CO_2 Emission Alert Project on Android Platform

Xiang Gao xzgao@cs.helsinki.fi
Junlong Xiang lxsgdtc@gmail.com
Feihua Qu qu@cs.helsinki.fi

16 December 2012

Abstract

Nowadays, the emission of carbon dioxide has emerged to be an upcoming challenge for human and environment. People are enjoying the environment-friendly style of life by using ecological applications. We developed CO2EmissionAlert, a lightweight Android application to inspect daily carbon footprints especially for transportation process. It features real-time tracking on Google map, multi-modal feedback with visual and speech alerts, and shake-triggered feedback. This report provides the details about design, development, practical experiments, evaluation and discussion on CO2EmissionAlert project. We have learned much from the process of collaborative working and finding solutions for problems.

Contents

1 Introduction **3**

2 Related work **4**

3 Design **7**

3.1 Functional Specification 7

3.2 Concept Art 8

3.3 Workflow 13

4 Development **14**

4.1 Environment & Preparation 15

4.2 Architecture & Algorithms 16

4.2.1 CO_2 Emission Calculation 18

4.2.2 WelcomeActivity 18

4.2.3 SettingActivity 19

4.2.4 MapTracking 20

4.2.5 Text-to-Speech 22

4.2.6 ShakeListener 22

4.2.7 Database & Data Visualization 25

4.3 Debugging & Testing 26

4.4 Facing Problems 28

4.5 Further Plan 29

5 Discussion **30**

6 Conclusion **30**

1 Introduction

Recent research on atmosphere [2] suggests that carbon dioxide that constitutes the body of greenhouse gases (GHG) is threatening our environment. "Global emissions of carbon dioxide have risen by 99%, or on average 2.0% per year, since 1971, and are projected to rise by another 45% by 2030, or by 1.6% per year." [2] The CO_2 emission problem is mainly caused by burning fuels of oil, coal and gas. As the modern society evolves, people are following the environment-friendly style of life by using ecological applications in their daily life. New eco-apps on smartphones are developed adopting interface technologies to calculate personal carbon footprint [7], i.e., the CO_2 emissions.

We are developing an Android application that monitors the real-time CO_2 emission information in transportation of various vehicles (motorcycles, bus, trams, cars, trains, bikes). According to our design, once initialized, the application should inform the user of

- the current distance (route) of trip on a map view,
- the real-time CO_2 emission amount based on the distance,
- the auditory alert upon reaching the CO_2 emission threshold predefined (hardcoded),
- the speech & vibration alert triggered by shaking the smartphone in transportation,
- the view of summary statistics after the tracking process.

Here is the description of an application scenario. A user is to start his trip from home to workplace by car. Upon firing the engine, he takes out his smartphone (e.g. Samsung Galaxy SIII), initializes the CO_2 emission alert application by setting the predefined parameters, and places his smartphone aside. There are three ways of getting feedbacks: the realtime CO_2 emission information is visually updated on the MapView; the user can hear an auditory alert when the car exceeds the predefined CO_2 emission limit, and the auditory feedback can be a speech message of the current CO_2 emission information; the user can also get the current CO_2 emission information whenever he shakes his smartphone, and the application responds with a speech message of the current CO_2 emission information. After the trip, the user checks the summary view which provides the total amount of CO_2 emission and the percentage of saving compared with other transportation modes (motorcycles, bus, trams, trains, bikes, etc.).

The tracking process utilizes the Global Positioning System (GPS) receiver (cell-network-based coarse positioning as a backup), compass and accelerometer as sensors, and the multi-modal feedback will be visual , vibration and speech output. The application tracks the geographical position automatically via the Google Map service, and depicts the current route on the map view. The real-time CO_2 emission amount shown on the map view is calculated by multiplying CO_2 emission coefficient for unit length with the current distance. The speech message as a feedback is synthesized from current CO_2 emission amount with a text-to-speech (TTS) system by invoking SDK interfaces.

In this report, we present the top-down view of our project. Section 2 discusses the previous work of selected publications. Section 3 provides the concept model, functional block

diagram and description of functions. Section 4 illustrates the detail process of development including task distribution, methods and approaches, and snapshots of the process. Section 5 discusses the reflection on the project: strong & weak points and improvement scheduled. Finally the report concludes in Section 6.

2 Related work

In this section, we present selected publications on carbon footprint calculation and mobile systems close to our project.

