iGesture: A Java Framework for the Development and Deployment of Stroke-Based Online Gesture Recognition Algorithms

Beat Signer
Institute for Information Systems, ETH Zurich
8092 Zurich, Switzerland
signer@inf.ethz.ch

Moira C. Norrie
Institute for Information Systems, ETH Zurich
8092 Zurich, Switzerland
norrie@inf.ethz.ch

Ueli Kurmann
Institute for Information Systems, ETH Zurich
8092 Zurich, Switzerland
ueli@smartness.ch

ABSTRACT

Existing gesture recognition tools and frameworks tend to focus on specific settings, gesture sets or algorithms. Further, they are often designed to support the developers of either applications or algorithms, but not both. Our goal was to develop a general and extensible framework that provides an integrated platform for the design and evaluation of algorithms, as well as for their deployment to a wide audience. The presented iGesture framework supports the definition and evaluation of new gesture sets. Furthermore, our gesture recognition framework enables an easy integration of new forms of input devices. We present the iGesture framework, show how it has been used to support the development of two new gesture recognition algorithms—an extension of Rubine called E-Rubine and SiGrid—and finally provide an evaluation of these algorithms.

General Terms

Algorithms, design, experimentation

Keywords

Gesture recognition algorithms, gesture recognition framework, pen-based input

1. INTRODUCTION

Emerging technologies for mobile and ubiquitous computing have led to increased interest in the use of gestures as a means of interacting with applications. Thus, in addition to traditional uses of gesture-based interaction in desktop environments, an increasing number of applications are using pen-based gesture recognition for not only PDAs and tablet PCs, but also interactive paper [13]. At the same time, the use of gestures is being widely investigated in the fields of pervasive and wearable computing to interact with applications based on either the movement of physical objects or parts of the body.

As a result, there has been a lot of research not only on the development of gesture recognition algorithms but also tools and frameworks to support the development of applications based on these algorithms. However, existing systems tend to either focus on specific settings—such as screen-based input—or on specific gesture sets and/or algorithms. Our goal was to develop a general and extensible gesture recognition framework that would support both application developers and the developers of new algorithms. The resulting iGesture framework [9] provides a simple gesture recognition application programming interface (API) that allows iGesture results to either be used directly or through an event manager that executes commands based on recognised gestures.

To facilitate the development of new algorithms as well as the evaluation of existing algorithms, the Java-based iGesture framework also supports the creation and management of gesture sets. A test bench enables the manual testing of algorithms and special functionality is provided to create the test data. We provide tools to evaluate different gesture recognition algorithms and their configurations in batch mode and visualise the results. Note that these evaluation tools can not only be used to analyse existing or new gesture recognition algorithms but also for verifying new gesture classes. For example, the result an evaluation will reveal if two gesture classes look too similar and therefore cannot be classified correctly by the recognition algorithm. This information helps designers of new gesture sets to define unambiguous gesture sets resulting in better gesture recognition performance.

In this paper, we present the iGesture framework and, as a demonstration of its effectiveness in supporting the development and evaluation of new gesture recognition algorithms, we describe two new algorithms that were developed and tested using the framework. The first one is an extension of the well-known Rubine algorithm, called E-Rubine, to handle single-stroke as well as multi-stroke features. The second algorithm is a simple grid-based signature algorithm called SiGrid that our evaluations have shown to be simple but effective in some application settings. We present these algorithms and also report on comparative evaluations generated using the iGesture test bench.

We start with an outline of related work and the requirements of a general gesture recognition framework in Section 2. The architecture and some implementation details of our iGesture framework are presented in Section 3. In Section 4 we then go on to present the algorithms currently supported by the gesture recognition framework, including our two new algorithms, and provide an initial evaluation of these algorithms in Section 5. Concluding remarks are given in Section 6.

2. BACKGROUND

A number of tools and frameworks for gesture recognition have been developed and we review the main ones in this section in order to highlight the benefits and requirements of a general, integrated framework.

GRANDMA (Gesture Recognizers Automated in a Novel Direct Manipulation Architecture) is an object-oriented toolkit for rapidly adding gestures to direct manipulation interfaces that was proposed by Rubine in 1991 [12]. It introduced a specific classification algorithm, now commonly referred to as the Rubine algorithm, that uses statistical single-stroke gesture recognition based on 13 different features. New gestures could be introduced by specifying a set of sample gestures, thereby enabling application developers to easily design their own gesture-based interfaces without having to program the corresponding recognition algorithms.

Hong and Landay later developed *SATIN* [8], a toolkit for informal ink-based applications that uses the Rubine algorithm in its recognition process. SATIN is a powerful toolkit targeted at pen-based applications and a major part of the framework deals with the interpretation and beautification of stroke data to build ink-based graphical Java Swing applications. A feature of SATIN was the introduction of *pie menus* to replace common pop-up menus.

SwingGestures [15] is another framework designed to support gestures in Java Swing applications. In this case, eight basic gestures (up, down, left, right and the four diagonals) are hard coded and other gestures can only be constructed based on these eight gestures. This severely limits the kinds of gestures that can be supported as it is nearly impossible to model curving gestures using only these basic gestures. Furthermore, the framework was not designed to support different recognition algorithms. While the tool is useful for simple Swing-based applications, it does not seem to scale well in terms of the number of supported gestures.

Microsoft also provides a gesture recognition tool in the form of the *Microsoft Tablet PC SDK* [4] that distinguishes between *system* and *user application gestures*. Microsoft provides a recommended interpretation of their gestures that developers should follow to avoid confusion. Unfortunately, the recogniser of the Microsoft Tablet PC SDK is limited to a predefined set of gestures and new gestures can only be integrated by implementing new recognition algorithms. Further, the Microsoft Tablet PC SDK was designed for screen-based applications and is implemented on the .NET platform.

A problem of developing gesture-based user interfaces is often the similarity of specific gestures which makes it difficult to develop robust gesture recognisers. *quill* [10] is a gesture design toolkit that addresses this problem by providing active feedback to the gesture designers when there is an ambiguity between different gestures. In such cases, it also assists them with textual advice to enable them to create more reliable gesture sets. A computational model based on geometric gesture features is used to measure the similarity of gestures.

From the above outline of existing systems, we can see that existing toolkits and frameworks tend to focus on specific settings and tasks. None of them supports the entire process of designing new gesture recognition algorithms, including deployment to a wide audience. The development of new algorithms should be supported by providing benchmarking and parameter optimisation tools. A gesture recognition framework should also provide tools to define new gestures and to efficiently capture gesture samples for testing. It should also be possible to work with different input devices and to easily integrate new devices to cater for emerging technologies and new modes of interaction. Last, but not least, the framework should offer an easy-to-use interface that enables application programmers to fully exploit the potential of gesture-based interfaces and latest developments in gesture recognition algorithms. Based on these requirements, we have developed a Java-based gesture recognition framework to support an integrated solution for the use and development of gesture recognition algorithms with a focus on extensibility and cross-application reusability.

