GART: The Gesture and Activity Recognition Toolkit

Kent Lyons, Helene Brashear, Tracy Westeyn,
Jung Soo Kim, and Thad Starner

College of Computing and GVU Center
Georgia Institute of Technology
Atlanta, GA 30332-0280 USA
{kent, brashear, turtle, jszzang, thad}@cc.gatech.edu

Abstract. The **G**esture and **A**ctivity **R**ecognition **T**oolit (GART) is a user interface toolkit designed to enable the development of gesture-based applications. GART provides an abstraction to machine learning algorithms suitable for modeling and recognizing different types of gestures. The toolkit also provides support for the data collection and the training process. In this paper, we present GART and its machine learning abstractions. Furthermore, we detail the components of the toolkit and present two example gesture recognition applications.

Key words: Gesture recognition, user interface toolkit

1 Introduction

Gestures are a natural part of our everyday life. As we move about and interact with the world we use body language and gestures to help us communicate, and we perform gestures with physical artifacts around us. Using similar motions to provide input to a computer is an interesting area for exploration. Gesture systems allow a user to employ movements of her hand, arm or other parts of her body to control computational objects.

While potentially a rich area for novel and natural interaction techniques, building gesture recognition systems can be very difficult. In particular, a programmer must be a good application developer, understand the issues surrounding the design and implementation of user interface systems and be knowledgeable about machine learning techniques. While there are high level tools to support building user interface applications, there is relatively little support for a programmer to build a gesture system. To create such an application, a developer must build components to interact with sensors, provide mechanisms to save and parse that data, build a system capable of interpreting the sensor data as gestures, and finally interpret and utilize the results.

One of the most difficult challenges is turning the raw data into something meaningful. For example, imagine a programmer who wants to add a small gesture control system to his stylus–based application. How would he transform the sequence of mouse events generated by the UI toolkit into gestures?

Most likely, the programmer would use his domain knowledge to develop a (complex) set of rules and heuristics to classify the stylus movement. As he further developed the gesture system, this set of rules would likely become increasing complex and unmanageable. A better solution would be to use some machine learning techniques to classify the stylus gestures. Unfortunately doing so requires extensive domain knowledge about machine learning algorithms.

In this paper we present the **G**esture and **A**ctivity **R**ecognition **T**oolkit (GART), a user interface toolkit designed to abstract away many machine learning details so an application programmer can build gesture recognition based interfaces. Our goal is to allow the programmer access to powerful machine learning techniques without requiring her to become an expert in machine learning. In doing so we hope to bridge the gap between the state of the art in machine learning and user interface development.

2 Related Work

Gestures are being used in a large variety of user interfaces. Gesture recognition has been used for text input on many pen based systems. ParcTab's Unistroke [8] and Palm's Graffiti are two early examples of gesture based text entry systems for recognizing handwritten characters on PDAs. EdgeWrite is a more recent gesture based text entry method that reduces the amount of dexterity needed to create the gesture [11]. In Shark2, Kristensson and Zhai explored adding gesture recognition to soft keyboards [4]. The user enters text by drawing through each key in the word on the soft keyboard and the system can recognize the pattern formed by the trajectory of the stylus through each letter. Hinckley et al. augmented a hand–held with several sensors to detect different types of interaction with the device (recognizing when it is in position to take a voice note, powering on when it is picked up, etc) [3]. Another use of gesture is as an interaction technique for large wall or tabletop surfaces. Several systems utilize hand (or finger) posture and gestures [5, 12]. Grossman et al. also used multi-finger gestures to interact with a 3D volumetric display [2].

From a high level, the basics of using a machine learning algorithm for gesture recognition is rather straightforward. To create a machine learning model, one needs to collect a set of data and provide descriptive labels for it. This process is then repeated many times for each gesture and then repeated again for all of the different gestures to be recognized. The data is used by a machine learning algorithm and is modeled via the "training" process. To use the recognition system in an application, data is again collected. It is then sent through the machine learning algorithms using the models trained above and the label of the model most closely matching the data is returned as the recognized value.

While conceptually this is a rather simple process, in practice it is unfortunately much more difficult. For example, there are many details in implementing most machine learning algorithms (such as dealing with limited precision), many of which may not be covered in machine learning texts. A developer might use one a machine learning software package created to encapsulate a variety of

algorithms such as Weka [1] or Matlab. An early predecessor to this work, the Georgia Tech Gesture Toolkit (GT^2k), was designed in a similar vein [9]. It was designed around Cambridge University's speech recognition toolkit (CU–HTK) [13] to facilitate building gesture based applications. Unfortunately, GT^2k requires the programmer to have extensive knowledge about the underlying machine learning mechanisms and leaves several tasks such as the collection and management of the data to the programmer.