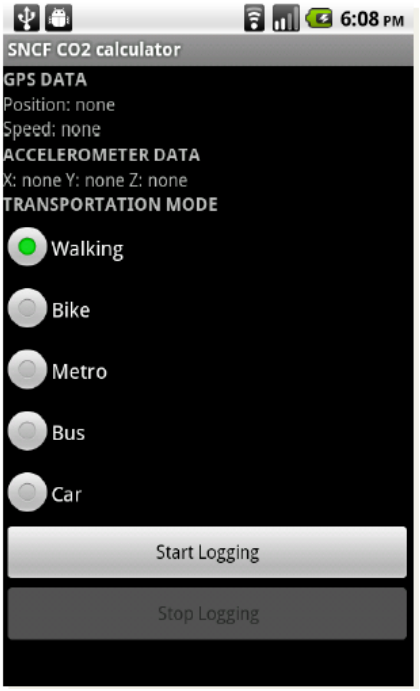

Figure 1: CO2GO interface of transportation mode inference [6].

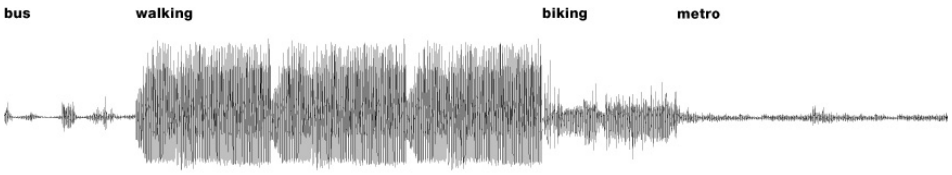

Figure 2: CO2GO: accelerometer traces for distinct transportation modes [3].

C.Ratti et al.'s patent [7] and V.Manzoni et al.'s technical report [6] introduce an innovative system CO2GO for estimating real-time personal CO_2 emissions. It is a featured

project developed by SENSEable city lab of MIT. CO2GO classifies transportation into eight modes as in Table 1 and computes CO_2 emissions based on the transportation mode and distances. The system features in automatic transportation mode inference, which utilizes data processed from accelerometers and GPS receivers to determine the identify. Google Nexus One with Android 2.2 samples the accelerometer and GPS data, which is then processed with FFT. Supervised machine learning algorithms based functional trees are adopted to extract the patterns for eight modes from processed accelerometer traces (see Figure 2), and OpenStreetMap API is combined to identify roads and lanes on the map. A second algorithm for battery saving is also used: sensors are activated from idle state once motion is detected. CO2GO application provides a further social function that users can share their routes and carbon footprints with peers in which case one can choose an alternative route with lower CO_2 emissions.

Figure 3: CarbonRecorder: system architecture [5].

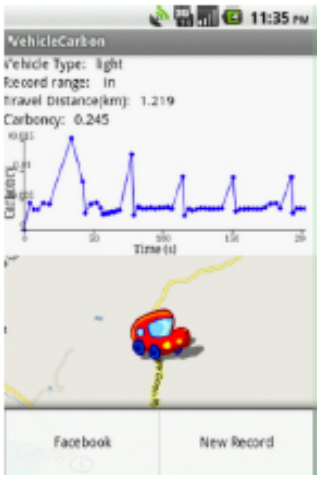

Figure 4: CarbonRecorder: live interface of carbon emission vs. time curve and route map [5].

B.Liu et al.'s paper [5] demonstrates Carbon Recorder, a mobile-social application developed by researchers from UCDevis. It is designed to record user's carbon footprint in daily life and share it with friends on Facebook. The authors' motivation is not only to encourage trip with lower CO_2 emissions, but also to facilitate the research on traffic management and social game issues. Figure 3 presents the hybrid architecture with mobile clients (Android), a back-end server and Facebook applications. The mobile client is used for recording and displaying the carbon footprint of user's trip. The back-end server receives recorded information from mobile client and provides data for the Facebook application. The first feature that differs from CO2GO is that Carbon Recorder needs users specify the transportation mode beforehand, just as our project. But Carbon Recorder adopts a fuel consumption model to compute the carbon footprint, a more accurate approach than ours. The second feature is the social functionality that enables sharing and visualizing on Facebook other than smartphones.

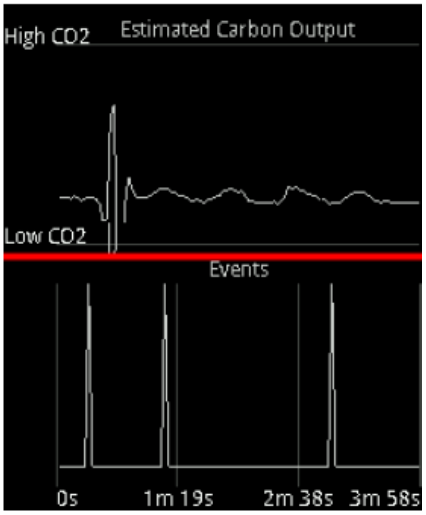

Figure 5: Eco Driving Tutor: real-time CO_2 tracking and events occurred summary [4].

