MOTION GESTURES

the next wave of interaction

API Guide
for
Gesture Recognition Engine

Version 1.3

Table of Contents

Gesture Recognition API

The Gesture Recognition API is designed for the cloud-based Gesture Recognition Engine (GRE). It allows a developer to send output of sensor or sensors to the GRE and receive gesture recognition results, thereby enabling deployment of a gestures-based user interface for any gadget, device, or application. The system currently supports gestures based on motion and touch sensors. Support for other sensors – such as vision sensors – will be added shortly.

NOTE: When working with motion (ie. IMU) sensors, it is strongly recommended to use a device containing both accelerometer and gyroscope for higher recognition accuracy.

API URI

The URI exposed for gesture recognition applications is a secured WebSocket endpoint:

wss://sdk.motiongestures.com/recognition

Communication over an insecure connection is not supported at this point.

Communication Protocol

Communication between client and the server is performed using Google's Protocol Buffers: https://developers.google.com/protocol-buffers/

The Gesture Recognition Engine's .proto file can be obtained from: https://sdk.motiongestures.com/protobuf/greapi.proto

Since the API is based on the WebSocket standard and the widely available Protobuf library, it is easy to write and test applications in your preferred language and on your preferred platform.

Getting Started

Before you can start writing and testing gesture recognition applications, you must register an account with Motion Gestures at https://sdk.motiongestures.com, create a project, and add gestures to that project. For more information on how to do this as well as to familiarize yourself with the capabilities of the Gesture Recognition Engine, please consult the **User Guide** (available on our website).

You may also need the following libraries:

Protobuf

Obtain the .proto file https://sdk.motiongestures.com/protobuf/greapi.proto and install the protobuf library and compiler from https://developers.google.com/protocol-buffers/. Using the protobuf compiler (protoc), generate the Protobuf classes from the greapi.proto file.

For example, to generate Java classes, you can run:
protoc --proto_path=../protobuf --java_out=src/main/java/ ../protobuf/greapi.proto

This assumes that the greapi.proto file is stored in the relative ../protobuf folder and that you want to output your java files to the src/main/java folder.

Of course, other generators can be used for other languages and platforms. Android applications have a Gradle plugin; Maven applications have a Maven plugin; and C++ CMake applications have a Protobuf API defined which can help automate this task easily.

The Protobuf's documentation page contains documentation on most of these tools and libraries.

WebSocket Library

You will probably want to use a library for WebSocket communication. However, any library will do, as long as it supports binary messages.