3. IGESTURE FRAMEWORK

The iGesture framework is based on a set of modular components and some common data structures as shown in Figure 1. We will start with a description of the common data structures and then go on to describe each of the components in turn, starting with the recogniser and then going on to describe the management console and evaluation tools.

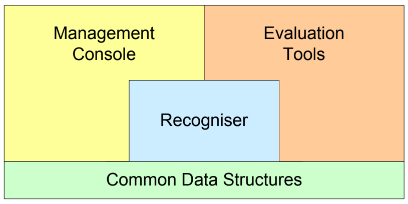


Figure 1: Architecture overview

The representation of gestures within the iGesture framework was a fundamental design decision since it had implications on all components. One requirement for the data structure was that it should be possible to represent single gestures as well as groups of gestures. Furthermore, it was clear that different algorithms need different descriptions of a gesture. Therefore, it is important that the model classes do not make any assumptions about a specific algorithm or provide algorithm-specific data. The UML class diagram of our general gesture data structure is shown in Figure 2.

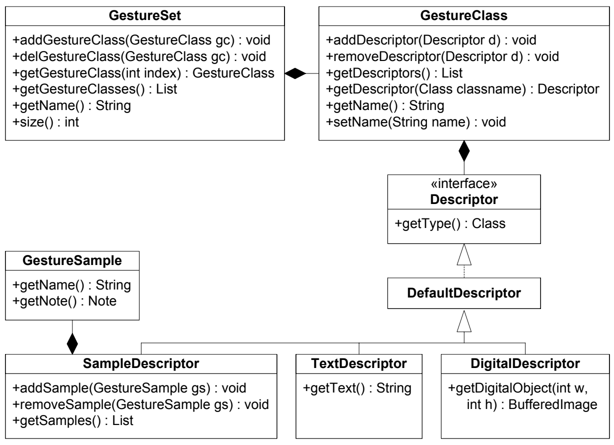
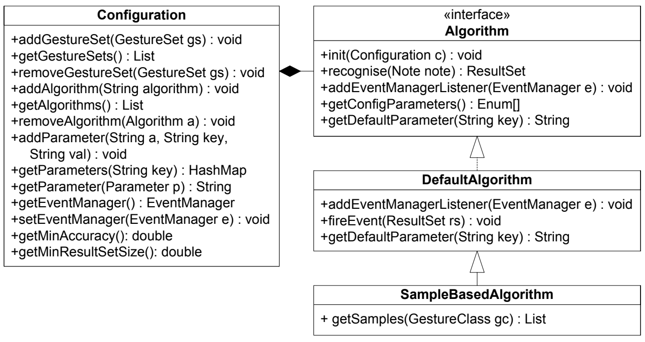

GestureSet
+addGestureClass(GestureClass gc) : void
+delGestureClass(GestureClass gc) : void
+getGestureClass(int index) : GestureClass
+getGestureClasses() : List
+getName() : String
+size() : int
GestureClass
+addDescriptor(Descriptor d) : void
+removeDescriptor(Descriptor d) : void
+getDescriptors() : List
+getDescriptor(Class classname) : Descriptor
+getName() : String
+setName(String name) : void
«interface» Descriptor
+getType() : Class
DefaultDescriptor
GestureSample
+getName() : String
+getNote() : Note
SampleDescriptor
+addSample(GestureSample gs) : void
+removeSample(GestureSample gs) : void
+getSamples() : List
TextDescriptor
+getText() : String
DigitalDescriptor
+getDigitalObject(int w, int h) : BufferedImage

Figure 2: Gesture representation

The `GestureClass` class represents an abstract gesture characterised by its name and a list of descriptors. For example, to cope with circles as a specific gesture, we instantiate a new `Gesture-Class` and set its name to ‘Circle’. Note that the class itself does not contain any information about what the gesture looks like and needs at least one descriptor specifying the circle as a graphical object. A set of gesture classes is grouped in a `GestureSet` which can then be used to initialise an algorithm. The `Descriptor` interface has to be implemented by any gesture descriptor. For example, we provide the `SampleDescriptor` class describing gestures by samples which is used by training-based algorithms. A single GestureSample is an instance of a gesture and contains the note captured by an input device. In addition to the sample descriptor, we provide a textual description of the directions between characteristic points of a gesture as well as a digital descriptor describing the gesture in terms of a digital image. Note that the digital descriptor is not used in the recognition process but rather acts as a visualisation for a recognised gesture to be used, for example, in graphical user interfaces.

In addition, we need a mechanism to persistently store any gesture samples for later retrieval. Again, our goal was to be flexible and not to rely on a single mechanism for storing data objects. The iGesture storage manager encapsulates any access to persistent data objects and uses a specific implementation of a storage engine interface to interact with the data source. We decided to use *db4objects* [7], an open source object database for Java, as the primary storage container. However, we also implemented a second storage engine that simply serialises the data objects into an XML document based on the x-stream Java library [17].

An iGesture recogniser is a general component that can be configured to use different recognition algorithms. To provide maximal flexibility in the design and use of algorithms, we decided to provide a compact interface as highlighted in Figure 3. The Algorithm interface provides methods for the initialisation, the recognition process, the registration of an event manager and for retrieving optional parameters and their default values.


Configuration
+addGestureSet(GestureSet gs) : void
+getGestureSets() : List
+removeGestureSet(GestureSet gs) : void
+addAlgorithm(String algorithm) : void
+getAlgorithms() : List
+removeAlgorithm(Algorithm a) : void
+addParameter(String a, String key,
String val) : void
+getParameters(String key) : HashMap
+getParameter(Parameter p) : String
+getEventManager() : EventManager
+setEventManager(EventManager e) : void
+getMinAccuracy(): double
+getMinResultSetSize(): double

«interface»
Algorithm
+init(Configuration c) : void
+recognise(Note note) : ResultSet
+addEventManagerListener(EventManager e) : void
+getConfigParameters() : Enum[]
+getDefaultParameter(String key) : String

DefaultAlgorithm
+addEventManagerListener(EventManager e) : void
+fireEvent(ResultSet rs) : void
+getDefaultParameter(String key) : String

SampleBasedAlgorithm
+ getSamples(GestureClass gc) : List

Figure 3: Algorithm class diagram

An algorithm always has to be initialised with an instance of the Configuration class containing gesture sets, an optional event manager and algorithm-specific parameters which are managed in a key/value collection. This configuration object can be created using the Java API or by importing the data from an XML document. The framework further offers an algorithm factory class [5] to instantiate algorithms based on information handled by a configuration instance.

While the algorithm interface is mainly used by the designer of new recognition algorithms, the application developer has access to the framework's recogniser component—configured with one or more recognition algorithms—based on a single Recogniser class (facade pattern) shown in Figure 4.

Note that multiple algorithms may be specified in a single configuration. The Recogniser class provides two methods with different behaviours depending on whether a single or multiple algorithms have been specified. The recognise(Note note) method goes through the algorithms in sequential order and terminates the recognition process as soon as an algorithm returns a valid result whereas the recognise(Note note, boolean all) method combines the results returned by all of the algorithms. A Note represents our data structure for storing information captured by an input device. Each Note contains one or more strokes consisting of a list of timestamped positions. The Recogniser always returns a result set which is either empty or contains an ordered list of result objects. We decided to return a set of potential result objects instead of a single one to enable specific applications to use any additional contextual information in the selection process.