A.Austin's report [4] on Eco Driving Tutor is another related example of mobile application on Android, designed for tracking the driving style of users and encouraging driving behavior with more efficient styles. Theoretically, the author has specified three styles of behavior that result in higher fuel consumption and higher CO_2 emissions: harsh braking, harsh acceleration and high speed. These so-called "inefficient" events when driving a car triggers the driver with both visual and audio alerts as real-time feedbacks. The approach of detecting inefficient events is sampling readings of the accelerometer and identifying the variance that exceeds the limit with history. A low-pass filter is used to eliminate noise in processing sensor measurements. Figure 5 shows the GraphView of real-time CO_2 emission tracking, realized by invoking a queue of data when refreshing the GraphView. Considering the calculation of CO_2 emission, inefficient events introduces different methods. The overall carbon footprint consists of the CO_2 emission with average speed (in this sense high speed events do not count) combined with accountable kinetic energy in braking events. After a journey, the application provides a summary of inefficient events and a performance score, with which the driver can achieve more efficient driving styles

(higher scores).

3 Design

In this section, we walk through the specification of functions & functional block diagram, concept art, and workflow model, following our process of designing CO2EmissionAlert application.

3.1 Functional Specification

First of all, we need mention the requirements that the design process should meet. We are developing an Android application that monitors the real-time CO_2 emission information in transportation of various vehicles (bicycles, bus, trams, cars, trains, etc.). According to our design, once initialized, the application should inform the user: the ability of location current position, and tracking route information on the map; the current distance (route) of trip on a map view; the real-time CO_2 emission amount based on the distance; the speech alert of CO_2 emission triggered by shaking the smartphone in transportation; the view of summary statistics after the tracking process.

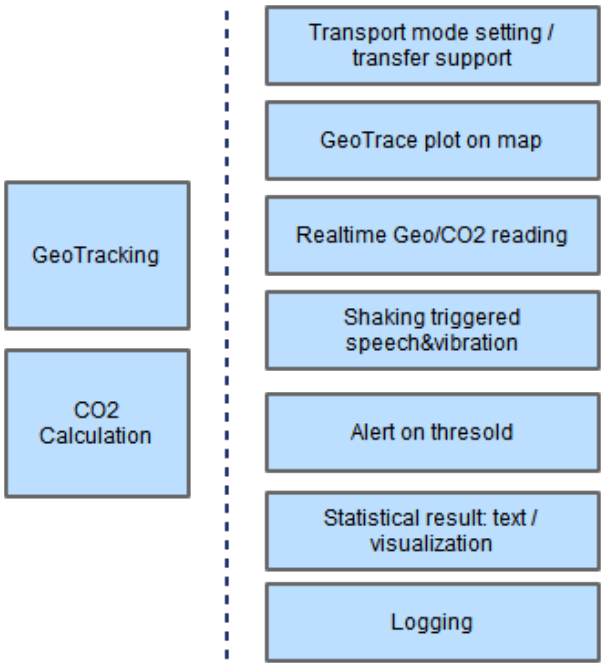

Figure 6: Functional block diagram of CO2EmissionAlert.

Figure 6 provides a general view of functions in high-level design. Geographical tracking and CO2 emission calculation are considered kernel functions. Now we can agree on the specification of lower-level description of functions in our application.

1. An interface is essential for choosing the suitable transportation mode (or background). It could also be like CO2GO system mentioned in references [7, 6] that adopted machine learning strategies for recognizing different transportation modes automatically. However, we agreed on a setting page allowing manually selecting of the predefined coefficient.

2. The widgets used for controlling the inspection process, like starting or stopping buttons, and the function of resetting on the map view. We found the buttons intuitive for users to trigger inspection in a welcoming manner.

3. Functions pertaining to the map view, with positioning and route-plotting as the main features. We decide to ensure that the map view tracks the current location of user with focus on current point and the route originating from start point. Basic information are indicated, such as the compass widget, the zooming widget, emphasized current point on the map, the transport/transfer marker, and the route containing information of transfer.

4. The ability to well present the real-time distance and carbon emission information coupled with the map view. There is visually updating indicator on the map, providing geo-location, distance, duration and CO2 emission information. We also implemented speech & vibration alert when triggered by shaking pattern recognition (passive strategy). There is still additional implementation for active speech alerts upon exceeding a predefined threshold for carbon emission, while the threshold is hardcoded in development.

5. Transfer support in transportation process. It's intrinsic property of such kind of applications to guarantee the layout of transfer information (hybrid modes of transportation) as well as the calculation logic behind the view. So we agreed on an option menu or (floating) context menu to update inspection environmental parameters (e.g. coefficient M when transferring distance values to carbon emissions). Route-plots are also augmented with colors and icon markers.

