 **Tutorial on Android Sensor &&** **Touch** **Movements Description** 

**EECS397/EECS600, Mobile Computing && Sensor Network, Fall 2015**

**Advisor: Prof. Ming-Chun Huang ( Due by 9/14/2015 12pm )**

**Introduction:**

In this project, we will learn how to retrieve multiple sensor data from Android phone simultaneously, and how to describe touch movements in terms of an action code and a set of axis values while unlocking a pattern. Pattern lock is one of the greatest way to lock your smart phone and prevent any unwanted key press. However, Smartphones are like mini-computers, but because of that, they are at risk of being hacked and the security problem is always there. Sensors are used in nearly every phone to give it the videogame-like ability to respond to tilt and motion. But mounting research has indicated that it also can be used for all forms of surveillance. Of course, to do this hackers would need to figure out a way to remotely access the sensors and access its data. With these sensor data and touch movement description, hackers can crack your lock pattern easily and get into your smartphone. This means it is necessary for us to know the potential risk and learn the related sensor and motion framework.

Figure 1

Figure 2

Figure 1 depicts a possible scheme for pattern lock. Through our tutorial, you should be able to get your random pattern by clicking the button. If you click “practice button”, you will enter into the practice mode and disable the “Generate Pattern” button. You can try to unlock the pattern in practice mode. Figure 2 is the data output sample. You need to log 27 kinds of attributes to a CSV file. The first column is the time stamp and the last column is used to keep track of which lines came from which patterns. The data will be saved to the file only when you enter the correct pattern. Otherwise, the useless data will be thrown away.

We will be using the above example throughout our tutorial. The tutorial is divided into three parts. The first part of the tutorial will explain a little bit of lock pattern on an Android smart phone. And you need to figure out how to generate pattern randomly. Then, we will introduce how to trace your finger using motion event in the second part. The third part will focus on how to get the sensor data from build-in sensor in smart phone. We will provide an overview of the sensors that are available on the Android platform and an introduction to the sensor framework.

Disclaimer: This tutorial is for educational purpose only and it is not a 100% step by step tutorial, which means students still need to finish some parts by themselves. It is intended for students who would like to quickly overview the sensor and motion events framework in Android. Without considering security and reliability, according to this tutorial, students can build a fast-working prototype for their class project.

***Pre-requisite:***

1. EECS 132 and EECS 233

2. Android Studio (recommended).

Report: There are four practices in this tutorial. Each team should submit ONE four-page report. In the report, you need to specify what your contribution is and what your partner’s contribution is in this project. Then, you need to explain your code in detail. In addition, you need to provide some screenshots of the final result, including the lock pattern and the data files. Finally, you need zip your source code and report in a zip file, and submit the zip file onto the blackboard. We will check your source code and your report to assign you a grade.

***Part I: Lock Pattern***

We have six different classes in the implementation of generating lock pattern and practicing unlock pattern. *Defaults class, EmergencyExit class, LockPatternView class, NodeDrawable class, PatternGenerator class* and *Point class.*

*Defaults class* stores some essential variable value. Such as, grid length, minimum pattern length, maximum pattern length.

*EmergencyExit class* is used to exit some process. If just building the drawables (draw the circle) is taking too long, bail!

A lock pattern is actually a list of Points and *Point class* holds two integer coordinates. For example, the following lock pattern (Fig.3) is [Point(1,1), Point(0,2), Point(1,2), Point(2,2)]. Note that x represents column and y represents row.

Figure 3

*NodeDrawable class* build shape of each circle node in the grid and draw the arrow exit indicator for each node in the pattern. Each node has three layer of circle (outer, middle, inner) with different color. In addition, there are 5 different state for each node, representing unselected (custom), selected, highlighted, correct, incorrect, respectively.

*LockPatternView class* extends *View class*. This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components.

To implement a custom view, you will usually begin by providing overrides for some of the standard methods that the framework calls on all views. You do not need to override all of these methods. The most important method in *View* is *onDraw(). onDraw()* is called when the view should render its content. Note that the parameter of *onDraw()* is Canvas. The Canvas class holds the "draw" calls. To draw something, you need 4 basic components: A Bitmap to hold the pixels, a Canvas to host the draw calls (writing into the bitmap), a drawing primitive (e.g. Rect, Path, text, Bitmap), and a paint (to describe the colors and styles for the drawing).