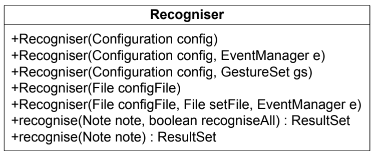
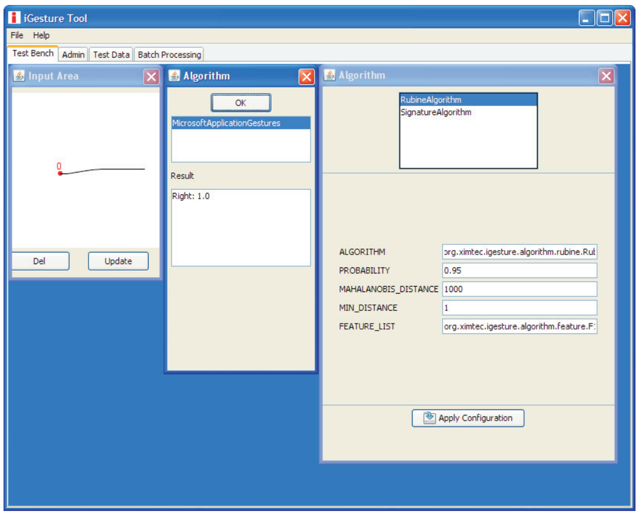

Recogniser
+Recogniser(Configuration config)
+Recogniser(Configuration config, EventManager e)
+Recogniser(Configuration config, GestureSet gs)
+Recogniser(File configFile)
+Recogniser(File configFile, File setFile, EventManager e)
+recognise(Note note, boolean recogniseAll) : ResultSet
+recognise(Note note) : ResultSet

Figure 4: Recogniser API

The iGesture framework's management console has been implemented as a Java Swing application providing support for testing and defining gestures as well as creating test sets as represented by the Test Bench, Admin, Test Data and Batch Processing tab components. The test bench tab, shown in Figure 5, provides functionality to acquire a single gesture from an input device and execute the recogniser with the chosen gesture set and algorithm. This enables a simple and quick manual testing of specific gestures. The entire collection of gesture sets of the currently opened persistent storage container are available and any registered algorithm may be used. This manual testing of gestures is relevant to get initial feedback about the quality of a specific algorithm's recognition rate.



MicrosoftApplicationGestures

Result
Right: 1.0

RubineAlgorithm
SignatureAlgorithm

ALGORITHM org.ximtec.igesture.algorithm.rubine.Rul
PROBABILITY 0.95
MAHALANOBIS_DISTANCE 1000
MIN_DISTANCE 1
FEATURE_LIST org.ximtec.igesture.algorithm.feature.F:

Figure 5: iGesture test bench tab

The admin tab is used to manage gesture sets, gesture classes and the corresponding descriptors as illustrated in Figure 6. New gestures are captured with the aid of the input device and shown in the Input Area. From there they can, for example, be added to the sample descriptor of a given gesture class.

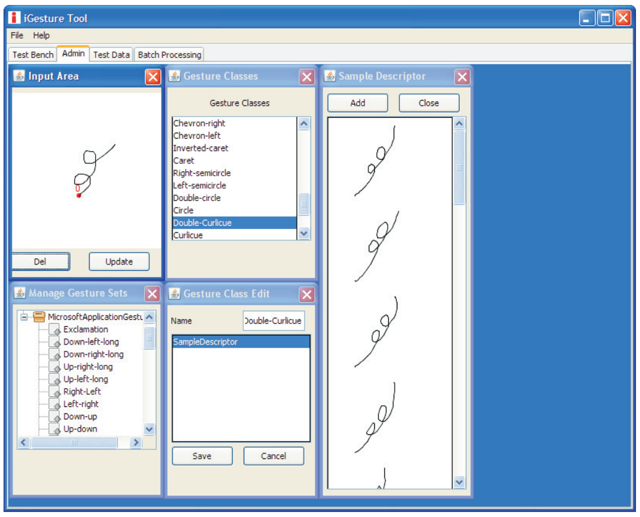


Figure 6: iGesture admin tab

Furthermore, the admin tab enables the creation, editing and deletion of gesture classes as well as the manipulation of descriptors and gesture sets. Additionally, functionality is provided to export and import complete gesture sets together with the corresponding gesture classes and their descriptors to/from a single XML document. These XML files can later be used to initialise the recogniser component independently of a specific storage manager.

The test data tab is used to create test sets for testing algorithms and their configurations. Any test set can be exported to an XML file which may then be used as a source for an automatic batch process evaluation. Specific batch process parameters can be defined in the batch processing tab. The goal of the batch processing tool is to simplify the evaluation of new algorithms and enable the comparison of different algorithms. It further supports the designer of a new algorithm in adjusting and optimising different algorithm parameters by means of running a single algorithm with different settings. A batch process is configured with an XML file specifying the configuration objects to be created. We provide a flexible way to specify an algorithm's parameters in terms of a combination of fixed values, sequences, ranges and power sets for specific parameters as shown in the example of Figure 7.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<iGestureBatch>
<algorithm name="org.igesture.alg.SiGridAlgorithm">
  <parameter name="GRID_SIZE">
    <for start="8" end="16" step="2" />
  </parameter>
  <parameter name="DISTANCE_FUNCTION">
    <sequence>
      <value>HammingDistance</value>
      <value>LevenshteinDistance</value>
    </sequence>
  </parameter>
  <parameter name="MIN_DISTANCE">
    <for start="1" end="5" step="1" />
  </parameter>
</algorithm>
</iGestureBatch>
```

Figure 7: XML batch configuration

Based on the XML configuration, all possible parameter permutations are generated and, for each configuration, the batch process instantiates the algorithm and processes the given test gesture set.

The results of a batch process containing the key figures for each run and gesture class, such as *precision*, *recall* and *F-measure*, as well as the configuration of the parameters, are collected in a test result data structure which is stored in an XML document. We also provide some XSLT templates to render the results as an HTML document and sort the data based on specific key figures. This allows the designer of a new recognition algorithm to easily identify the most promising parameter settings for a given algorithm and test set.

The algorithm designer also has to provide a configuration stored in a file which can be used by the application developer to instantiate the recogniser with a given algorithm and parameter set. Note that the settings stored in the configuration file may be a direct result of previous batch process evaluations. The code snippet in Figure 8 shows how easy it is for the application programmer to use the gesture recognition engine.