6. Logging and summary function weigh much in concluding a complete trip. We implemented a lightweight database to log the GPS location information, with which the application produces the CO2 emission rate chart as a summary. Also, we provide total and average statistics of CO2 emissions in the summary view.

3.2 Concept Art

Following the functional specification, we proceed to the concept art section, i.e. the mockups. Various mockup tools are available online or offline, but what we had sketched were basic mockups, enough for CO2EmissionAlert as a lightweight mobile application without a back-end server. Totally four views were sketched as pages: the welcome page, the setting page with the list of transportation modes and start & stop buttons, the map view with simple layout of map and an option menu, and the summary page presenting the carbon footprint data and chart. We had in mind further features such as social sharing of carbon footprints, but the corresponding view of login page and sharing page are not prepared in this phase of design & development.

Here are the specifications of mockup views. We need mention the implicit requirement: the hardware buttons as well as the touching screen. More specifically, we take Samsung Galaxy S3 as the model, equipped with 4.7 inch WXGA screen and home & back hardware buttons.

Figure 7: Prototype of Welcome View in CO2EmissionAlert.

1. Welcome View:

   As Figure 7 sketched, welcome view is the very front page starting CO2EmissionAlert application. The layout is simple: the main area is filled with a theme picture indicating application name 'CO2 Emission Alert'; two TextViews are added: the version indicator residing at the upright corner, and the information of our group at the bottom of view. We can see further in development phase, `activity_welcome.xml` is used to describe the RelativeLayout of welcome view.

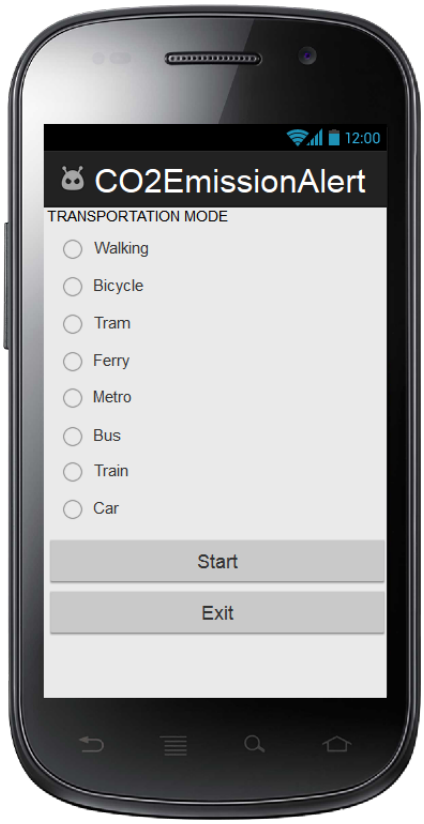

Figure 8: Prototype of setting view in CO2EmissionAlert.

2. Setting View:

   Figure 8 presents a LinearLayout (see `activity_setting.xml`). The top TextView below the action bar indicates that this is for selecting transportation modes. Then the next widget is a group of radio buttons (for single selection). Eight radio buttons groups as a column, with indicator of transportation modes shown. Then below are two control buttons: start button and exit button, with each width stretched to the margins. No option or context menus are planned in setting view.

Figure 9: Prototype of map view in CO2EmissionAlert.

3. Map View:

   Map view (see Figure 9) is the kernel view of design & development job. It presents two sketches: the map view with and without the explicit option menu. It's simple layout launching map view with the MapView filling the whole area below the action bar (see `/res/layout/activity_maptracking.xml`). Upon pressing the hardware option button, the option menu (the middle sketch) pops up from the bottom with 6 items : Transfer, Satellite View, Map View, Reset, Stop, and Commit(see `/res/menu/activity_maptracking.xml`).

   Upon touching "Transfer" item, a sub-menu pops up as a floating menu with caption "Transfer" and list of transportation modes (see the rightmost figure). The sub-menu disappears upon touching any item inside. The "Satellite View" and "Map View" items are used for switch between satellite & normal map style in Google Maps. "Reset" item is used for triggering resetting inspection process and clearing overlays on the MapView. "Stop" item marks a cross icon at the final point of route on the map view, and stops the timer and logging in background. "Commit" item commits the end-game of inspection process and switches to summary view.