3 GART

The Gesture and Activity Recognition Toolkit (GART) is a user interface toolkit. It is designed to provide a high level interface to the machine learning process facilitating the building of gesture recognition applications. The toolkit consists of an abstract interface to the machine learning algorithms (training and recognition), several example sensors and a library for samples.

To build a gesture based application using GART, the programmer first selects the sensor she will use to capture information about the gesture. We currently support three basic sensors in our toolkit: a mouse (or pointing device), a set of Bluetooth accelerometers, and a camera sensor. Once a sensor is selected, the programmer builds an application that can be used to collect training data. This program can be either a special mode in the final application being built, or an application tailored just for data collection. Finally, the programmer instantiates the base classes from the toolkit (encapsulating the machine learning algorithms, and library) and sets up the callbacks between them for data collection or recognition. The remainder of the programmer's coding effort can then be devoted to building the actual application of interest and using the gesture recognition results as desired.

3.1 Toolkit Architecture

The toolkit is composed of three main components: *Sensors*, *Library*, and *Machine Learning*. *Sensors* collect data from hardware and may provide post–processing. The *Library* stores the data and provides a portable format for sharing data sets. The *Machine Learning* component encapsulates the training and recognition algorithms. Data is passed from the sensor and machine learning components to other objects through callbacks. The flow of data through the system for data collection involves the above three toolkit components and the application (Figure 1). A sensor object collects data from the physical sensors and distributes it. The sensor will likely send raw data to the application for visualization as streaming video, graphs, or for other displays. The sensor also bundles a set of data with its labeling information into a sample. The sample is sent to the library where it stored for later use. Finally, the machine learning component can pull data from the library and use it to train the models for recognition. Figure 2 shows the data flow for a recognition application. As before, the sensor can send raw data to the application for visualization or user

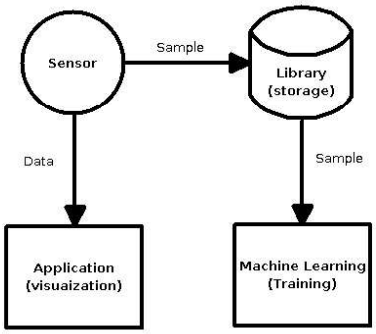

Fig. 1. Data collection

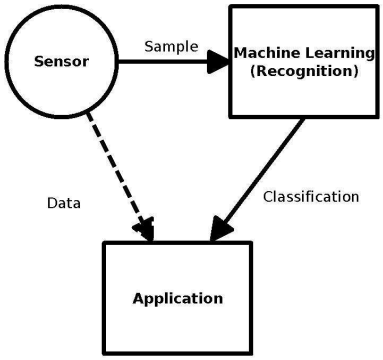

Fig. 2. Gesture recognition

feedback. The sensor also sends samples to the machine learning component for recognition, and recognition results are sent to the application.

Sensors Sensors are components that interface with the hardware, collect data, and may provide parsing or post–processing of the data. Sensors are also designed around an event–based architecture that allows them to notify any listeners of available data. The sensor architecture allows for both synchronous or asynchronous reading of sensors.

Our toolkit sensors support sending data to listeners in two formats: samples and plain data. Samples are well defined sets of data that represents gestures. A sample can also contain meta information such as gesture labels, a user name, time stamps, notes, etc. Through a callback, sensors send samples to other toolkit components for storage, training, or recognition. The toolkit has been designed for extensibility particularly with respect to available sensors. Programmers can generate new sensors by inheriting from the base sensor class. This class provides event handling for interaction with the toolkit. The programmer can then implement the sensor driver and any necessary post–processing. The toolkit supports event based sensors as well as polled sensors and it streamlines data passing through standard callbacks. Three sensors are provided with the toolkit:

- **Mouse:** The Mouse sensor provides an abstraction for using the mouse as the input device for gestures The toolkit provides three implementations of the mouse sensor. MouseDragDeltaSensor generates samples which are composed of Δx and Δy from the last mouse position. The MouseDragVectorSensor generates sample which consists of the same information in polar coordinates (θ and radius from the previous point). Finally, MouseMoveSensor is similar to the vector drag sensor, but does not segment the data using mouse clicks.
- **Camera:** The SimpleImage sensor is a simple camera sensor which reads input from a USB camera. The sensor provides post–processing that tracks

an object based on a color histogram. This sensor produces samples that are composed of the (x, y) position of the object in the image over time.