```java
Recogniser recogniser = new Recogniser(
    ConfigurationTool.importXML("config.xml"));
ResultSet result = recogniser.recognise(note);

if (!result.isEmpty() {
  logger.log(result.getResult().getName());
}
```

Figure 8: Recogniser

In addition to the explicit handling of the results by the client application, iGesture also provides an event manager where a client can register actions to be triggered when a specific gesture class has been recognised.

We mainly use the digital pen and paper technology provided by the Swedish company Anoto [1] as an input device for the iGesture framework. However, since the iGesture framework should not depend on a specific hardware technology, all the components work on an abstract input device interface. This makes it easy to integrate new devices and use them for capturing new gesture sets as well as controlling specific applications. So far we support different Anoto digital pens as well as the standard computer mouse as a gesture input device. Furthermore, we are currently integrating the Wintab tablet API [16] to also acquire data from arbitrary graphics tablet solutions.

To simplify the time-consuming process of capturing gesture samples from different users, we provide a component to generate interactive gesture capture paper forms as shown in Figure 9. After a set of gesture classes has been defined, the corresponding interactive paper forms can be generated automatically and printed out with the position encoding pattern provided by Anoto.

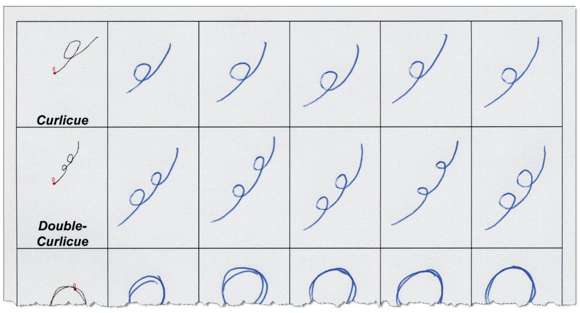


Figure 9: Interactive paper capture form

Each row of the form contains a sample of the gesture to be captured and some empty fields that the user has to fill in with the digital pen. The pen data is continuously streamed to the iGesture application and stored in the corresponding gesture sample set. To enable the exchange of gesture sets with other applications, we further provide support for importing and exporting documents in *Ink Markup Language* (InkML) [3] format.

4. ALGORITHMS

In this section, we introduce four different gesture recognition algorithms that we have implemented so far for the iGesture framework. These include two existing solutions—the *Rubine* algorithm for specifying gestures by example [12] and the *Simple Gesture Recogniser* (SiGeR) algorithm developed by Swigart [14]. In addition we have implemented two new algorithms which we also present in this section. *E-Rubine* is an extended version of the Rubine algorithm dealing with additional single and multi-stroke features while *SiGrid* is a new signature-based algorithm that classifies gestures based on distance functions. Note that the implementations of the different gesture recognition algorithms presented in this section conform to the `Algorithm` interface introduced earlier.

4.1 Rubine Algorihtm

The Rubine algorithm represents one of the first algorithms published for the recognition of mouse and pen-based gestures. The gestures are not described programmatically but instead learnt by examples. With the appropriate tools, such as the iGesture management console, it is easy to create new gestures and add them to the gesture recognition engine. In our implementation of the Rubine algorithm, we use the equations described in the original Rubine paper [12]. We now present the main features of the algorithm and show some of the equations involved in order that the reader can obtain, not only an understanding of the algorithm, but also how it is integrated into iGesture and how we extended it in E-Rubine.

Features are extracted from gestures which consist of timestamped points and then used in the recognition process. In Rubine's paper 13 features are described to be used in the recognition process. This includes the cosine and sine of the initial angle with respect to the x-axis (f_1 and f_2), the length of the bounding box diagonal (f_3), the angle of the bounding box (f_4), the distance between the first and last point (f_5), the cosine and sine of the angle between the first and last point (f_6 and f_7), the total gesture length (f_8), the total traversed angle (f_9, f_{10} and f_{11}), the maximum speed squared (f_{12}) and the stroke duration (f_{13}). For the implementation of the standard Rubine algorithm evaluated later in this paper, we implemented these 13 features.

The classification does not depend on specific features which allows us to use it for different recognition tasks as long as the classifiable objects can be described by feature vectors. In a first step, the feature vector $f_{\hat{c}ei}$ is computed for each example gesture e for $0 \leq i < F$, where F is the number of selected features. These vectors are then summarised in a mean vector for each gesture class $\hat{c}$, where $E_{\hat{c}}$ represents the number of training examples for gesture class c. The mean vector $\bar{f}_{\hat{c}}$ is simply the average of the sample feature vectors for a given gesture class as illustrated in Equation 1.

$$\bar{f}_{\hat{c}i} = \frac{1}{E_{\hat{c}}} \sum_{e=0}^{E_{\hat{c}}-1} f_{\hat{c}ei} \tag{1}$$

Based on these feature vectors, the covariance matrix $M_{\hat{c}}$ shown in Equation 2 is computed.

$$M_{\hat{c}ij} = \sum_{e=0}^{E_{\hat{c}}-1} \left(f_{\hat{c}ei} - \bar{f}_{\hat{c}i}\right)\left(f_{\hat{c}ej} - \bar{f}_{\hat{c}j}\right) \tag{2}$$

For each gesture class and the covariance matrices are averaged to a single covariance matrix M as shown in Equation 3, with C representing the number of gesture classes.

$$M_{ij} = \frac{\sum_{c=0}^{C-1} \frac{M_{\hat{c}ij}}{E_{\hat{c}}-1}}{-C + \sum_{c=0}^{C-1} E_{\hat{c}}} \tag{3}$$

The single covariance matrix allows us to estimate the weights $\omega_{\hat{c}j}$ of the vector components shown in Equation 4 and the initial weight $\omega_{\hat{c}0}$ for each gesture class as illustrated in Equation 5.

$$\omega_{\hat{c}j} = \sum_{i=1}^{F} \left(M^{-1}\right)_{ij} \bar{f}_{\hat{c}i} \tag{4}$$

$$\omega_{\hat{c}0} = -\frac{1}{2} \sum_{i=1}^{F} \omega_{\hat{c}i} \bar{f}_{\hat{c}i} \tag{5}$$

Note that all these steps can be accomplished in the algorithm's initialisation phase and the weights computed for each gesture class will not change during classifications. The classification itself is realised with the linear function shown in Equation 6. For an input gesture, the feature vector with the same features is computed and each component of the vector is multiplied with the corresponding gesture class weight. The classified gesture is denoted by the gesture class yielding the maximal value.

$$v_{\hat{c}} = \omega_{\hat{c}0} + \sum_{i=1}^{F} \omega_{\hat{c}i} f_i \tag{6}$$

This kind of classification has the drawback that a result will always be returned even if an input gesture does not have any similarities with a trained example gesture. Therefore, Rubine proposed a mechanism for rejecting gestures which are not similar to the trained ones and also the ones which are ambiguous. In our implementation a threshold can be set through an `AMBIGUITY` parameter to reject gestures which are too close to multiple gesture classes. Furthermore, to detect outliers the *Mahalanobis distance* [11] introduced in Equation 7 is used.

$$\delta^2 = \sum_{j=1}^{F} \sum_{k=1}^{F} \left(M^{-1}\right)_{jk} \left(f_j - \bar{f}_{ij}\right)\left(f_k - \bar{f}_{ik}\right) \tag{7}$$

In addition to the `AMBIGUITY` parameter introduced earlier, in our implementation of the Rubine algorithm, other parameters can be specified. The `MIN_DISTANCE` denotes the minimal distance between two succeeding points of a gesture. This enables us to automatically filter points that are too close together and could have a negative impact in the computation of specific features. A feature of our Rubine implementation has to conform to the `Feature` interface. The `FEATURE_LIST` parameter holds a list of comma-separated fully qualified class names of feature objects that will be instantiated in the algorithm's initialisation phase using dynamic class loading. Finally, the maximal distance for outliers can be defined by the `MAHALANOBIS_DISTANCE` parameter. For further details about Rubine's algorithm and feature definitions please refer to [12].

4.2 E-Rubine Algorithm