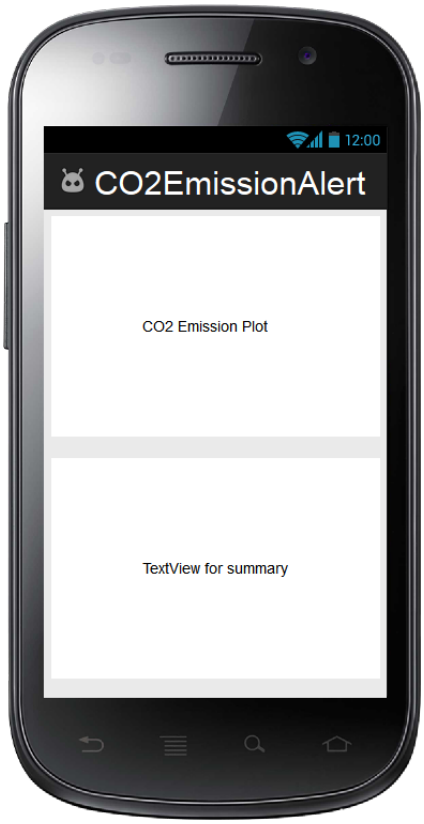

Figure 10: Prototype of summary view in CO2EmissionAlert.

4. Summary View:

   This view is planned for presenting the summary statistics with data visualization (see Figure 10). The top area is reserved for a plot of carbon emission rate vs. time, based on logging function. The bottom TextView presents the total & average values of distance and carbon emissions.

Figure 11: Prototype of exit view in CO2EmissionAlert.

5. Exit View:

   As Figure 11 sketched, exit view is the final page of CO2EmissionAlert application. The layout is simple: the main area is filled with a goodbye picture indicating application theme. It is coded in `activity_exit.xml` with RelativeLayout.

3.3 Workflow

This section describes the designed workflow (as Figure 12) of using CO2EmissionAlert application under a common scenario. We can also follow the scenario below.

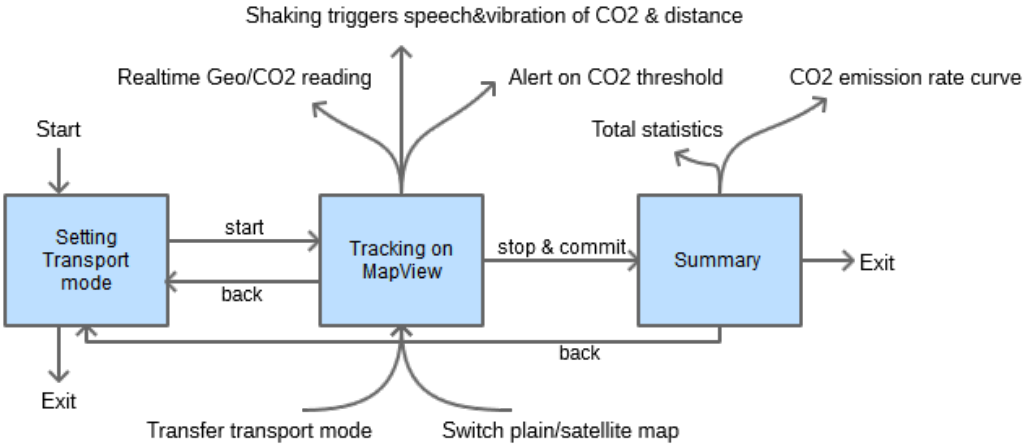

Figure 12: Workflow chart for using CO2EmissionAlert.

Upon beginning his trip from home to school, user A starts CO2EmissionAlert application in his smartphone. Then on the setting page after welcome view, he selects upcoming transportation mode (e.g. Walking) and presses the "Start" button. The screen switches to map view with his current (first points in route chain) presented. As he is walking towards the bus stop, his route is recorded on map view as colored chaining lines (e.g. red line). When he gets on the bus, he selected "Transfer"$\rightarrow$"Bus", and the map view updates with following route in a different color (e.g. cyan). He can switch to and from satellite map using option menu. And he gets the real-time value of distance and carbon emissions either from the screen where the TextView is updated whenever location change occurs, or by shaking the smartphone back and forth. Then the speech and vibration give the information alert. Along his trip, anytime the CO_2 emission exceeds a predefined threshold, the application actively alerts with a speech message. When he arrives his destination, he commits his trip by pressing “Stop” and “Commit” and got the summary of his carbon footprint along the trip. When he need start a new trip, he just resets the map view or jumps back to setting page.

4 Development

On assigning the project, we three as a group tried to figure out an accurately controlled process for this project. We were given around three weeks before the commitment date for submission (Dec 26th, which was not scheduled beforehand). We spent the first week in designing functions, modules and mockups for the application, and submitted a partial report on introduction and related work. Then the following two weeks were used for coding project files. Since we are novices for Android development, the timetable is well scheduled but not fully committed. The general strategy is incremental iterations: adding or altering a section of codes each time running a test on a debugging smartphone. After a dozen of iterations, the modules under the main architecture have formed. We faced with many problems in the process, big and small, for each of which we collaboratively discussed and tried until a suitable solution is found. And by knowing more about Android

development, iterations became rapid and smooth.

