Developer Manual

FROG Recognizer of Gestures

Team Better Recognize
Version 1.0
April 13, 2010

Copyright ©2009-2010 Computer Science Department, Texas Christian University

Revision Sign-off

By signing the following the team member asserts that he/she has read the entire document and has, to the best of his or her knowledge, found the information contained herein to be accurate, relevant, and free of typographical error.

Name	Signature	Date
Josh Alvord		
Alex Grosso		
Jose Marquez		
Sneha Popley		
Phillip Stromberg		
Ford Wesner		

Revision History

The following is a history of revisions of this document.

Document Version	Date Edited	Changes
Version 1.0	04/13/10	Initial Draft

Table of Contents

1. Introduction to the System

1.1 What is FROG?

FROG is a gesture recognition framework coded in Java. It is unique in that all its parts are designed to be swapped out with custom components. The training and recognition process goes through many steps, and many of those steps can easily be rewritten or replaced with a developer's own process.

1.2 How FROG is Organized

FROG is split up in to two main "pipelines." One is for training, while the other is for recognition. These two pipelines are contained within and managed by the Session class. A Session is responsible for the majority of work that FROG performs on gestures.

The training pipeline is made up of three parts: Filters, Quantizer, and GestureModel. The recognition pipeline is made up of four parts: Filters, Quantizer, GestureModel and Classifier. These interfaces can be found in the *frog* package.

1.3 Before You Get Started…

1.3.1 Install the JDK

For developing in Java, a Java Development Kit, or JDK, is required. A JDK can be obtained from http://java.sun.com/javase/downloads/index.jsp. The setup process is a platform-independent, easy-to-follow wizard.

1.3.2 Download the latest FROG source

Source code for the FROG project can be found on the FROG Project CD. The source code contains all the .java source files as well as any images and sound effects used in the creation of FROG.

2. Developing a Device Plug-in

This type of plug-in will allow the use of a particular device as a source of acceleration data for the FROG system. Device plug-ins are loaded by FROG when the program is started.

2.1 Implementing the Interface

For device plug-ins to work with FROG, they will need to implement the interface: *Plugin*. *Plugin* will contain several methods that FROG will need in order to communicate effectively with your Device. Important methods in the *Plugin* interface are described in detail in the next few sections. Also consult the Javadoc for FROG on the FROG Project CD for more details.

2.1.1 Add Filter

Some devices support on-board filtering. This method allows additional filters to be added to the Device remotely. A list of filters available for a particular plug-in can be returned from the getAvailFilters method.

2.1.2 Calibrate

If a device supports calibration, this method will allow for calibrating it. The calibrate method can launch its own dialog window or message to help the user understand how to calibrate a device. Calibration is performed to eliminate factors that can affect the measurement accuracy over time. Some accelerometers, for instance, might experience reduced accuracy over time due to their handling or the constant effect of gravity on their components. An example calibration technique is to set the accelerometer to measure a standard value. In the case of acceleration, this could be achieved by laying a device flat on a surface so that an “at rest” reading can be taken. This can then be equated to the standard value of 1G since we know to expect this as the acceleration under those conditions.

2.1.3 Connect

The *connect* method formally connects to a chosen Device. Connecting to a device generally means attaching a *DeviceListener* so that incoming accelerations can be processed by a Session object.

2.1.4 Disconnect

Notify a device that it is no longer needed for acceleration input. This method should perform the necessary clean up for connections to a device and also stop anything from listening to the device.

2.1.5 Discover

Discover runs in its own thread and returns immediately. The Vector of Devices received as a parameter will be filled with nearby devices as they are detected. These Devices are not yet connected and therefore require the c*onnect* method to be called on the one to be utilized for acceleration input.

2.1.6 Get Active Filters

Returns the filters that are currently in use on the device. If there are no filters in use or the device does not support filtering, this method does nothing at all.

2.1.7 Get Available Filters

Returns the filters that are available for the device (if the device supports filtering).

2.1.8 Get Sample Rate

Returns the sample rate of the device in hertz. The sample rate is how many times per second a reading is taken from the device's accelerometers.

2.1.9 Remove Filter

Removes a filter that is in use on the device. If the device has no filters or the device does not support filtering, this method does nothing.