For our extended version of the Rubine algorithm, we developed new single-stroke and multi-stroke features. Our E-Rubine algorithm uses the core of the Rubine implementation presented in the previous section together with the new features presented in this section. We first introduce the single stroke features (f_{14} to f_{19}) and then present an extension for multi-stroke gestures (f_{20} to f_{24}). Note that some of our new features contain references to the original Rubine features (f_1 to f_{13}).

4.2.1 *Single-Stroke Features*

A single-stroke gesture S can be defined as a totally ordered set of points $p_i = (x_i, y_i)$, where $1 \leq i \leq n$.

$$S = \{p_1, \ldots, p_n\} \tag{8}$$

The x and y coordinates of any given point p_i can be accessed by the $.x$ and $.y$ operators. Furthermore, we define a function φ always returning the last point of a given list of points.

$$\text{If } S = \{p_1, \ldots, p_n\}, \text{ then } \varphi(S) = p_n \tag{9}$$

The distance δ between two points p_i and p_j can then be defined as

$$\delta(p_i, p_j) = \sqrt{(p_j.x - p_i.x)^2 + (p_j.y - p_i.y)^2} \tag{10}$$

Number of stop points
Our first new single stroke feature f_{14} is the number of so-called stop points within a gesture. A stop point occurs when the drawing speed falls below a specific lower bound threshold value. The average speed for the entire gesture is computed and the lower bound value is set to one third of the average speed. The speed between two points is then always compared to this lower bound value. Finally, the number of stop points is defined by the number of sequences having a speed below the specified lower bound. Stop points typically occur on directional changes and at the end of strokes. If $StopPoints$ is the set of all stop points, then

$$f_{14} = |StopPoints| \tag{11}$$

Distance from start to centre point in relation to the diagonal
This feature measures whether the first or second part of the gesture is more winding. An integer division is used to access the centre point $S[n/2]$ in the computation of the distance d_1 from the start point to the centre point as shown in Figure 10.

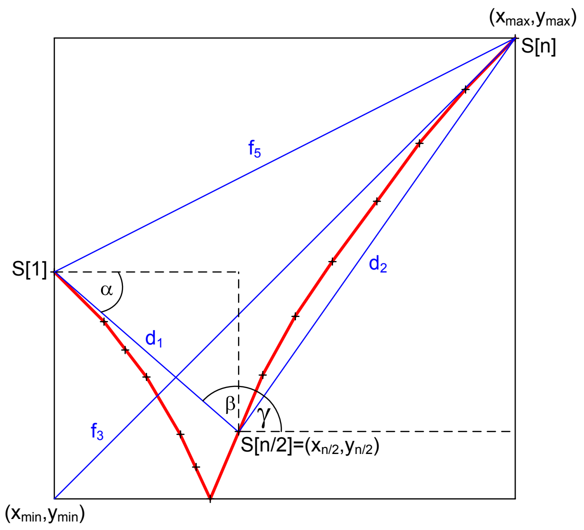


Figure 10: E-Rubine single stroke features

$$d_1 = \delta(S[1], S[n/2]) \tag{12}$$

$$f_{15} = \frac{d_1}{f_3} \tag{13}$$

Direction of the first half of the stroke
Feature f_{16} measures the general direction of the first part of the stroke based on the sine of the angle between d_1 and the x-axis.

$$f_{16} = \sin\alpha \ = \ \frac{S[n/2].y - S[1].y}{d_1} \tag{14}$$

Direction of the second half of the stroke
The cosine of the angle between the straight line d_2 from the centre point to the end point of the single-stroke gesture and the x-axis is used to compute f_{17}.

$$d_2 = \delta\left(S[n/2], S[n]\right) \tag{15}$$

$$f_{17} = \cos\gamma \ = \ \frac{S[n].x - S[n/2].x}{d_2} \tag{16}$$

Angle between the first and the second half of the stroke
The cosine between d_1 and d_2 is used to compute this feature describing the change of direction with respect to the gesture's centre point.

$$f_{18} = \cos\beta \ = \ \frac{{d_1}^2 + {d_2}^2 - {f_5}^2}{2d_1d_2} \quad (17)$$

Distance from start to end point in relation to the diagonal
Our last single stroke feature f_{19} is a gesture size-independent replacement of Rubine's original feature f_5.

$$f_{19} = \frac{f_5}{f_3} \quad (18)$$

4.2.2 Multi-Stroke Features

To improve the performance of the original Rubine algorithm for new multi-stroke gestures, we designed and implemented five additional multi-stroke features. Note that these features are used in combination with the single stroke features. A multi-stroke gesture is defined as a totally ordered set M of m strokes $S_1, \ldots S_m$

$$M = \{S_1, \ldots, S_m\} \quad (19)$$

Each stroke S_i is thereby defined as a totally ordered set of points $p_j = (x_j, y_j)$, where $1 \leq j \leq |S_i|$.

$$S_i = \{p_1, \ldots, p_{|S_i|}\} \quad (20)$$

Number of strokes
This feature takes into account the number of strokes that a gesture contains. Note that if most of the gestures to be recognised contain the same number of strokes, feature f_{20} may not be a good classifier.

$$f_{20} = |M| \quad (21)$$

Straightness
The straightness of a multi-stroke gesture is represented by our new feature f_{21} dividing the sum of each stroke's distance from the start to its end point ($\Delta(S_i)$) by the sum of the stroke lengths ($l(S_i)$).

$$\Delta(S_i) = \delta(S_i[1], \varphi(S_i)) \quad (22)$$

$$l(S_i) = \sum_{j=1}^{|S_i|-1} \delta\left(S_i[j], S_i[j+1]\right) \quad (23)$$

$$f_{21} = \frac{\sum_{i=1}^{|M|} l(S_i)}{\sum_{i=1}^{|M|} \Delta(S_i)} \quad (24)$$

Total distance between strokes
Feature f_{22} sums up the distances between the strokes and thereby provides information about their distribution. In order to make the feature size-independent, the sum is divided by the diagonal of the gesture's bounding box.

$$f_{22} = \frac{1}{f_3} \sum_{i=1}^{|M|-1} \delta\left(\varphi\left(S_i\right), S_{i+1}[1]\right) \quad (25)$$

Angle between strokes
Feature f_{23} sums up the angles between the different strokes. The angle for each stroke, represented by the angle between the straight line from its start point to its end point and the x-axis, is computed and subtracted from the previous stroke's angle.

$$f_{23} = \sum_{i=1}^{|M|-1} \mathit{Angle\ between\ Stroke}\ S_i\ \mathit{and}\ S_{i+1} \quad (26)$$

Stroke distances in relation to each other
Last but not least, feature f_{24} puts the stroke distances $\Delta(S_i)$ in relation to each other.

$$f_{24} = \Delta(S_1) \prod_{i=2}^{|M|} \frac{1}{\Delta(S_i)} \quad (27)$$

Clearly, there was a lot of experimentation to test the effectiveness of proposed new features and compare the performance of the extended algorithm to that of the original Rubine algorithm and also other algorithms. The iGesture test bench gave us a lot of support for these experiments and enabled us to quickly arrive at an extension that yields promising results as reported in Section 5.

4.3 SiGeR Algorithm

The third algorithm that we have implemented with the help of the iGesture framework is the *SiGeR* algorithm that classifies gestures based on regular expressions [14]. Gestures are described with the eight cardinal points `(N, NE, E, SE, S, SW, W and NW)` and some statistical information. Out of such a description, a regular expression is created. These regular expressions are then applied to input gestures and, in the case that a class description matches the input string, the corresponding gesture class is returned as a result. Therefore the classification is binary and it is not possible to rate different results.

For example, the gesture shown in Figure 11 starting at the red dot is described by the following character string: `E, N, W, S`. Out of these characters, the regular expression `(E)+(N)+(W)+(S)+` can be created. Because hand drawn lines may not always be straight, Swigart proposed a more general form of a regular expression which also accepts neighbouring distances. The extended regular expression for our example gesture would be `(NE|E|SE)+(NW|N|NE)+(SW|W|NW)+(SE|S|SW)+`.

The input gesture is transformed into a character string and the distance between two points is described by the directions corresponding to the cardinal points. Additionally, statistical information is

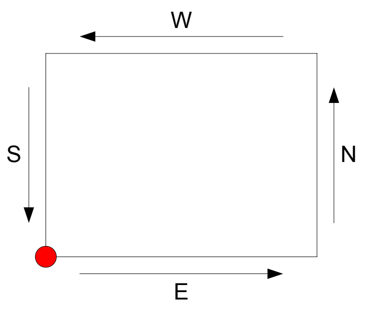


Figure 11: Gesture example

extracted out of the input gesture. Therefore, each direction is counted and information about the proportions of the directions provided. The proximity of the start and end points and the number of stop points is further counted to enable a more reliable description of gestures.

The description of gesture classes may impose constraints concerning this statistical information. For example, for a circle, we can define the constraints that there has to be an equal number of straight and diagonal elements and that the gesture's start and end point have to be close to each other.

Our implementation of the algorithm differs in some points from the original SiGeR version. The most important change is the possibility to describe gesture classes in textual form using some keywords to describe the directions and also to make use of the statistical information. This allows us to use the existing text descriptor for gesture classes. In our implementation of the SiGeR algorithm, gestures are described using the language shown in Figure 12.