– **Accelerometers:** Accelerometers are devices which measure static and dynamic acceleration and can be used to detect motion. Our accelerometer sensor interfaces with small wearable 3 axis Bluetooth accelerometers we have created [10]. The accelerometer sensor provides synchronization of the data from multiple sensors and generates a sample of Δx, Δy, and Δz indicating changes in acceleration for each axis.

Library The library component in the toolkit is responsible for storing and organizing data. This component is not found in most machine learning libraries but is a critical portion of a real application. The library is composed of a collection of samples created by a data collection application. The machine learning component then uses the library during training as the source of labeled gestures. The library also provides methods to store samples in an XML file.

Machine Learning The machine learning component provides the toolkit's abstraction for the machine learning algorithms and is used for modeling data samples (training) and recognizing gesture samples. During training, it loads samples from a given library, trains the models, and returns the results of training. For recognition, the sensor sends samples to the machine learning object which in turn sends a result to all of its listeners (the application). A result is either the label of the classified gesture or any errors that might have occurred. One of the main goals of the toolkit was to abstract away as many of the machine learning aspects of gesture recognition as possible. We have also provided defaults for much of the machine learning process. However, at the core of the system are hidden Markov models (HMMs) which we currently use to model the gestures. There has been much research supporting the use of HMMs to recognize time series data such as speech, handwriting and gesture recognition. [7, 6, 10].

The HMMs in GART are provided by CU-HTK [13]. Our HTK class wraps this software which provides an extensive framework for training and using hidden Markov models (HMMs), as well as a grammar based infrastructure. GART provides the high level abstraction of our machine learning component and integration into the rest of the toolkit. We also have an options object which keeps track of the necessary machine learning configuration information such as the list of gesture to be recognized, HMM topologies, and models generated by the training process.

While the toolkit currently uses hidden Markov models for recognition, the abstraction of machine learning component allows for expansion. These expansions could include other popular techniques such as neural networks, decision trees or support vector machines. An excellent candidate for this expansion would be the Weka machine learning library, which includes implementations for a variety of different algorithms [1].

3.2 Code Samples

The basics of setting up a new application using the toolkit components described above requires relatively little code. To set up a new gesture application the programmer needs to create a set of options (using the defaults provided by the toolkit) and a library object. The programmer then initializes the machine learning component, HTK, with the options. Finally a new sensor is created.

```
Options myOpts=new GARTOptions();
Library myLib= myOpts.getLibrary();
HTK htk=new HTK(options);
Sensor=new MySensor();
```

For data collection, the programmer needs to connect the sensor to the library so it can save the samples.

```
sensor.addSensorSampleListener(library);
```

Finally for recognition, the programmer configures the sensor to send samples to the HTK object for recognition. The recognition results are then sent back to the application for use in the program.

```
sensor.addSensorSampleListener(htk);
htk.addResultListener(myApplication);
```

The application may also want to listen to the sensor data to provide some user feedback about the gesture as it is happening (such as a graph of the gesture).

```
sensor.addSensorDataListener(myApplication);
```

Finally, the application may need to provide some configuration information for the sensor on initialization and it may need to segment the data by calling startSample() and stopSample() on the sensor.

GART was developed using the Java JDK 5.0 from Sun Microsystems. It has been tested in the Linux, Mac OS X, and Windows environments. The core GART system requires CU–HTK, free software that may be used to develop applications, but not sold as part of a system.

4 Sample Applications

We have built several different gesture recognition applications using our toolkit. Our first set of applications demonstrate the capabilities of each sensor in the toolkit, and here we will discuss the WritingPad application. Virtual Hopscotch is more fully featured and was built by a student in our lab that had no direct experience with the development of GART.

The WritingPad is an application that uses our mouse sensor. It allows a user to draw a gesture with a mouse (or stylus) and have it recognized by the system. To create a gesture, the user depresses the mouse button, draws the intended shape, and releases the mouse button. This simple system uses the toolkit to recognize a few different handwritten characters and some basic shapes.

