iGesture: A General Gesture Recognition Framework

Beat Signer, Ueli Kurmann, Moira C. Norrie
Institute for Information Systems, ETH Zurich
CH-8092 Zurich
{signer,norrie}@inf.ethz.ch

Abstract

With the emergence of digital pen and paper interfaces, there is a need for gesture recognition tools for digital pen input. While there exists a variety of gesture recognition frameworks, none of them addresses the issues of supporting application developers as well as the designers of new recognition algorithms and, at the same time, can be integrated with new forms of input devices such as digital pens. We introduce iGesture, a Java-based gesture recognition framework focusing on extensibility and cross-application reusability by providing an integrated solution that includes tools for gesture recognition as well as the creation and management of gesture sets for the evaluation and optimisation of new or existing gesture recognition algorithms. In addition to traditional screen-based interaction, iGesture provides a digital pen and paper interface.

1. Introduction

Over the last few years, we have developed a platform for interactive paper (iPaper) to integrate paper and digital media [8]. While the iPaper platform supports both the capture and real-time processing of digital pen information, so far there has been no support for gesture recognition. Since none of the existing gesture recognition frameworks completely satisfied our requirements, we decided to develop a general and extensible gesture recognition framework.

The resulting iGesture framework provides a simple gesture recognition application programming interface (API). An application can either handle iGesture results directly or use an event manager that executes commands based on recognised gestures. So far, we have implemented four different algorithms to be used in the gesture recognition process.

With our new iGesture tool, gesture sets can be created and managed. A test bench supports the manual testing of algorithms and special functionality is provided to create the test data. Last but not least, we provide tools to evaluate different gesture recognition algorithms and their configurations in batch mode and visualise the results. In addition to traditional screen-based interaction, the iGesture framework also provides a digital pen and paper interface enabling the capture of gestures from paper documents.

We start in Section 2 with an overview of existing mouse and pen-based gesture recognition frameworks and outline their main features as well as some of their limitations. In Section 3, we present the architecture of our iGesture framework and highlight some of its core features. Section 4 then describes the different user interfaces for designers of new gesture recognition algorithms as well as for application developers. Some comments about evaluating different gesture recognition algorithms and concluding remarks are given in Section 5.

2 Existing Gesture Recognition Frameworks

Specifying Gestures by Example [9] was published by Rubine in 1991. The paper describes *GRANDMA*, an object-oriented toolkit for rapidly adding gestures to direct manipulation interfaces and introduces a specific classification algorithm using statistical single-stroke gesture recognition based on 13 different features.

Hong and Landay developed *SATIN* [5], a Java-based toolkit for informal ink-based applications based on the Rubine algorithm. SATIN provides various components including different recognisers as well as the concept of a multi-interpreter supporting the creation of pen-based applications. The toolkit is targeted at screen-based applications and therefore mainly deals with the interpretation and beautification of stroke data to build ink-based graphical Swing applications.

A problem of gesture-based user interfaces is often the similarity of specific gestures which makes it difficult to develop robust gesture recognisers. *quill* [7] is a gesture design toolkit addressing this problem by providing active feedback to gesture designers when there is an ambiguity between different gestures and assisting them with textual advice to create more reliable gesture sets.

Microsoft provides a gesture recognition tool in the form of the *Microsoft Tablet PC SDK* [2] that distinguishes between system and user application gestures. Unfortunately, the recogniser is limited to a predefined set of gestures and new gestures can only be integrated by implementing new recognition algorithms. Due to its restriction to a fixed set of gestures, the tool cannot be used to investigate new gestures for digital pen and paper based user interfaces.

Finally, *Swing Gestures* [12] is a framework aiming to add simple gestures to Java Swing applications. 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, thereby limiting the power of the framework.

None of these frameworks meets all of the requirements of a general gesture recognition framework that caters for new forms of interaction such as paper-based interfaces as well as existing ones. Such a framework should be easy to use by application developers and, at the same time, be extendible for upcoming requirements of new applications. Furthermore, it should provide a platform for designers of new gesture recognition algorithms to implement their algorithms and make them available to a wide audience. The development of new algorithms should be supported by providing benchmarking and parameter optimisation tools. The framework should also provide tools to define new gestures and to efficiently capture gesture samples for testing.

3 iGesture Architecture

The iGesture framework is based on three main components—the recogniser, a management console and evaluation tools for testing and optimising algorithms—as shown in Figure 1. In addition, our framework provides some common data structures and model classes that are used by all three components. We describe the three main components and highlight some of our architectural choices to make the gesture recognition framework as flexible and extensible as possible.

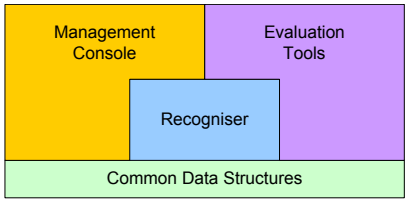

Figure 1. Architecture overview

One of the main goals of iGesture is to support different gesture recognition algorithms. To provide maximal flexibility in the design and use of algorithms, we decided to provide a compact interface as highlighted in Figure 2. 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.