```
description = directions [";" constraints];
directions  = direction ["," directions];
direction   = "N" | "NE" | "E" | "SE" |
              "S" | "SW" | "W" | "NW";
constraints = constraint ["AND"
              constraints];
constraint  = operand operator (operand |
              real);
operator    = "EQ" | "NEQ" | "GT" | "GTE" |
              "LT" | "LTE";
operand     = "N" | "NE" | "E" | "SE" |
              "S" | "SW" | "W" | "NW" |
              "DIAGONAL" | "STRAIGHT" |
              "PROXIMITY" | "STOPPOINTS";
real        = digit {digit} "." {digit};
digit       = "0" | "1" | "2" | "3" | "4" |
              "5" | "6" | "7" | "8" | "9";
```

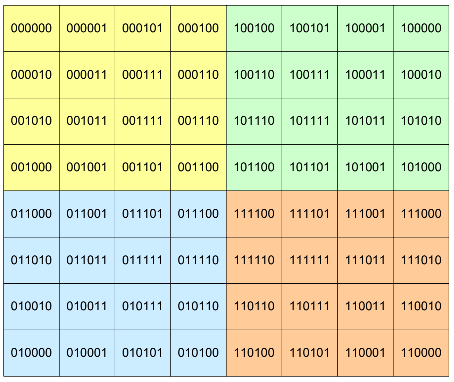
Figure 12: SiGeR description language

Using our new description language, the rectangle presented earlier would, for example, be described as `E,N,W,S;STRAIGHT GT 0.8 AND PROXIMITY LT 0.2`. The part before the semicolon describes the form of the rectangle with the directions. The two constraints define that at least 80% of the directions have to be straight and that the distance between the start and end point should be less than 20% of the gesture diagonal's length. The algorithm's `Configuration` object supports the definition of a `MIN_DISTANCE` parameter denoting the minimal distance between two succeeding points of a gesture.

4.4 SiGrid Algorithm

The *SiGrid* algorithm is a new algorithm that we developed that approximates gestures with a grid-based signature. The signature of a gesture class is computed based on sample gestures as done in the Rubine algorithm. These signatures are then compared to input gestures based on distance functions resulting in a classification of the gestures. To create the signatures for our new SiGrid algorithm, we use a grid consisting of equally sized squares. Each square has its unique bit string identifier and the identifier of two neighbouring squares always differs in exactly one bit.

During the preprocessing phase, the gestures are stretched to a uniform size and mapped to the grid. Each point of the gesture can now be represented with the bit string of its related square as shown in Figure 13. The full gesture signature consists of the concatenation of these bit strings.



000000	000001	000101	000100	100100	100101	100001	100000
000010	000011	000111	000110	100110	100111	100011	100010
001010	001011	001111	001110	101110	101111	101011	101010
001000	001001	001101	001100	101100	101101	101001	101000
011000	011001	011101	011100	111100	111101	111001	111000
011010	011011	011111	011110	111110	111111	111011	111010
010010	010011	010111	010110	110110	110111	110011	110010
010000	010001	010101	010100	110100	110101	110001	110000

Figure 13: Example 8×8 **grid**

Due to the limited resolution of the grid, the mapping of the gesture points to positions in the grid deliberately introduces some fuzziness into the gesture representation. The fuzziness is increased further by removing succeeding points that are mapped to the same grid square. In this way, we can guarantee that only significant points remain. In addition, our SiGrid algorithm applies a mechanism to remove points which do not change the signature significantly. A point is seen as irrelevant if the angle of the direction remains within a defined range.

We implemented two distance functions, the *Hamming Distance* and the *Levenshtein Distance*, for comparing different signatures and these can be selected by a parameter. Additional functions can be added by implementing the `DistanceFunction` interface. The Hamming Distance counts the number of bits which have to be flipped to make two signatures equal. The drawback of this function is that a displacement may trigger a lot of after-effects which may produce bad results. The Levenshtein distance is a generalisation of the Hamming distance. In addition to the flipping of bits, this measure is also able to add or remove bits to achieve the smallest possible distance between two bit strings. Based on this, after-effects can be minimised. The drawback of this function is that signatures might be changed drastically and so the result of the recognition is falsified. For each example gesture, the signature is created and an input gesture is compared with all the signatures. The accuracy of the result denotes the number of coinciding bytes.

Different parameters can be specified using the SiGrid algorithm's `Configuration` object. The `GRID_SIZE` parameter defines the number of cells within a single grid row. The `RASTER_SIZE` parameter defines the width and height to which a gesture will be stretched. The full qualified class name of a distance function implementing the `DistanceFunction` interface may be specified by the parameter `DISTANCE_FUNCTION`.

5. EVALUATION

We now provide initial results of the evaluation of the four algorithms presented. The three algorithms based on sample descriptors (i.e. Rubine, E-Rubine and SiGrid) and the textual descriptor-based SiGeR have been tested with three different kinds of gesture sets: the *Palm Graffiti* letters and numbers, the *Microsoft Application Gestures* and a customised set of multi-stroke gestures. Note that the preliminary experiments were used to evaluate the algorithms as well as a to validate the iGesture framework's batch processing functionality.

5.1 Key Figures

An input can be classified in one of three different result categories by an algorithm. First, the algorithm may correctly recognise an input as the gesture it actually represents. We name this category *Correct*. The *Error* category contains incorrectly recognised gestures. In this case, the algorithm returns a wrong result which may lead to unintentionally triggered actions. The third category contains the rejected gestures *Reject* which have been returned as unclassifiable by the algorithm. Note that the test sets that we used for our experiments contained only classifiable gestures and therefore as few gestures as possible should be rejected. We evaluated the algorithms using the notions of *Precision* and *Recall* which we defined as follows.