The application is composed of three objects. The first object is the main WritingPad application which initializes the program, instantiates the needed GART objects (MouseDragVectorSensor, Library, Options and and HTK) and connects these for training as described in Section 3.2. This object also creates

the main application window and populates it with the UI components (Figure 3). At the top is an area for the programmer to control the toolkit parameters needed to create new gestures. In a more fully featured application, this functionality would either be in a separate program or hidden in a debug mode. On the left is an area used to label new gestures. Next, there is a button to save the library of samples and another button to train the model. Finally at the top right, there is a toggle button that changes the application state between data collection and recognition modes. The change in modes is accomplished by calling a method in the main WritingPad object which alters the *sensor* and *result* callbacks as described above (Section 3.2). In recognition mode, this object receives the results from the machine learning component and opens a dialog box with the label of the recognized gesture (Figure 3). A more realistic application would act upon the gesture to perform some other action. Finally, the majority of the application window is filled with a CoordinateArea, a custom widget that displays on-screen user feedback. This application demonstrates the basic components needed to use mouse gestures.

The Virtual Hopscotch application is a gesture based game inspired by the traditional children's game, Hopscotch. This game was developed over the course of a weekend by a student in our lab who had no prior experience with the toolkit. We gave him instructions to create a game using two accelerometers and our applications that demonstrate the use of the different sensors. From there, he designed and implemented the game.

The Virtual Hopscotch game consists of a scrolling screen with squares displayed to indicate how to hop (Figure 4). The player wears our accelerometers on her ankles and follows the game making different steps or jumps (right foot hop, left foot hop, and jump with both feet). As the squares scrolls into the central rectangle, the application starts sampling and the player performs her hop gesture. If the gesture is recognized as correct, the square changes color as it scrolls off the screen and the player wins points. Figure 4 show the game in action. The blue square in the center is the indication that the player should stomp on her left foot. The two squares just starting to show at the top of the screen are the next move to be made, in this case jumping with both feet.

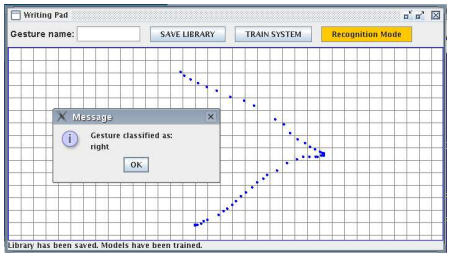

Fig. 3. The WritingPad application showing the recognition of the "right" gesture.

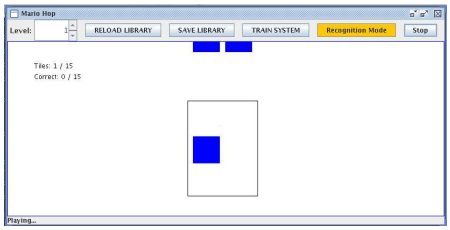

Fig. 4. The Virtual Hopscotch game based on accelerometer sensors.

For Writing pad, the majority of application code (approximately 300 lines) is devoted to the user interface. In contrast, only a few dozen lines are devoted to gesture recognition. Similarly, Virtual Hopscotch has a total of 878 lines of code and again, most of which are associated with the user interface. Additional code was also created to manage the game infrastructure. Of the six classes created, three are for maintaining game state. The other three have direct correspondence to the WritingPad example. There is one class for the application proper, one for the main window and one for the game visualization.

5 Discussion

Throughout the development of GART, we have attempted to provide a simple interface to gesture recognition algorithms. We have distilled the complex process of implementing machine learning algorithms down the essence of collecting data, providing a method to train the models, and obtaining recognition results. Another important feature of the toolkit is the components that support data acquisition with the sensors, sample management in the library, and simple callbacks to route the data. These are components required to build gesture recognition applications often not provided by other systems. Together, these components enable a programmer to focus on application development instead of the gesture recognition system.

We have also designed the toolkit to be flexible and extensible. This aspect is most visible in the sensors. We have created several sensors that all have the same interface to an application and the rest of the toolkit. A developer can swap mouse sensors (which provide different types of post–processing) by changing only a few lines of code. Changing to a dramatically different type of sensor requires minimal modifications. In building the Virtual Hopscotch game, our developer started with a mouse sensor and used mouse based gestures to understand the issues with data segmentation and to facilitate application development. After creating the basics of the game, he then switched to the accelerometer sensor. While we currently have only one implementation of a machine learning back-end (the CU-HTK), our interface would remain the same if we had different underlying algorithms.

While we have abstracted away many of the underlying machine learning concepts, there are still some issues the developer needs to consider. Two such issues are data segmentation and sensor selection. Data segmentation involves denoting the start and stop of a gesture. This process can occur as an internal function of the sensor or as a result of signals from the application. Application signals can be from either user actions such as a button press or from the application structure itself. The MouseDragSensor uses internal functions to segment its data. The mouse pressed event starts the collection of a sample, and the mouse released function completes the sample and sends it to its listeners. Our camera sensor uses a signal generated by a button press in the application to segment its data. In Virtual Hopscotch, the application uses timing events

corresponding to when the proper user interface elements are displayed on-screen to segment the accelerometer data.