Project's API Key

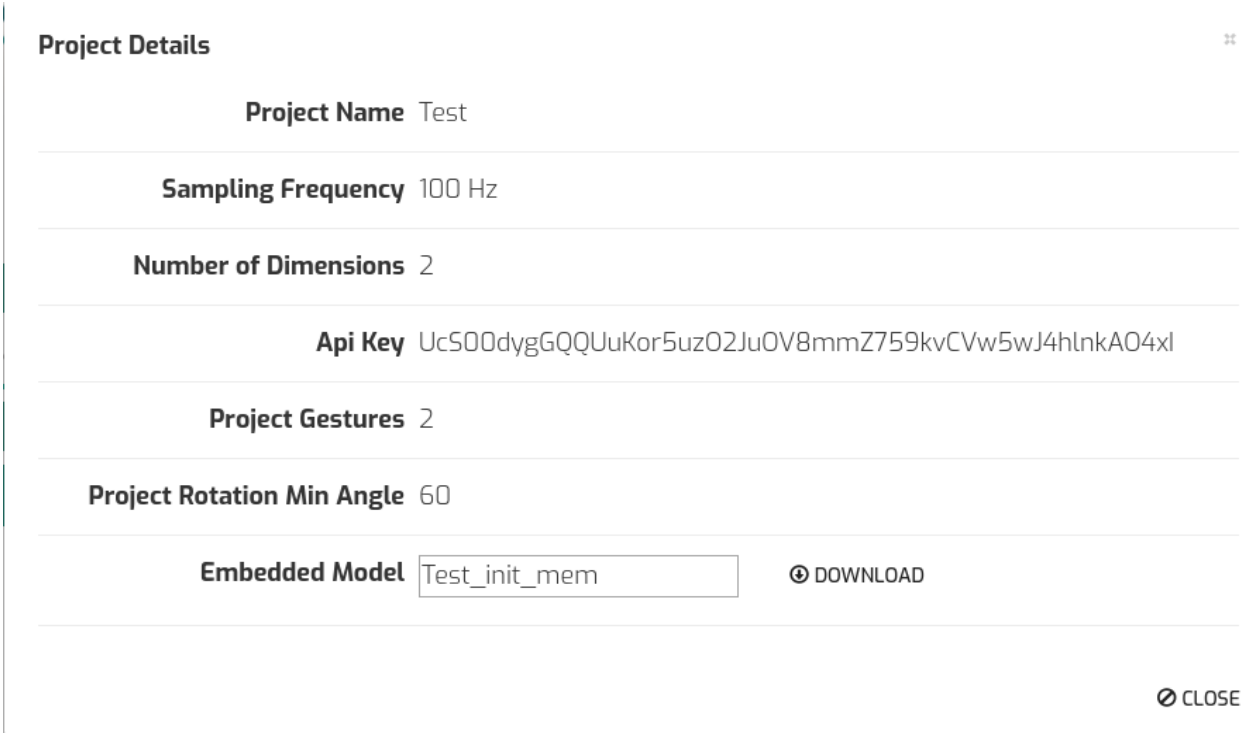
When a project is created in the GRE, an API Key is generated for it. You'll need the project API Key when making a connection to the WebSocket endpoint.

The project API Key needs to be passed as an URI query parameter:

wss://sdk.motiongestures.com/recognition?api_key=sXLo6Gi22U1Tg89qh71EYf8lOTD2RusmYUTBQKKZF3vds4DrJw

To view your project's API key, go to [https://sdk.motiongestures.com,](https://sdk.motiongestures.com) log into your account, open your project, and then click on the cog icon near the top right of the page:

Select DETAILS from the pull-down menu. A dialog box containing the project API KEY will open up:



Motion Sensor: Basic Workflow

In order to obtain gesture recognition results from motion sensor(s), a gestures-based application needs to perform the following steps:

1. Continuously listen for sensor data, keeping at least the last 125 ms of samples in a buffer. Minimum number of samples can be calculated as: Project Frequency / 8
2. Create a secure websocket connection to wss://sdk.motiongestures.com/recognition?api_key=<your api key>
3. After the websocket connection has been made successfully, send the data in the cache, with the activeGesture flag set to *false* (as those samples were not part of a gesture)
4. At regular intervals, send the RecognitionRequest message with its Acceleration, Gyroscope and Magnetometer properties set, to the server over the opened websocket connection. The intervals can range from one second to longer. Now the activeGesture flag is set to *true*.
5. When receiving a message from the server, convert the binary data to a RecognitionResponse object and look at the status member. If the status is GestureEnd then a gesture has been recognized and the label member will contain the label of the recognized gesture.
6. Perform whatever action you deem necessary based on recognized gesture.
7. Before closing the connection, send another request (with potentially no sensor samples) but with activeGesture flag set to false again, to signify the end of the gesture

Touch Sensor: Basic Workflow

In order to obtain gesture recognition results from touch sensor, a gestures-based application needs to perform the following steps:

1. Create a secure websocket connection to wss://sdk.motiongestures.com/recognition?api_key=<your api key>
2. Send the RecognitionRequest message whenever drawing of the gesture has been finished on the touch screen. Do not forget to set the requestType member to TouchRequest
3. When receiving a message from the server, convert the binary data to a RecognitionResponse object and look at the status member. If the status is GestureEnd then a gesture has been recognized and the label member will contain the label of the recognized gesture.
4. Perform whatever action you deem necessary based on recognized gesture.