Precision

The *Precision* shown in Equation 28 denotes the proportion of correct results versus all results. A value of 1 means that all recognised gestures are identified correctly and the result contains no errors.

$$Precision = \frac{|Correct|}{|Correct| + |Error|} \tag{28}$$

Recall

The *Recall* illustrated in Equation 29 provides information about the fraction of gestures which can be classified out of all classifiable gestures.

$$Recall = \frac{|Correct| + |Error|}{|Correct| + |Error| + |Reject|} \tag{29}$$

F-Measure

The *F-Measure* shown in Equation 30 is the weighted harmonic mean of the precision and the recall value. We decided to weight precision and recall equally.

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{30}$$

5.2 Palm Graffiti

We ran five experiments using the 26 letters and 10 numbers of the Palm Graffiti alphabet [6] as a gesture set. They cannot be combined into a single gesture set because several gestures in the number set are similar to letters. Overall, we collected training and test data from four persons which provides for a more objective estimation than with a single user.

5.2.1 Experiment 1: Graffiti Numbers

Experiment 1 used the Graffiti numbers as the gesture set trained with 15 examples for each gesture class by one person. The test set had a size of 150 valid samples and was collected by three different persons. To test the SiGeR algorithm, a textual description of the numbers was constructed manually. Table 1 shows the results for the best configuration of each algorithm defined as the one with the maximal *F-Measure*. This means that the configuration should have a low error rate as well as a high number of correctly recognised gestures.

	E-Rubine	Rubine	SiGrid	SiGeR
Correct	140	134	132	129
Error	9	15	15	8
Reject	1	1	3	13
Precision	0.940	0.899	0.898	0.942
Recall	0.993	0.993	0.980	0.913
F-Measure	0.966	0.944	0.937	0.927

Table 1: Graffiti numbers

Although the example-based algorithms were trained by only one person, nearly all algorithms reached a *Precision* of at least 90%. The result shows that the new features used in our extended version of the Rubine algorithm significantly improve the recognition quality. Even the simple SiGrid algorithm has almost the same precision as the original implementation of the Rubine algorithm. The SiGeR algorithm also has good test results, but since the textual description of the gesture classes were done on the basis of the test set, it should be noted that they were optimised for this specific experiment.

5.2.2 Experiment 2: Graffiti Numbers

The Graffiti numbers were again used as the gesture set but the training data was collected from four different persons. We trained each gesture class with 4 times 4 examples and the test set has a size of 140 samples collected from a single person. We compared only sample-based algorithms and so the SiGeR algorithm is not taken into account. Again the test set contains only valid samples which should all be recognised.

	E-Rubine	Rubine	SiGrid
Correct	140	135	130
Error	0	4	10
Reject	0	1	0
Precision	1	0.971	0.929
Recall	1	0.993	1
F-Measure	1	0.982	0.963

Table 2: Graffiti numbers

The Rubine algorithm provides better key figures than in experiment 1 as shown in Table 2. The extended Rubine algorithm achieved a perfect result whereas the Signature algorithm has a

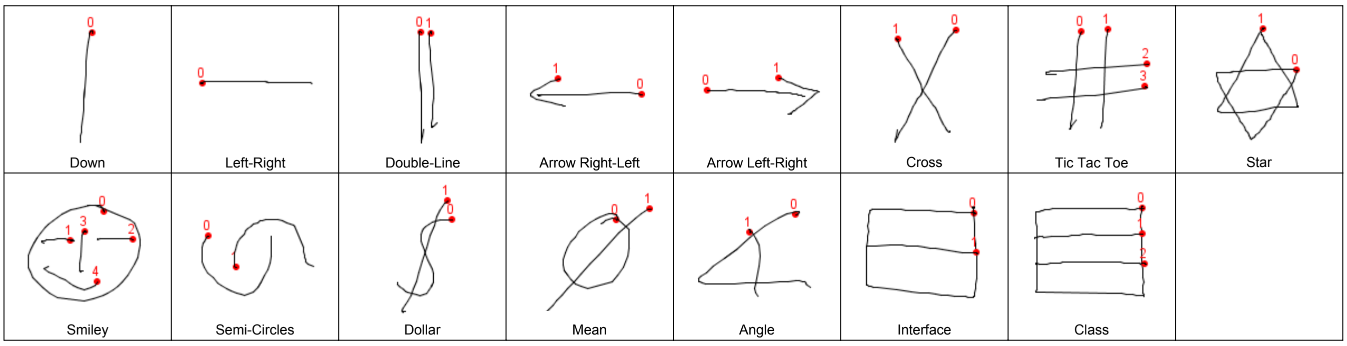


Figure 14: Multi-stroke gestures

higher error rate than the original Rubine algorithm. It can be assumed that the Rubine algorithm works significantly better with a broader variety of training data.

5.2.3 Experiment 3: Graffiti Letters

The 26 Graffiti letters were used as the gesture set and, as for Experiment 1, the training data was collected from a single person and the algorithms trained with 15 examples per gesture class. The test set has a size of 390 samples collected from three different persons. Again the test set contains only valid data.

	E-Rubine	Rubine	SiGrid
Correct	335	280	274
Error	52	107	113
Reject	3	3	3
Precision	0.866	0.724	0.708
Recall	0.992	0.992	0.992
F-Measure	0.925	0.837	0.826

Table 3: Graffiti letters

The figures shown in Table 3 are significantly worse than those for Experiment 1. The reason is the size of the gesture set which is 2.5 times larger while the size of the training data is the same. These results highlight that the number of samples has to grow with the size of the gesture set. We note that several letters had a very high error rate whereas others were recognised perfectly. The reason for this behaviour is that the training data was collected by a single person. Better results would be expected using training data from different users.

5.2.4 Experiment 4: Graffiti Letters

As in Experiment 2, each gesture class was trained with 4 times 4 examples from different users. The test set had a size of 363 samples and was produced by the same persons used for the training of the algorithm. Again the test set contains only valid gestures.

	E-Rubine	Rubine	SiGrid
Correct	343	305	297
Error	18	48	66
Reject	2	10	0
Precision	0.950	0.864	0.818
Recall	0.994	0.972	1
F-Measure	0.972	0.915	0.900

Table 4: Graffiti letters

Table 4 shows figures that are significantly higher than in Experiment 3 where the algorithms were trained by just one person and there are no longer several gesture classes which have poor recognition.

5.3 MS Application Gestures

The next set of experiments used the Microsoft Application Gestures [2] as the gesture set. Originally this set consisted of 42 gestures but two of them which represent mouse- or pen-clicks cannot be recognised with our feature-based algorithm because the gesture consists of only one or two points and does not provide enough input for computing the different features. Therefore the set used in the evaluation has a size of 40 gestures.

5.3.1 Experiment 5: MS Application Gestures