Section 4.1 introduces the environment set up for development as well as issues pertaining to project control. Section 4.2 describes the architecture and modules of this project, kernel algorithms for each module with important snippets of code. Section 4.3 indicates the settings for testing scenarios, the evaluation and snapshots of our activities. Section 4.4 enumerates two examples of handling problems in development: what we met with and how we found the solution. Finally, Section 4.5 illustrates our plan for further development of CO2EmissionAlert.

4.1 Environment & Preparation

The hardware for development is a Samsung Galaxy S3, with a 4.7' WXGA (1280x720 xhdpi) capacitive touch screen, 1024 MB RAM, 48 MB VM heap, 16 GB internal storage, and sensors needed equipped (accelerometer, gyrometer, GPS receiver). Android 4.0.3 is its operating system. In the later phase of development and experiment, we use a Motorola ME525 as an alternative, which has the equivalent sensors and Android 2.3.6 as its operating system.

We decided to start the development using an IDE, the official bundled toolkit for Android development (ADT) provided by Google. It consists of an eclipse environment, and the Android SDKs with Google Map APIs coupled. It was easy installing ADT and setting up its preferences and workspace. The two important tools are configured: the SDK manager and AVD manager. We use the SDK manager to selectively download and install suitable versions of Android SDKs. Since our debugging hardware is Galaxy S3, we need Android SDK inversion 15 (required by Android 4.0.3), constituted by the SDK platform, samples and source code, Google API in version 15, and an ARM EABI v7a system image. AVD manager is used to create and manage virtual devices for virtual emulation. One can create target virtual device from a screen metric (e.g., 4.7' WXGA) by specifying corresponding parameters of real devices. We created a virtual device named "Galaxy_S3" with the specifications above. It works for static presentation of layouts and logic, however, not suitable for debugging sensor-assisted applications like our GPS and accelerometer enabled CO2EmissionAlert. In the emulator, the map is resolved with out GPS tracking, and the only method we can adopt to the emulation and debugger was manually inputting location shift to the emulator. The debugging process was as satisfactory as using wired smartphone in debugging mode, but sensor-related data was far away from practical results. We elaborates on testing issues in Section 4.3.

With ADT set up, we embarks on starting our project in eclipse ("File"→"New Android Application"). We were required to specify the following segments to setup our project: the name for application and project (CO2EmissionAlert), the package name (`com.example.co2emissionalert`), minimum required SDK (API 8: Android 2.2 Froyo), target and compiling SDK (API 15: Android 4.0.3 IceCreamSandwich), theme for layout (Holo light with dark action bar). The next few steps was customizing preferences like workspace, and creating an activity to start with.

It is important to understand the structure of an Android project. Take CO2EmissionAlert project as an example. `/res` is the directory for resources, `/res/layout` includes all layout xml files defining layouts for each activity. The elements in xmls are editable in both

visual and coding manners. Each element in a layout file has a key-value pair, with id used for invoking elements and value defined in `/res/value/strings.xml`. The pictures and icons are stored in `/res/drawable-xxxx` directories. The reason for which we use `/res` category is to easily refer to elements in the layout as "R.layout.xxxx". For each layout, there is an Activity supporting the logic behind the screen (java files in the package `com.example.co2emissionalert`). For this project, we have `WelcomeActivity.java` for `activity_welcome.xml`, `SettingActivity.java` for `activity_setting.xml`, and MapTracking.java with ShakeListener.java (one MapActivity) for `/layout/activity_maptracking.xml`. We notice that `/res/menu` directory stores layout definition of menus (`/menu/activity_maptracking.xml`). Finally, `AndroidManifest.xml` needs configuration for acquiring permissions, e.g, access to fine and coarse location services.

4.2 Architecture & Algorithms

To illustrate the interface technologies applied, we get Figure 13, the block diagram for input & output through human-computer interface. Setting selection, transfer selection and shaking are input variables via the touch screen; Data visualization, route on map, toast reading are visual output variables via the screen, while active speech alert and triggered speech alert are auditory output variables via the speaker. We notice that the MapTracking activity is the kernel module in CO2EmissionAlert application. MapTracking handles positioning, drawing overlay, CO2 calculation, and TTS issues, so it's quite complex logic which we are to figure out in Section 4.2.4.