We call Canvas.drawLine() to connect the correct node in a specific lock pattern. mNodeDrawable[][] is actually a 3\*3 matrix, storing the 9 node.

*onTouchEvent()* is another very important method for *lockPatternView*. When your finger move on the screen in practice mode, you try to unlock the lock pattern by drawing a pattern. Actually, it is achieved by appending one node to the tail node for each time. After you move your finger up, *testPracticePattern()* method will test your pattern and tell you the result.

In this tutorial, we will introduce *onTouchEvent()* in detail in **Part II (Motion Event)**.

*PatternGenerator class* is used to generate the pattern randomly.

The most important method in the *PatternGenerator class* is *getPattern(),* whichgenerates the pattern randomly according to the pre-defined parameter, such as pattern maximum length and minimum length. In our tutorial, we use 3\*3 grids and the grid length is 3. An instance of *Random* *class* is used to generate a stream of pseudorandom numbers. *Random.nextInt(int n)* returns a pseudorandom, uniformly distributed *int* value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence. You can refer the following link to learn how to use Random() in detail.

<http://docs.oracle.com/javase/7/docs/api/java/util/Random.html>

You can refer the following link to learn how to use List() in detail.

<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

**Homework 1: Using Random() and List(), write your own getPattern(). Once you call the getPattern(), you should randomly get a new pattern. The length of the new pattern is between the minPathLength and maxPathLength.**

**public static List<Point> getPattern() {**

**List<Point> pattern = new ArrayList<>();**

**…….**

**…….**

**return pattern;**

**}(**

**Hint 1: If you randomly get a start point, how can you find the next point?**

**1. Get the initial candidatePointList = allAvailablePointList – start point.3**

**2. Go through the whole candidatePointList and remove this point if there is other unused point between this one and the start point. For example, the following situation is not allowed. There is an unused point between Point(1, 0) and Point(1, 2). So, if Point(1, 0) is your start point, Point(1, 2) cannot be your candidate next point.**

The left pattern is wrong!

**Hint 2: How to know whether there is an unused point between candidate point and start point?**

**1. Calculate the delta.x = candidatePoint.x – startPoint.x and delta.y = candidatePoint.y – startPoint.y**

**2. Compute the greatest common division (gcd) of delta.x and delta.y.**

**3. If gcd>1, do the iteration for each j, 1<=j<gcd.**

**Unused.x = start.x + delta.x/gcd\*j**

**Unused.y = start.y + delta.y/gcd\*j**

**4. If unused point exists, this candidate point should be removed from the candidate point list.**

**After you finish the homework 1, you should be able to generate lock pattern randomly when you click the button. You can design your personal user interface. For example, the position of the button, size, color, etc. Your “Generate Pattern Randomly” button should call the function of *getPattern().* After you click the “Generate Pattern Randomly” button, a random lock pattern should be generated. After you click the “Practice” button, the app will enter into practice mode and the “Generate Pattern Randomly” button is disabled. In practice mode, you should able to unlock the current pattern. You don’t need to change other class files. The only two things you need to do are write the *getPattern()* by yourself and design UI (enable button).**

***Part II: Motion Event Overview***

Motion events describe movements in terms of an action code and a set of axis values. The action code specifies the state change that occurred such as a pointer going down or up. The axis values describe the position and other movement properties.

For example, when the user first touches the screen, the system delivers a touch event to the appropriate [View](http://developer.android.com/reference/android/view/View.html) with the action code [ACTION\_DOWN](http://developer.android.com/reference/android/view/MotionEvent.html" \l "ACTION_DOWN) and a set of axis values that include the X and Y coordinates of the touch and information about the pressure, size and orientation of the contact area.

When your finger first touch the screen, you will get Action\_Down event which means a pressed gesture has started, the motion contains the initial starting location. Action\_Move means a change has happened during a press gesture (between Action\_Down and Action\_Up). Action\_Up means a pressed gesture has finished, the motion contains the final release location as well as any intermediate points since the last down or move event.

**Homework 2: When your finger touch the screen, you should save the coordinate (x,y) of touch point. In addition, you also need to save the pressure and size of the contact area. When your finger move on the screen like pattern lock, you should also save the velocity of your finger in x and y direction. The data format should be same with Fig.2. For each column, the attribute is position\_X, position\_Y, velocity\_X, velocity\_Y, pressure, size. Design your motion event and save the finger position, velocity, pressure, touch size data into a .CSV file in** the DCIM folder in SD Card.