2.1.10 Set Sample Rate

Assigns a new sampling frequency to this device in hertz. Some devices do not support changing their sample rate in which case this method will do nothing.

2.2 The Importance of the Device Class

Aside from initializing them, FROG never interacts directly with any plug-in (classes implementing the *Plugin* interface). FROG interacts instead with an intermediary object known as the Device object. Device objects are created by plug-ins when the *Discover* method is called. Device objects have a reference back to the plug-in that created them which facilitates their own methods. Device objects are final, meaning there is no need to create your own special version of the Device class. All Devices look the same to FROG and it is therefore the responsibility of the Device to keep track of the plug-in that backs it.

2.3 Adding the Interface to FROG

FROG is designed to automatically load any classes that implement the Plugin interface in its JAR at runtime. Adding your particular plug-in will require that you add your classes to the JAR as well as any libraries they are dependent on to its classpath.

3. Developing a Filter

The *Filter* interface allows an implementing class to operate on the incoming acceleration data from a device. FROG contains two implementations: IdleState and DirectorialEquivalence. FROG has a list of available filters in addition to a chain of active filters that work on acceleration data in the order in which they were added to the chain.

3.1 Implementing the Interface

A class that implements *Filter* must contain a constructor with no parameters as FROG instantiates *Filter* objects with the *newInstance()* method. *Filter*s provide the following methods for performing work on incoming acceleration data.

3.1.1 Filter

The filter method takes an acceleration vector as its input and either returns another acceleration vector representing the result of filtering, or returns *null* in the case of a removal. In this sense, an acceleration vector should be assigned to its filtered value for replacement, as *filter* should not directly modify the parameter vector. In the case of FROG, *filter* only returns the original vector or *null* as the *Filter*s implemented deal only with removal of data (as opposed to modification). This is not, however, a general constraint on the *filter* method.

3.1.2 Setup

Given a comma-separated parameter String, changes the behavior of the *filter* method. Some filters may require no arguments; others may require many. The comma-separated values can be anything you wish but remember to document what parameters can be changed and how to change them with the String.

3.2 Adding the Interface to FROG

FROG contains an array of filters as a static field. For your filter to be useable in FROG it must be listed in that array. Once it is in the array, Session objects will be able to instantiate it and add it to their filter chains.

4. Developing a Quantizer

The *Quantizer* interface specifies a data quantizer object that operates purely on Accel3D data. FROG contains only a single implementation: Kmeans. To extend the quantizing capabilities of FROG, one must implement the interface as well as tie the *Quantizer* to a *GestureModel* and/or Session as detailed below.

4.1 Implementing the Interface

A new implementation of *Quantizer* should be assigned a unique type index to be denoted in *Quantizer* interface and accessed by *getType*. The *NUM_TYPES_SUPPORTED* field should be increased to signify the additional implementation. Note that the type indices should range exactly from 0 to the value of (*NUM_TYPES_SUPPORTED* -1). It is assumed that there is an implementation for every positive index less than *NUM_TYPES_SUPPORTED*.

4.1.1 Translate

To implement *Quantizer*, the primary task lies in implementing the *translate* method. This method receives an Accel3D and returns an appropriate integer representative, effectively translating Accel3D's into an integer alphabet. The alphabet size (total number of all potential return values) should be equivalent to the size of the quantizer, as accessed by *getSize*. In general, the *translate* method is assumed to be accessible validly at any time after the *Quantizer* object is constructed. This means that if any training (e.g. the k-means algorithm in Kmeans) of the *Quantizer* is required, it should be carried out from within the constructor or expressly noted in the documentation for the overridden *translate* method.

4.2 Adding the Interface to FROG

FROG contains one GestureModel implementation, GestureHMM, which requires quantizing to perform its duties. Thus GestureHMM could be extended to utilize quantizers other than Kmeans. Its training uses only generic calls to a *Quantizer* object, but the class itself lacks a method for training using quantizers aside from Kmeans (as no other implementations are currently supported by FROG). To implement additional *Quantizer* support in GestureHMM, additional training methods that accept appropriate parameters must be added. In a parallel fashion, new training methods should be added to Session that accept appropriate *Quantizer* parameters.