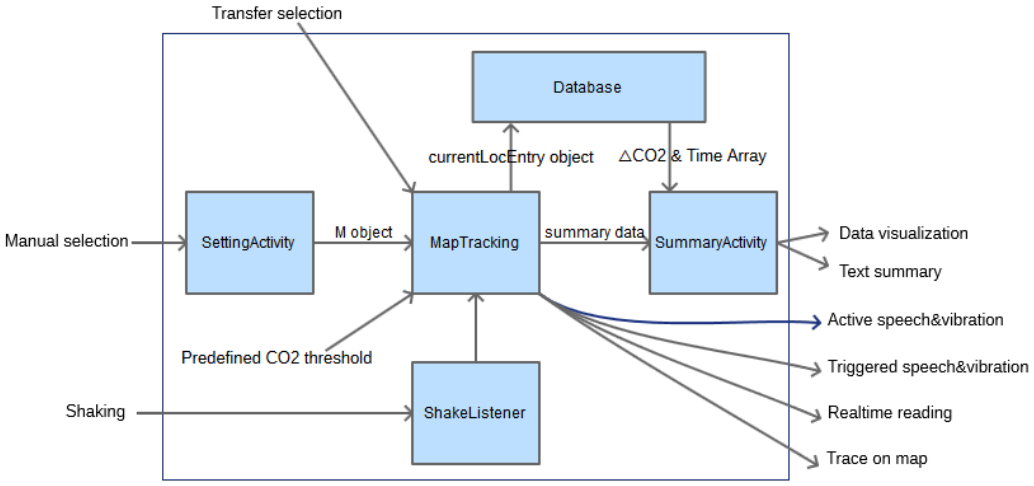

Figure 13: Input-Output block diagram for CO2EmissionAlert.

We have mentioned the structure of project files in Section 4.1. We noticed that the mechanisms behind Activity is very helpful for we coders to program and understand execution in an Android application better.

In android development, the concept "Activity" stands for activities of a window just as its

name hints, and coders extend Activity and override its methods to commit activities on screen. Very often Activity is used to interact with user interfaces, especially the screen (window). `setContentView(View)` is applied to create the window based on layout and other resources. In our project, classes `SettingActivity`, `WelcomeActivity` and `ShakeListener` all extends Activity class. Android system stores activities in a stack, in which old activities can resume only after newer activities exits. An activity can be running, paused, stopped or destroyed. We need refer to activity lifecycle model (see Figure 14) to illustrate how programs execute in Android system.

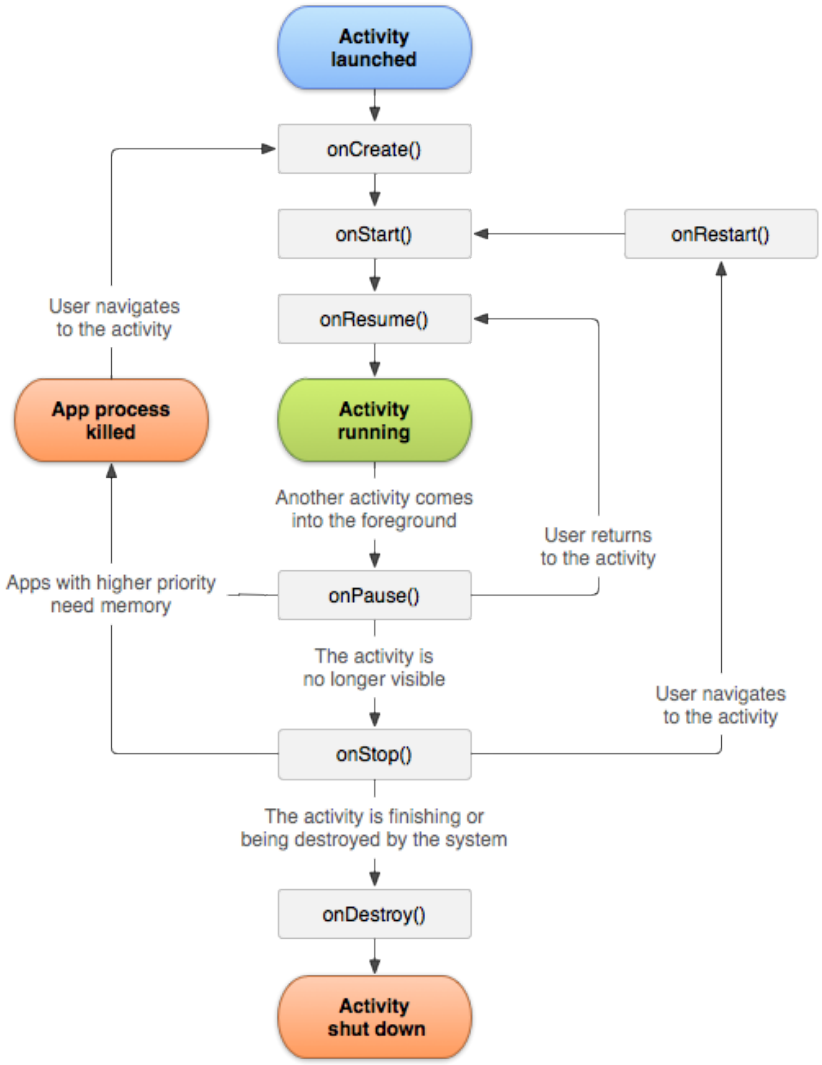

Figure 14: Activity lifecycle model (from `http://developer.android.com/reference/android/app/Activity.html`).

