rowe LA. 'Those Look Similar!': issues in automating gesture design advice. Proceedings of the 2001 Workshop on Perceptive User Interfaces, PUI ’01. ACM; 2001:1-5.
69. Keskin C, Cemgil AT, Akarun L. DTW based clustering to improve hand gesture recognition. In: Salah AA, Lepri B, eds. Human Behavior Understanding. Vol 72–81. Springer; 2011.
70. Rousseeuw PJ. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. J Comput Appl Math. 1987;20:53-65. https://doi.org/10.1016/0377-0427(87)90125-7
71. Pittman C, Taranta EMM, LaViola JJ. A $-$family friendly approach to prototype selection. IUI '16. ACM; 2016:370-374.
72. Costagliola G, Fuccella V, Leo A, Lomasto I, Romano S. The design and evaluation of a gestural keyboard for entering programming code on mobile devices. Proceedings of the 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC); 2018:49-56.
73. Brooke J. SUS-A quick and dirty usability scale. Usability Eval Ind. 1996;189(194):4-7.
74. Girden E. ANOVA: Repeated Measures. No. 84 in ANOVA: Repeated Measures. SAGE Publications; 1992.
75. Bangor A, Kortum P, Miller J. Determining what individual SUS scores mean: adding an adjective rating scale. J Usability Stud. 2009;4(3):114-123.

How to cite this article: Bufano R, Costagliola G, De Rosa M, Fuccella V. POLYREC GESTURE DESIGN TOOL: A tool for fast prototyping of gesture-based mobile applications. Softw Pract Exper. 2022;52(2):594-618. https://doi.org/10.1002/spe.3024