The RecognitionRequest message has the following structure:

```
message RecognitionRequest {
    string id = 1;
    Acceleration acceleration = 2;
    Gyroscope gyroscope = 3;
    Magnetometer magnetometer = 4;
    uint32 sensitivity = 5;
    bool activeGesture = 7;
    RequestType requestType = 8;
    Touch touch = 9;
}
```

ID: This is the session's identifier. It is best to generate it from an UUID. The session identifier needs to be the same until a gesture is recognized. After that it can be regenerated or reused.

Acceleration: Acceleration information as gathered from accelerometer

Gyroscope: Gyroscope data

Magnetometer: Magnetometer information

Sensitivity: The sensitivity of the recognition. That is, how much noise is tolerated in the sensor data. 0 is very high sensitivity (no noise) and anything over 250 is very low sensitivity (a lot of noise).

ActiveGesture: A flag that marks samples that should be part of a gesture.

RequestType: MotionRequest or TouchRequest. If not set MotionRequest is assumed. Set it to TouchRequest when recognizing a touch gesture and the Touch message is set.

Touch: Touch message containing the points captured from the touch interface.

The RecognitionResponse message has the following structure:

```
message RecognitionResponse {
      float confidence = 1;
      repeated int32 labels = 7;
      int32 length = 3;
      Status status = 4;
      repeated string names = 8;
      GestureType gestureType = 6;
}
```

Confidence: System's confidence in its response. Probability can be calculated with $e^{confidence}$ (where e is Euler's number)

Labels: Numeric identifiers of all the recognized gestures

Length: Number of samples that made the gestures

Status: Status of the recognition

Name: Names of the recognized gestures

GestureType: Type of the recognized gestures (NONE, MOTION, ROTATIONAL, TOUCH)

Error Codes

The following error codes are used by the system:

- 4892: Recognition Limit Exceeded – This error code is returned when the account exceeded the limit for recognizing gestures.
- 4893: Invalid API key – The provided api key is invalid/not found
- 4894: Recognition Error – An internal server error occurred during recognition
- 4895: Invalid project model – The project's model couldn't be loaded
- 4896: Reserved

Development API

We provide a free perpetual license to developers for development purposes. The license facilitates free use of the SDK by processing up to 5,000 gesture recognition hits per day.

In order to get developer privileges for free testing, developers must use a different API endpoint: **wss://sdk.motiongestures.com/developmentRecognition**

Unlike the API used for commercial production, the Development API expects an access token URI parameter. Therefore, the invocation for Development API looks like:

wss://sdk.motiongestures.com/developmentRecognition?api_key=<key>&access_token=<token>

The access token is obtained by authenticating against the application.

Authentication API is a basic HTTPS based API which is executed against https://sdk.motiongestures.com:

POST /api/authenticate

Request Header: Content-Type: application/json

Request Body:

{

"username":<username>,

"password":<password>,

"rememberMe":<true|false>

}

Successful response:

Status - 200 OK

Response body - {“id_token”: <access_token>}

For example, a curl command to obtain the access token is:

curl -H "Content-Type: application/json" -X POST -d '{ "username" : "username" , "password" : "password" }' https://sdk.motiongestures.com/api/authenticate

To which a response body would look like:

{"id_token":"eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJzZyIsImF1dGgiOiJST0xFX1VTRVIiLCJleHAiOjE1MTY4OTk0MzV9.Kh8zLAKhrekq5JxrZ_d9f4dkduAjUjbzkInVI1REtnalVvJgtyzKw2uPEcbcfc9SwFcvN94_wWC9n0qDMka3DA"}

Motion Sensor: Gesture Recognition Example

As an illustration of motion sensor-based gesture recognition, let's create an Android application that will display a list of recognized gestures using motion sensors (ie. accelerometer, gyroscope and magnetometer). It is necessary to use both accelerometer and gyroscope for higher accuracy.

The full contents of the Android project can be found at:
https://github.com/motiongestures/MotionGesturesExample

Create an Android Studio Project

We will first need to create an Android Studio project with a Basic Activity template.