In addition to segmentation, a key component in designing a gesture-based application is choosing the appropriate data to sense. This process includes selecting a physical sensor that can sense the intended activities as well as selecting the right post–processing to turn the raw data into samples. The data from one sensor can be interpreted in many ways. Cameras, for example, have a myriad of algorithms devoted to the classification of image content. For an application that uses mouse gestures, change in location $(\Delta x, \Delta y)$ is likely a more appropriate feature vector than absolute position (x, y). By using relative position, the same gesture can be composed in different locations.

We have designed GART to be extensible and much of our future work will be expanding the toolkit in various ways. We are interested in building an example "sensor fusion" module to provide infrastructure for easily combining multiple sensors of different types (i.e. cameras and accelerometers). We would also like to abstract out the data post–processing to allow greater code reuse between similar sensors. As previously mentioned, the machine learning back end is designed to be modular and to allow different algorithms to "plug in". Finally, we are interested in extending the toolkit to make use of continuous gesture recognition. Right now each gesture must be segmented by the user, the application, or using some knowledge about the sensor itself. While quite powerful, other applications would be enabled by adding a continuous recognition capability.

6 Conclusions

Our goal in creating GART was to provide a toolkit to simplify the development process involved in creating gesture-based applications. We have created a high-level abstraction of the machine learning process whereby the application developer selects a sensor and collects example gestures to use for training models. To use the gestures in an application, the programmer connects the same sensor to the recognition portion of our toolkit which in turn sends back classified gestures. The machine learning algorithms, associated configuration parameters and data management mechanisms are provided by the toolkit.

By using such a design, we allow a developer the ability to create gesture recognition systems without first needing to become experts in machine learning techniques. Furthermore, by encapsulating the gesture recognition, we reduce the burden of managing all of the associated data and models to build a gesture recognition system. Our intention is that GART will provide a platform to allow further exploration of gesture recognition as an interaction technique.

7 Acknowledgments

We want to give special thanks to Nirmal Patel for building the Virtual Hopscotch game. This material is supported, in part, by the Electronics and Telecommunications Research Institute (ETRI).

References

1. E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg. Weka - a machine learning workbench for data mining. In O. Maimon and L. Rokach, editors, *The Data Mining and Knowledge Discovery Handbook*, pages 1305–1314. Springer, 2005.
2. T. Grossman, D. Wigdor, and R. Balakrishnan. Multi-finger gestural interaction with 3d volumetric displays. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 61–70. ACM Press, 2004.
3. K. Hinckley, J. Pierce, M. Sinclair, and E. Horvitz. Sensing techniques for mobile interaction. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 91–100. ACM Press, 2000.
4. P. O. Kristensson and S. Zhai. Shark2: a large vocabulary shorthand writing system for pen-based computers. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 43–52. ACM Press, 2004.
5. S. Malik, A. Ranjan, and R. Balakrishnan. Interacting with large displays from a distance with vision-tracked multi-finger gestural input. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 43–52. ACM Press, 2005.
6. T. Starner, J. Weaver, and A. Pentland. Real-time American Sign Language recognition using desk and wearable computer-based video. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 20(12), December 1998.
7. C. Vogler and D. Metaxas. ASL recognition based on a coupling between HMMs and 3D motion analysis. In *ICCV*, Bombay, 1998.
8. R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications*, 2(6):28–33, Dec 1995.
9. T. Westeyn, H. Brashear, A. Atrash, and T. Starner. Georgia tech gesture toolkit: supporting experiments in gesture recognition. In *Proceedings of the 5th International Conference on Multimodal Interfaces, (ICMI 2003)*, pages 85–92. ACM, November 5-7 2003.
10. T. Westeyn, K. Vadas, X. Bian, T. Starner, and G. D. Abowd. Recognizing mimicked autistic self-stimulatory behaviors using hmms. In *Ninth IEEE International Symposium on Wearable Computers (ISWC 2005)*, pages 164–169. IEEE Computer Society, October 2005.
11. J. O. Wobbrock, B. A. Myers, and J. A. Kembel. Edgewrite: a stylus-based text entry method designed for high accuracy and stability of motion. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 61–70. ACM Press, 2003.
12. M. Wu and R. Balakrishnan. Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 193–202. ACM Press, 2003.
13. S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. *The HTK Book (for HTK Version 3.3)*. Cambridge University Engineering Department, 2005.