The 40 gestures were trained with 15 examples for each gesture class by one person and tested with 5 instances of each gesture class provided by the same person. Again the test set contains only valid gestures. Table 5 highlights that the extended Rubine algorithm had good performance even though the amount of training data used was relatively small compared to the size of the gesture set.

	E-Rubine	Rubine	SiGrid
Correct	195	178	164
Error	4	19	34
Reject	1	3	2
Precision	0.980	0.904	0.828
Recall	0.995	0.985	0.990
F-Measure	0.987	0.943	0.902

Table 5: MS application gestures

Although this experiment was clearly not realistic because the training data and the test data were produced by the same person, it still shows that with the extended Rubine algorithm good results can be achieved even with a small amount of training data if the coverage is good.

5.4 Multi-Stroke Gestures

We defined a set of 15 multi-stroke gestures shown in Figure 14. Two of them consist of a single stroke only and are prefixes of other multi-stroke gestures in the set. The set is quite small but it should be a proof of concept that also multi-stroke gestures can be recognised by the implemented algorithms.

5.4.1 *Experiment 6: Multi-Stroke Gestures*

The algorithms were trained with 15 examples collected from one person and tested with 5 samples for each gesture class collected from the same person. The results shown in Table 6 are good with algorithms having a precision higher than 96%. However, again good results were expected due to the algorithm being trained and tested by the same person. Furthermore, the relatively high number of training-examples for each gesture class has a positive effect.

	E-Rubine	Rubine	SiGrid
Correct	75	72	73
Error	0	3	2
Reject	0	0	0
Precision	1	0.960	0.973
Recall	1	1	1
F-Measure	1	0.980	0.986

Table 6: Multi-stroke gestures

5.4.2 *Experiment 7: Multi-Stroke Gestures*

In this experiment the algorithms were trained with 20 examples collected from one person and tested with 5 samples for each gesture class collected from another person. As shown in Table 7, all algorithms achieved nearly the same high level of results as in the previous experiment. This is probably due to the high number of training samples compared to the size of the gesture set and also the kind of gestures we used since they were quite distinct from each other.

	E-Rubine	Rubine	SiGrid
Correct	74	73	75
Error	1	1	0
Reject	0	1	0
Precision	0.987	0.986	1
Recall	1	0.987	1
F-Measure	0.993	0.987	1

Table 7: Multi-stroke gestures

The experiments have shown the behaviour of the chosen algorithms in different setups. These initial results show that only the extended Rubine algorithm seems to be good enough for general practical usage. However, the other algorithms may be useful in particular settings. Simple geometric figures in a small gesture set could also be recognised effectively with the SiGeR algorithm without having to create a large amount of example data and this in turn has the advantage of allowing gesture sets to be easily extended. The SiGrid algorithm also yielded good results when both the gesture set and the example sets were relatively small. In this setting, it may even be superior to the Rubine algorithm since it needs several examples per gesture class to compute the weights. If the number of gesture examples is too small, it is actually possible that the Rubine algorithm cannot be instantiated because the determinant of the covariance matrix is zero and therefore the inverse cannot be computed. In the future we plan to conduct further experiments with larger test sets.

6. CONCLUSION

We have presented a general and flexible framework for gesture recognition and described how we implemented two existing gesture recognition algorithms and also developed two new algorithms using the framework. In addition to providing interfaces that made it very simple to implement these algorithms and also use them in various applications, the iGesture framework supported the experimentation and evaluation processes. The fact that batch tests for different algorithms and parameter settings could easily be run against the same captured test sets greatly simplified the task of carrying out comparative evaluations and refinements of the algorithms. We believe that the fact that we were able to develop two new algorithms with very promising evaluation results within a short period of time can largely be attributed to the availability of the iGesture framework.

7. REFERENCES

[1] Anoto AB, http://www.anoto.com.

[2] MS Application Gestures, http://msdn2.microsoft.com.

[3] Y.-M. Chee, M. Froumentin, and S. M. Watt. Ink Markup Language (InkML). Technical report, W3C, October 2006.

[4] M. Egger. Find New Meaning in Your Ink With Tablet PC APIs in Windows Vista. Technical report, Microsoft Corporation, May 2006.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Palm Grafitti, http://www.palm.com.

[7] R. Grehan. The Database Behind the Brains. Technical report, db4objects, Inc., March 2006.

[8] J. I. Hong and J. A. Landay. SATIN: A Toolkit for Informal Ink-Based Applications. In *Proceedings of UIST 2000, 13th Annual ACM Symposium on User Interface Software and Technology*, pages 63–72, San Diego, USA, November 2000.

[9] iGesture, http://www.igesture.org.

[10] A. C. Long. *quill: A Gesture Design Tool for Pen-Based User Interfaces*. PhD thesis, University of California at Berkeley, 2001.

[11] P. C. Mahalanobis. On the Generalized Distance in Statistics. *Natl. Inst. Science*, 12:49–55, 1936.

[12] D. Rubine. Specifying Gestures by Example. In *Proceedings of ACM SIGGRAPH '93, 18th International Conference on Computer Graphics and Interactive Techniques*, pages 329–337, New York, USA, July 1991.

[13] B. Signer. *Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces*. PhD thesis, ETH Zurich, Switzerland, 2006.

[14] S. Swigart. Easily Write Custom Gesture Recognizers for Your Tablet PC Applications. Technical report, Microsoft Corporation, November 2005.

[15] Swing Gestures, http://sourceforge.net.

[16] WinTab specification 1.1, http://www.wacomeng.com.

[17] XStream, http://xstream.codehaus.org.