Since our application will need internet access, we will have to update the AndroidManifest.xml file with that request: <uses-permission android:name="android.permission.INTERNET" />

Download the greapi.proto file from https://sdk.motiongestures.com/protobuf/greapi.proto and save it in the project's folder. (You can save it anywhere you want, but then be prepared to change the protobuf fileTree's location accordingly).

Setup Needed Libraries

We will add in the app/build.gradle file the two libraries which our application will use: Protobuf and a WebSocket library.

To use protobuf and to have the necessary classes generated for us whenever we build our application, we need to add the protobuf-gradle-plugin to our buildscript:

```
buildscript {
   repositories {
      maven { url "https://plugins.gradle.org/m2/" }
   }
   dependencies {
      classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.2'
   }
}
```

And to apply the plugin to our application:

```
apply plugin: 'com.google.protobuf'
```

Next we need to configure the protobuf task for gradle:

```
protobuf {
   protoc {
      artifact = "com.google.protobuf:protoc:3.0.0"
   }
   plugins {
```

```
        lite {
            artifact = 'com.google.protobuf:protoc-gen-javalite:3.0.0'
        }
    }
    generateProtoTasks {
        all()*.plugins {
            lite {}
        }
    }
}
```

And finally, execute it as a dependency:

```
dependencies {
        …
        protobuf fileTree("../")
        implementation 'com.google.protobuf:protobuf-lite:3.0.1'
}
```

For the websocket library, just add the chosen library as a compile dependency to our project:

```
implementation 'com.neovisionaries:nv-websocket-client:2.3'
```

Note: we're using gradle 4.4 here with build tools 3.1.0

Specify Application's Layout

Our application will have a ToggleButton which will start/stop acceleration listening and display a list of items that have been recognized. A simple way to realize this is to set our activity's layout to a vertical Linear layout and add needed components.

The entire activity_main.xml file will look something like:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    tools:context="com.motiongestures.gesturerecognitionexample.MainActivity">
    <ToggleButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/test_toggle"
        android:textOff="Start Test"
        android:textOn="Stop Test"
```

```
    />
  <ListView
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:id="@+id/recognizedGesturesList"
    android:layout_weight="1"
    >
  </ListView>
</LinearLayout>
```

To add items to a ListView we need to create an item template for it. To do that, create a new Layout resource file in the layout folder, which will have only a TextView:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
  android:layout_height="match_parent">
</TextView>
```

MainActivity.java

Now let's go to MainActivity.java file.

To add items to a ListView we need to attach a ListAdapter to it. For our case, since we are only displaying a String, an array adapter will suit us just fine.

```
gesturesListAdapter = new ArrayAdapter<>(this,R.layout.gesture_item);
ListView recognizedGesturesList = findViewById(R.id.recognizedGesturesList);
recognizedGesturesList.setAdapter(gesturesListAdapter);
```

We also need to associate a ToggleButton object to the layout's view and to add a checked listener to it to be able to respond to it:

```
toggleButton = findViewById(R.id.test_toggle);
toggleButton.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
  @Override
  public void onCheckedChanged(CompoundButton compoundButton, boolean checked) {
    if(checked) {
      //start
            connect();
    } else {
      //stop
             disconnect();
```

```
    }
  }
});
```

When *connect* will be called, we will create a WebSocket connection to the server, specifying our project's API key as a parameter and register a WebSocketAdapter (an internal class that just extends WebSocketAdapter) as a listener for events.

You will, of course, need to replace the api_key parameter with your project's API key.

```
private SocketAdapter socketAdapter = new SocketAdapter();
private void connect() {
  try {
    webSocket = new
WebSocketFactory().createSocket( "wss://sdk.motiongestures.com/recognition?api_key=sXLo6Gi22U1
Tg89qh71EYf8lOTD2RusmYUTBQKKZF3vds4DrJw" );
     webSocket.addListener(socketAdapter);
    currentSessionId = UUID.randomUUID().toString();
    webSocket.connectAsynchronously();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

In order listen to our device's sensors, we will designate our activity as the sensor listener by implementing the SensorEventListener interface. In the implemented onSensorChanged method, we will receive the sensors values.

To register for sensor messages, we will obtain a SensorManager and the needed Sensor objects from our Android platform:

```
private SensorManager sensorManager;
private Sensor accelerometer;
private Sensor gyroscope;
private Sensor magnetometer;
```

And we define the types of sensors we need:

```
private static final int ACCELEROMETER_TYPE = Sensor.TYPE_ACCELEROMETER;

private static final int GYROSCOPE_TYPE = Sensor.TYPE_GYROSCOPE_UNCALIBRATED;

private static final int MAGNETOMETER_TYPE = Sensor.TYPE_MAGNETIC_FIELD;
```

and in the onCreate method:

```
sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
accelerometer = sensorManager.getDefaultSensor(ACCELEROMETER_TYPE);
```

```
gyroscope = sensorManager.getDefaultSensor(GYROSCOPE_TYPE);
magnetometer = sensorManager.getDefaultSensor(MAGNETOMETER_TYPE);
```

In the onResume method we will register ourselves to listen for sensor data and in the onPause we will remove ourselves from the list of listeners.

```
@Override
protected void onResume() {
    sensorManager.registerListener(MainActivity.this, accelerometer, 10_000);
    sensorManager.registerListener(MainActivity.this, gyroscope, 10_000);
    sensorManager.registerListener(MainActivity.this, magnetometer, 10_000);
    super.onResume();
}
```

The sampling period of 10,000 is used because our project has been defined as having a frequency of 100Hz. Since registerListener expects that field to be expressed in microseconds and we want 100 samples per second, that parameter will be 10,000.

Now we will start receiving sensor information in our onSensorChanged method.

We will collect that information in a simple ArrayList and send it to the server once we have received 100 samples (one per second) in any sensor.

Our onSensorChanged method will look like this:

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {
float x = sensorEvent.values[0];
float y = sensorEvent.values[1];
float z = sensorEvent.values[2];
Greapi.SensorSample sample =
Greapi.SensorSample.newBuilder().setX(x).setY(y).setZ(z).setIndex(index).build();

switch(sensorEvent.sensor.getType())
{
    case ACCELEROMETER_TYPE:
        if(activeGesture) {
            accelerationList.add(sample);
        }
        addSampleToCache(accelerationSamplesCache,sample);
        break;
    case GYROSCOPE_TYPE:
        if(activeGesture) {
            gyroscopeList.add(sample);
        }
        addSampleToCache(gyroscopeSamplesCache,sample);
        break;
    case MAGNETOMETER_TYPE:
        if(activeGesture) {
```

```
                magnetometerList.add(sample);
            }
            addSampleToCache(magnetometerSamplesCache,sample);
            break;
    }
    index++;
    if(shouldSendData()) {
        try {
            sendSamples(accelerationList,gyroscopeList,magnetometerList);
            accelerationList.clear();
            gyroscopeList.clear();
            magnetometerList.clear();
        } catch (IOException ex) {
            Log.e(TAG, "Error sending acceleration data to the server", ex);
        }
    }

}

private void sendSamples(Iterable<? extends Greapi.SensorSample> accelerations,

                                                    Iterable<? extends Greapi.SensorSample>
gyroscope,
                                                    Iterable<? extends Greapi.SensorSample>
magnetometer) throws IOException {
    Greapi.Acceleration accelerationMessage = Greapi.Acceleration.newBuilder()
            .addAllSamples(accelerations)
            .setUnit(Greapi.AccelerationUnit.SI)
            .build();
    Greapi.Gyroscope gyroscopeMessage = Greapi.Gyroscope.newBuilder()
            .addAllSamples(gyroscope)
            .setUnit(Greapi.GyroscopeUnit.RADS)
            .build();
    Greapi.Magnetometer magnetometerMessage = Greapi.Magnetometer.newBuilder()
            .addAllSamples(magnetometer)
            .build();
    Greapi.RecognitionRequest recognition = Greapi.RecognitionRequest.newBuilder()
            .setId(currentSessionId)
            .setSensitivity(100)
            .setActiveGesture(activeGesture)
            .setAcceleration(accelerationMessage)
            .setGyroscope(gyroscopeMessage)
            .setMagnetometer(magnetometerMessage)
            .build();
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    recognition.writeTo(outputStream);
    webSocket.sendBinary(outputStream.toByteArray());
}
```

currentSessionId is a string that has been initialized with UUID.randomUUID().toString() when we have established the websocket connection. Index is an integer that is initialized with 0 at the same time.