***Part III: Sensor Sensing***

Although this tutorial will help you to dig into related knowledge quickly, we highly recommend you to go through the following two links by yourself.

<http://developer.android.com/intl/ru/guide/topics/sensors/sensors_overview.html> Sensors Overview

<http://developer.android.com/intl/ru/reference/android/view/MotionEvent.html> Motion Event

After all, self-learning ability is the basic requirement of a college student.

***Sensors Framework***

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the [android.hardware](http://developer.android.com/reference/android/hardware/package-summary.html) package and includes the following classes and interfaces:

[SensorManager](http://developer.android.com/reference/android/hardware/SensorManager.html)

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

[Sensor](http://developer.android.com/reference/android/hardware/Sensor.html)

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

[SensorEvent](http://developer.android.com/reference/android/hardware/SensorEvent.html)

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

[SensorEventListener](http://developer.android.com/reference/android/hardware/SensorEventListener.html)

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

**Now, let’s get started with the implementation in an actual Android application.**

First, we need to set up our Android app with the required members, get the SensorManager and initialize our sensor listeners, for example, in the *onCreate* method:

Notice that the application implements the SensorEventListener interface. So we’ll have to implement the two methods onAccuracyChanged and onSensorChanged. We will leave onAccuracyChanged empty since it is not necessary for this tutorial. The more important function is onSensorChanged. It updates our sensor data continuously.

We use the onResume and onPause callback methods to register and unregister the sensor event listener. As a best practice you should always disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours because some sensors have substantial power requirements and can use up battery power quickly. Register the sensor event listener means initializing the sensor listeners.

After the listeners are initialized, the onSensorChanged() method is called automatically whenever new sensor data is available. The data is then copied or processed, respectively.

Homework 3: Get the following sensor data from your smart phone and save them into a .CSV file in the DCIM folder in SD Card.

Sensor: Accelerometer, Magnetic Field, Gyroscope, Rotation Vector, Linear Acceleration, Gravity.

Note that: You need to save the data in exactly the same data format with Fig.2. For each column, the attribute is TimeStamp, TYPE\_ACCELEROMETER\_X, TYPE\_ACCELEROMETER\_Y, TYPE\_ACCELEROMETER\_Z, TYPE\_MAGNETIC\_FIELD\_X, TYPE\_MAGNETIC\_FIELD\_Y, TYPE\_MAGNETIC\_FIELD\_Z, TYPE\_GYROSCOPE\_X, TYPE\_GYROSCOPE\_Y, TYPE\_GYROSCOPE\_Z, TYPE\_ROTATION\_VECTOR\_X, TYPE\_ROTATION\_VECTOR\_Y, TYPE\_ROTATION\_VECTOR\_Z, TYPE\_LINEAR\_ACCELERATION\_X, TYPE\_LINEAR\_ACCELERATION\_Y, TYPE\_LINEAR\_ACCELERATION\_Z, TYPE\_GRAVITY\_X, TYPE\_GRAVITY\_Y, TYPE\_GRAVITY\_Z.

Each of these sensor type generate three dimensional data (X,Y,Z).

Homework 4:   
In homework 1, you generated a random unlock pattern and practiced unlocking the phone.  In homework 2, you logged the finger motion data (from *onTouchEvent()*) to a CSV file.  In homework 3, you logged the related sensor data to a CSV file.  Now, for homework 4, look at homework 1 again. When the practice mode is enabled and user’s finger touches the screen, you should log all the data in homework 2 and 3 to a CSV file line by line. However, note that when the user's finger touches down and moves on the screen, instead of the lines being appended directly to a file, they should instead be stored in a temporary buffer.  When the user's finger lifts up from the screen, the pattern should be tested to see whether it is correct or not.  If the entered pattern is correct, then the data in the buffer are good and should be written to the CSV file, using the "Counter" field to keep track of which lines came from which patterns (number should be same for all lines from each entered pattern).  Otherwise, if the entered pattern is incorrect, the data should be discarded. mCurrentPattern field is used to record the current pattern. Figure 2 gives you an overview of the final result.