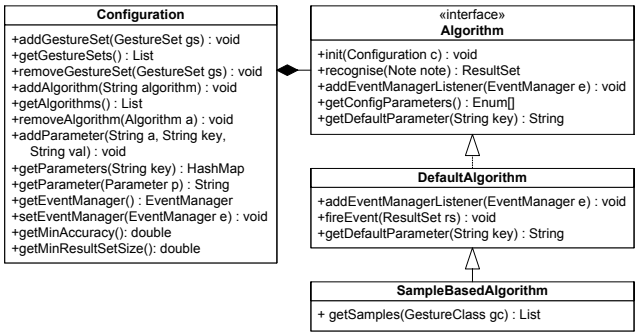

Figure 2. Algorithm class diagram

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 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 3.

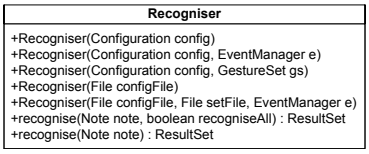

Figure 3. Recogniser API

In general, the `Recogniser` is initialised with a configuration object that contains information about the algorithms to be used and is loaded from an XML file. Note that multiple algorithms may be specified in a single configuration file. The `Recogniser` class provides two methods with different behaviours if multiple algorithms have been defined: the `recognise(Note note)` method goes through the algorithms in sequential order and terminates the recognition process as soon as the first algorithm returns a valid result whereas the `recognise(Note note, boolean recogniseAll)` method combines the results returned by all of the algorithms. The `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 locations. 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 results instead of a single one to enable potential applications to use any additional contextual information in the selection process.

The representation of gestures within the iGesture framework was a fundamental design decision since it had implications on all the other system parts depending on the gesture data structure. 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 4.

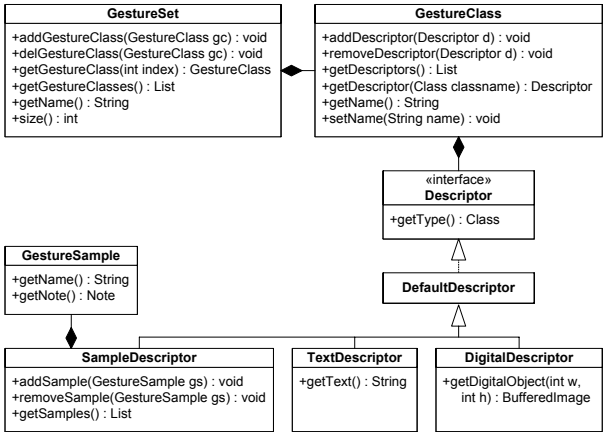

Figure 4. Gesture representation

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 `GestureClass` 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 instance, 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 offer a textual description specifying the directions between characteristic points of a gesture as well as a digital descriptor representing 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 concrete implementation of a storage engine interface to interact with the data source.

We decided to use *db4objects* [4], an open source object database for Java, as the primary storage container. However, we implemented a second storage engine that simply serialises the data objects into an XML document based on the x-stream Java library [13].

4 User Interface

The management console of the iGesture framework is a Java Swing application consisting of three main parts to test gestures, define new gestures and create test sets which are represented by the `Test Bench`, `Admin` and `Test Data` tabs shown in Figure 5. The graphical user interface is based on the Model-View-Controller (MVC) design pattern [3] and can easily be extended with additional functionality if required—even without recompiling the main view—since the list of tabs to be dynamically instantiated is loaded from a property file.

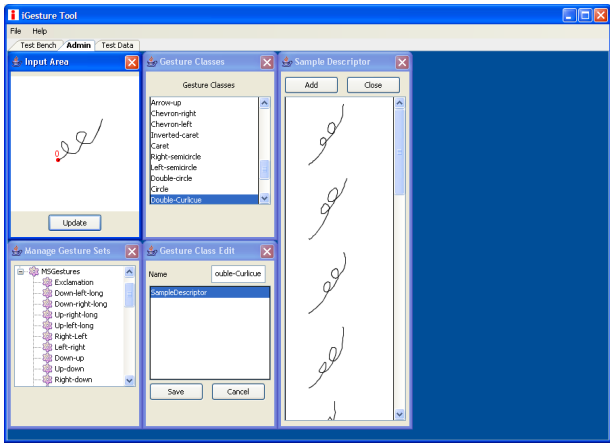

Figure 5. iGesture management console

The test bench tab provides functionality to acquire a single gesture from an input device and execute the recogniser with the gesture set and algorithm of the user's choice. This enables a simple and quick manual testing of specific gestures. All gesture sets of the currently opened persistent storage container are available and any registered algorithm may be used.

Figure 5 shows the admin tab which is used to administer gesture sets, gesture classes and the corresponding descriptors. Any new gesture captured from the input device is shown in the `Input Area` and can, for example, be added to the sample descriptor of a given gesture class. Further, it is possible to create, edit and delete new gesture classes as well as manipulate the descriptors and gesture sets. The admin tab also provides functionality to export and import complete gesture sets together with the corresponding gesture classes and their descriptors to a single XML document which later can be used to initialise the recogniser component independently of a specific storage manager.