In our SocketAdapter we need to override the onBinaryMessage method so that we can receive server's responses. The byte array that we receive from the server needs to be deserialized into a RecognitionResponse object. If that object's status field is GestureEnd, it means that we have recognized a gesture.

A potential implementation of the onBinaryMessage method is as follows:

```
@Override
public void onBinaryMessage(WebSocket websocket, byte[] binary) throws Exception {
   try{
ByteArrayInputStream inputStream = new ByteArrayInputStream(binary);
final Greapi.RecognitionResponse recognitionResponse =
Greapi.RecognitionResponse.parseFrom(inputStream);
if(recognitionResponse.getStatus() == Greapi.Status.GestureEnd) {
      runOnUiThread(new Runnable() {
            @Override
            public void run() {
                  int size =
Math.min(recognitionResponse.getNamesCount(),recognitionResponse.getLabelsCount());
                  for(int i =0;i<size;i++) {
                        gesturesListAdapter.add("Recognized gesture " +
recognitionResponse.getNames(i) + " with label " + recognitionResponse.getLabels(i));
                  }
                  toggleButton.setChecked(false);
            }
      });
} else {
      Log.d(TAG,"Received recognition response with status
"+recognitionResponse.getStatus());
}

   }catch(IOException ex) {

      Log.e(TAG,"Error deserializing the recognition response",ex);
   }
}
```

We are done!

We can now test our application on an Android device.

Toggling the Start/Stop button will cause our application to send motion sensors data to the server for interpretation and receive gesture recognition results.

Touch Sensor: Gesture Recognition Example

As an illustration of touch sensor-based gesture recognition, let's create an Android application that will display a list of recognized gestures using the touch sensor.
The full contents of the Android project can be found at:
https://github.com/motiongestures/TouchGesturesExample .

Create an Android Studio Project

We will first need to create an Android Studio project with a Basic Activity template.

Since our application will need internet access, we will have to update the AndroidManifest.xml file with that request: <uses-permission android:name="android.permission.INTERNET" />

Download the greapi.proto file from https://sdk.motiongestures.com/protobuf/greapi.proto and save it in the project's folder. (You can save it anywhere you want, but then be prepared to change the protobuf fileTree's location accordingly).

Setup Needed Libraries

We will add in the app/build.gradle file the two libraries which our application will use: Protobuf and a WebSocket library.

To use protobuf and to have the necessary classes generated for us whenever we build our application, we need to add the protobuf-gradle-plugin to our buildscript:

```
buildscript {
   repositories {
      maven { url "https://plugins.gradle.org/m2/" }
   }
   dependencies {
      classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.2'
   }
}
```

And to apply the plugin to our application:

```
apply plugin: 'com.google.protobuf'
```

Next we need to configure the protobuf task for gradle:

```
protobuf {
   protoc {
      artifact = "com.google.protobuf:protoc:3.0.0"
   }
   plugins {
```

```
        lite {
            artifact = 'com.google.protobuf:protoc-gen-javalite:3.0.0'
        }
    }
    generateProtoTasks {
        all()*.plugins {
            lite {}
        }
    }
}
```

And finally, execute it as a dependency:

```
dependencies {
        …
        //change this path to wherever the proto file is saved
        protobuf fileTree("../greapi.proto")

        implementation 'com.google.protobuf:protobuf-lite:3.0.1'

}
```

For the websocket library, just add the chosen library as a compile dependency to our project:

```
implementation 'com.neovisionaries:nv-websocket-client:2.3'
```

Note: we're using gradle 4.4 here with build tools 3.1.0

Specify Application's Layout

Our application will have a TouchView onto which we can draw our gestures and below a list of recognized gestures. A simple way to realize this is to set our activity's layout to a vertical Linear layout and add needed components.