5. Developing a Gesture Model

The *GestureModel* interface specifies a three dimensional acceleration-based gesture modeling object. FROG contains only a single implementation: GestureHMM. To extend the modeling capabilities of FROG, one must implement the *GestureModel* interface as well as tie the *GestureModel* to Session as detailed below.

5.1 Implementing the Interface

A new implementation of *GestureModel* should be assigned a unique type index to be denoted in the *GestureModel* interface and accessed by *getType*. The *NUM_TYPES_SUPPORTED* field should be increased to signify the additional implementation. Note that the type indices should range exactly from 0 to the value of (*NUM_TYPES_SUPPORTED* -1). It is assumed that there is an implementation for every positive index less than *NUM_TYPES_SUPPORTED*.

In addition to a series of traditional accessor and modifier methods, the *GestureModel* interface contains specifications for several statistical tracking methods as well as the important *train* method. Only the statistical and training methods will be overviewed here. For information on all *GestureModel* methods, consult the FROG Javadoc on the FROG Project CD.

5.1.1 Statistical Methods

The *GestureModel* specification is designed so as to require any implementation to track statistical data relevant to the gesture training and recognition process. The methods *correct, incorrect, notRecognized*, and *matchedWithProbability* should be implemented so as to modify and record statistics related to recognition. For example, *correct* should at a minimum increment a counter for correct recognition events. The methods *correct, incorrect,* and *notRecognized* should correspond naturally with *getNumCorrect*, *getNumIncorrect,* and *getNumNotRecognized*. The method *matchedWithProbability* should be implemented to modify a running average recognition probability statistic (so as to interact naturally with *getAverageRecognitionProbability*).

5.1.2 Train

The FROG *GestureModel* specification assumes a modeling approach that requires some form of training to represent a gesture after instantiation. The *train* method should be implemented to train the model based on a gesture set provided in construction or over time. As *train* requires no parameters, any implementation of *GestureModel* should receive all parameters in construction, utilize modifier methods to set parameters, make use of default parameters, include additional training methods that require additional parameters, or any combination of the above. In addition, the *getProbability* method is not assumed to return valid data until training has occurred.

5.2 Adding the Interface to FROG

FROG deals with *GestureModel* objects within the Session class. The session class has its own *train* method(s) that are responsible for training individual *GestureModel* objects. The *train()* and *train(int index)* methods are set up using a switch statement on the current model type of the Session. Currently, however, as there is only a single *GestureModel* implementation, GestureHMM, there is only a single case in this block. This is the case of a model type corresponding to GestureHMM. To insert a new *GestureModel* implementation, simply add an additional case statement corresponding to its particular type index within these training methods that performs training on the current or specified *GestureModel* object.

If, however, the implementation requires additional parameters for construction or training, additional training methods can be implemented to accommodate. The additional training methods for GestureHMM can be used as an example for how to construct these. Training methods focused on a specific implementation of *GestureModel* must validate that the Session is currently using the correct model type and report training failure if this is not the case.

6. Developing a Classifier

The *Classifier* interface specifies a *GestureModel* classification object. A *Classifier* requires a set of *GestureModel* objects with which to perform classification.

6.1 Implementing the Interface

A new implementation of *Classifier* should be assigned a unique type index to be denoted in *Classifier* interface and accessed by *getType*. The *NUM_TYPES_SUPPORTED* field should be increased to signify the additional implementation. Note that the type indices should range exactly from 0 to the value of (*NUM_TYPES_SUPPORTED* - 1). It is assumed that there is an implementation for every positive index less than *NUM_TYPES_SUPPORTED*.

Classifier presents a fairly easy situation for implementation. The method *classify* should return the *GestureModel* recognized for a given gesture instance. The method *getLastProbability* should return the recognition probability of the last classification event. The accessor *getModels* should return the set of *GestureModel*s contained in the *Classifier*.

6.2 Adding the Interface to FROG

FROG deals with *Classifier* objects within the Session class. Adding an additional *Classifier* implementation to FROG is as simple as modifying the private *createClassifier* method in Session. Similarly to the *GestureModel* situation, this method functions using a switch statement on the current classifier type of the Session. For each new *Classifier* implementation, a new case statement for its particular type index should be added. The case statement should assign the *classifier* variable to a *Classifier* object of type corresponding to the type index.