The test data tab is used to create test sets for evaluating 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. 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 providing a mechanism to automatically run a single algorithm with different settings. A batch process is configured with an XML file specifying the configuration objects to be created. We provide different mechanisms for specifying an algorithm's parameters. It is possible to define fixed parameter values or to provide sequences, ranges and power sets a parameter has to be tested with as shown in Figure 6.

```
<?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 6. 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 gestures 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 file which can be used by the application developer to instantiate the recogniser with a given algorithm and parameter set. For the application programmer, it becomes very easy to use the gesture recognition engine as shown in Figure 7. 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.

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

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

Figure 7. Recogniser

We mainly use the digital pen and paper technology provided by the Swedish company Anoto 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 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 8. 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.

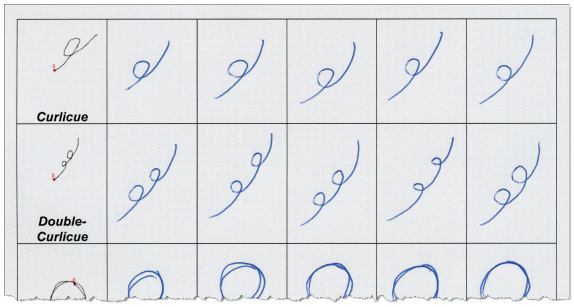

Figure 8. Interactive paper capture form

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 support the exchange of gesture sets with other applications, we further provide an import and export for the Ink Markup Language (InkML) [1].

5 Evaluation and Conclusion

So far we have implemented two existing algorithms for the iGesture framework: the Rubine algorithm [9] and the *Simple Gesture Recogniser* (SiGeR) [11] algorithm developed by Swigart. In addition, we designed and implemented two new algorithms: an extension of the Rubine algorithm, called *E-Rubine*, that introduces new features and *SiGrid* an algorithm comparing gestures based on their signatures.

We have also assembled various test sets including Microsoft Application Gestures and the Palm Graffiti alphabet and numbers, as well as some customised multi-stroke gesture sets. All the algorithms have been evaluated with the different test sets and the initial results for our E-Rubine algorithm are quite promising. An example of such an evaluation with the graffiti letters test set is shown in Table 1.

	E-Rubine	Rubine	SiGrid
Correct	342	305	297
Error	18	48	66
Reject	3	10	0
Precision	0.950	0.864	0.818
Recall	0.991	0.968	1.000
F-Measure	0.970	0.913	0.900

Table 1. Graffiti letters

The category *Correct* contains the input gestures that have been correctly recognised by the algorithm whereas the *Error* category contains incorrectly recognised gestures. A third category *Reject* contains the rejected gestures which have been returned as unclassifiable by the algorithm. In the example shown in Table 1, each gesture class was trained with 4 different users each providing 4 samples. The test set had a size of 363 samples and was produced by the same persons used for the training of the algorithm. The figures show that, in this setting, our E-Rubine algorithm clearly outperforms the original Rubine algorithm but also the simple SiGrid algorithm still provides good results. It is out of the scope of this paper to describe the two new E-Rubine and SiGrid algorithms in detail. However, more information about these algorithms as well as additional initial evaluations can be found in [10].

The iGesture framework has been helpful in implementing and testing existing algorithms as well as developing new algorithms. It was very convenient that the captured test sets could be used for any algorithm implemented, enabling the results of different algorithms to be easily compared. As mentioned earlier, the framework also simplifies the task of any application developer who wants to add some gesture recognition functionality to their application since there is a single API that works with all available algorithms. iGesture has already been successfully integrated into some of our interactive paper applications, including an interactive paper laboratory notebook application. We do not see iGesture as a replacement for existing frameworks but rather as an integration platform which should provide a single tool for designers of new recognition algorithms as well as for application developers and thereby assist the exchange of new research results. Last but not least, the iGesture framework [6] has recently been released under an Apache 2 open source license making it accessible to a broader community.

References

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

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

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

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

[5] J. I. Hong and J. A. Landay. SATIN: A Toolkit for Informal Ink-Based Applications. In *Proceedings of UIST '00, 13th Annual ACM Symposium on User Interface Software and Technology*, pages 63–72, San Diego, USA, November 2000.

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

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

[8] M. C. Norrie, B. Signer, and N. Weibel. General Framework for the Rapid Development of Interactive Paper Applications. In *Proceedings of CoPADD 2006, 1st International Workshop on Collaborating over Paper and Digital Documents*, pages 9–12, Banff, Canada, November 2006.

[9] 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.

[10] B. Signer, M. C. Norrie, and U. Kurmann. iGesture: A General Tool to Support the Development and Deployment of Pen-Based Gesture Recognition Algorithms. Technical Report 561, Department of Computer Science, ETH Zurich, 2007.

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

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

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