The entire activity_main.xml file will look something like:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    android:weightSum="8"
    tools:context=".MainActivity">
    <com.motiongestures.touchgesturesexample.TestTouchGestureView
        android:layout_width="match_parent"
        android:layout_height="0dp"
```

```
        android:layout_weight="5"
        android:id="@+id/touchView"  />
    <ListView
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:id="@+id/touchRecognizedGesturesList"
        android:layout_weight="3"
        >
    </ListView>
</LinearLayout>
```

To add items to a ListView we need to create an item template for it. To do that, create a new Layout resource file in the layout folder, which will have only a TextView:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
  android:layout_height="match_parent">
</TextView>
```

TestTouchGesureView.java

The touch surface is a simple view which responds to touch events and draws the received points, notifying the registered listener when the drawing has finished. To do this it overrides the onTouchEvent method and onDraw for custom drawing.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    int size = event.getHistorySize();
    for(int i=0;i<size;i++) {
        handleTouch(event.getAction(),event.getHistoricalX(i),event.getHistoricalY(i));
    }
    float x = event.getX();
    float y = event.getY();
    return handleTouch(event.getAction(), x,y);
}

private boolean handleTouch(int action,float x, float y) {

    switch (action) {
        case MotionEvent.ACTION_DOWN:
            touchStart(x, y);
            invalidate();
            break;
        case MotionEvent.ACTION_MOVE:
            touchMove(x, y);
            invalidate();
            break;
        case MotionEvent.ACTION_UP:
```

```
                touchUp();
                invalidate();
                if(listener != null) {
                    listener.drawingFinished(new TestGestureDrawingFinishedEvent(this,points));
                }
                break;
            case MotionEvent.ACTION_CANCEL:
                clearDrawing();
                invalidate();
                break;
        }
        return true;
    }
    private void touchStart(float x, float y) {
        Log.d(TAG,"Touch start");
        mX = x;
        mY = y;
        clearDrawing();
        gesturePath.moveTo(x, y);
        points.add(new TouchGesturePoint(x,y));
        circlePath.addCircle(mX, mY, CIRCLE_RADIUS, Path.Direction.CW);
    }
    private void touchMove(float x, float y) {
        Log.d(TAG,"Touch move");
        float dx = Math.abs(x - mX);
        float dy = Math.abs(y - mY);
        if (dx >= TOUCH_TOLERANCE || dy >= TOUCH_TOLERANCE) {
            points.add(new TouchGesturePoint(x,y));
            gesturePath.quadTo(mX, mY, (x + mX)/2, (y + mY)/2);
            mX = x;
            mY = y;
            // circlePath.reset();
            circlePath.addCircle(mX, mY, CIRCLE_RADIUS, Path.Direction.CW);
        }
    }
    private void touchUp() {
        Log.d(TAG,"Touch up");
        gesturePath.lineTo(mX, mY);
        //TouchGesturePoint point = gesture.addPoint(mX,mY);
        // commit the path to our offscreen
        mCanvas.drawPath(gesturePath, gesturePaint);
        mCanvas.drawPath(circlePath, circlePaint);
        // kill this so we don't double draw
        gesturePath.reset();
        circlePath.reset();
    }
```

And drawing the created paths:

```
@Override

protected void onDraw(Canvas canvas) {
```

```
        super.onDraw(canvas);
        canvas.save();
        canvas.drawBitmap( mBitmap, 0, 0, mBitmapPaint);
        canvas.drawPath(gesturePath, gesturePaint);
        canvas.drawPath( circlePath,  circlePaint);
        canvas.drawPath(gridPath,gridPaint);
        canvas.restore();
}
```

MainActivity.java

To add items to a ListView we need to attach a ListAdapter to it. For our case, since we are only displaying a String, an array adapter will suit us just fine.

```
gesturesListAdapter = new ArrayAdapter<>(this,R.layout.gesture_item);
ListView recognizedGesturesList = findViewById(R.id.recognizedGesturesList);
recognizedGesturesList.setAdapter(gesturesListAdapter);
```

The activity will register itself as a drawingFinished listener, to receive the TestGestureDrawingFinishedEvent events. In that method, the points are transformed from a TouchGesturePoint (as the view sends them) to a Protobuf, Greapi.Point objects:

```
@Override
public void drawingFinished(TestGestureDrawingFinishedEvent event) {
        lastReceivedPoints = new ArrayList<>(event.getPoints().size());
        for(TouchGesturePoint point : event.getPoints()) {

lastReceivedPoints.add(Greapi.Point.newBuilder().setX(point.getX()).setY(point.getY()).build());
        }
        connect();
}
```

When *connect* will be called, we will create a WebSocket connection to the server, specifying our project's API key as a parameter and register a WebSocketAdapter (an internal class that just extends WebSocketAdapter) as a listener for events.

You will, of course, need to replace the api_key parameter with your project's API key.

```
private SocketAdapter socketAdapter = new SocketAdapter();
private void connect() {
   try {
      webSocket = new
WebSocketFactory().createSocket( "wss://sdk.motiongestures.com/recognition?api_key=sXLo6Gi22U1
Tg89qh71EYf8lOTD2RusmYUTBQKKZF3vds4DrJw" );
      webSocket.addListener(socketAdapter);
     currentSessionId = UUID.randomUUID().toString();
     webSocket.connectAsynchronously();
```

```
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

In the SocketAdapter's onConnected method (which notifies us when the connection has been established) we can send the touch data that we have gathered:

```
@Override
public void onConnected(WebSocket websocket, Map<String, List<String>> headers) throws
Exception {
        Log.d(TAG,"Connected to server");
        sendLatestSamples();
}
```

The sendLatestSamples method simply constructs the protofbuf message and sends it:

```
private void sendLatestSamples() throws IOException {
        Greapi.Touch touchMessage = Greapi.Touch.newBuilder()
                        .addAllPoints(lastReceivedPoints)
                        .build();
        Greapi.RecognitionRequest recognition = Greapi.RecognitionRequest.newBuilder()
                        .setId(currentSessionId)
                        //IMPORTANT: set the request type to touch request
                        .setRequestType(Greapi.RequestType.TouchRequest)
                        .setTouch(touchMessage)
                        .build();
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        recognition.writeTo(outputStream);
        webSocket.sendBinary(outputStream.toByteArray());
}
```

currentSessionId is a string that has been initialized with UUID.randomUUID().toString() when we have established the websocket connection. Index is an integer that is initialized with 0 at the same time.

In our SocketAdapter we need to override the onBinaryMessage method so that we can receive server's responses. The byte array that we receive from the server needs to be deserialized into a RecognitionResponse object. If that object's status field is GestureEnd, it means that we have recognized a gesture.

A potential implementation of the onBinaryMessage method is as follows:

```
@Override
public void onBinaryMessage(WebSocket websocket, byte[] binary) throws Exception {
        try{
                ByteArrayInputStream inputStream = new ByteArrayInputStream(binary);
                final Greapi.RecognitionResponse recognitionResponse =
Greapi.RecognitionResponse.parseFrom(inputStream);
                if(recognitionResponse.getStatus() == Greapi.Status.GestureEnd) {
```

```
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        int size =
Math.min(recognitionResponse.getNamesCount(),recognitionResponse.getLabelsCount());
                        for(int i =0;i<size;i++) {
                            gesturesListAdapter.add("Recognized gesture " +
recognitionResponse.getNames(i) + " with label " + recognitionResponse.getLabels(i));
                        }
                    }
                });
            } else {
                Log.d(TAG,"Received recognition response with status
"+recognitionResponse.getStatus());
            }
        }catch(IOException ex) {
            Log.e(TAG,"Error deserializing the recognition response",ex);
        }
        disconnect();
}
```

Now we can test our application on an Android device.

Drawing a gesture on the screen will draw registered points, send points to the server for interpretation, and display the obtained recognition results.