Between activities, intent is used to switch from current view to target view. For example, using "`newIntent(WelcomeActivity.this,SettingActivity.class)`" in `WelcomeActivity`, an Intent object is created to establish the transfer from `WelcomeActivity`.

`this` to `SettingActivity.class`. Another example of intent is to pass extra parameters (key-value) along with the transfer process. As "`intent.putExtra("M",M);intent.setClass(SettingActivity.this,MapTracking.class);`" indicates, a M value with name "M" is passed to `MapTracking`.

4.2.1 CO_2 Emission Calculation

The real-time CO_2 emission amount shown on the map view is calculated by multiplying CO_2 emission coefficient for unit length with the current distance. The consumption of fuel and the amount of CO_2 emission is used in calculation as the formula [4] below:

$$CO_2\ emission\ amount = 0.2407kg/kWh \times Energy\ of\ fuel\ used$$

$$Energy\ of\ fuel\ used\ = \frac{Distance\ traveled}{Distance\ per\ unit\ of\ fuel} \times Energy\ per\ unit\ of\ fuel$$

So current CO2 emission amount is the function of accumulated distance multiplied by a single coefficient, with the assumption of no transportation transfer. In the more complex situation, transfer is needed and logged, so the coefficient value should be changed accordingly, as well as the route color and transfer marker. A further specification is discussed in MapTracking activity.

But another other question emerged: how to obtain the coefficient for various vehicles. It is viable to implement a table of corresponding coefficients for vehicle models, but we choose to implement a coarse mapping between CO_2 emissions and distances for each transportation mode. A practical estimation of the coefficient provided by French Environmental Agency is shown in Table 1. We notice that there are various results on the Internet for each entry of Table 1, as statistics is different depending on references.

Transportation mode	CO_2 emission
Subway	3.3 g/(traveler·km)
Bus	100 g/(traveler·km)
Train	43 g/(traveler·km)
Car(extra-urban roads)	85 g/(traveler·km)
Car(urban roads)	149 g/(traveler·km)
Motorcycle	125 g/(traveler·km)
Walking	180 g/(traveler·km)
Bike	75 g/(traveler·km)

Table 1: CO_2 emission per transportation mode [7] provided by French Environmental Agency.

4.2.2 WelcomeActivity

The class `WelcomeActivity` is responsible to create a view of welcome page and switch to class SettingActivity after a predefined time delay.

```
new Handler().postDelayed(new Runnable(){
    @Override
    public void run() {
        startActivity(new Intent(WelcomeActivity.this , SettingActivity.
            class));
    WelcomeActivity.this.finish();
  }
}, SPLASH_DELAY_TIME);
```

The code snippet above is part of `onCreate(BundlesavedInstanceState)` method, the delay constant `SPLASH_DELAY_TIME` is set to 3000 ms. Method `booleanandroid.os.Handler.postDelayed(Runnabler,longdelayMillis)` will execute `Runnable` after delay constant. Intent object is created to establish the transfer from `WelcomeActivity.this` to `SettingActivity.class`, then `startActivity` launches the new Activity (i.e. `SettingActivity.class`). `finish()` is a method of Activity to terminate this activity and conduct garbage collection.

4.2.3 SettingActivity

SettingActivity class is responsible for both getting & sending out the selection of "M" coefficient (used in CO_2 calculation) and controlling the right to start new activity (MapTracking).

In the onCreate method, transportationGroup (list of radio buttons) and start exit buttons are bundled with the corresponding items defined in layout file. Then listeners monitoring the onClick onCheckChange events are registered:

```
...
startButton.setOnClickListener(new MyButtonListener());
stopButton.setOnClickListener(new StopButtonListener());
transportationGroup.setOnCheckedChangeListener(new MyGroupListener());
...
```

When the radio button in transportationGroup changes, `onCheckedChanged(RadioGroup, int)` method of MyGroupListener class is invoked, assigning M coefficient corresponding value classified by transportation mode ids. And when the start button is clicked, the listener invokes the `onClick(View)` method in MyButtonListener class, in which the M value with key "M" is passed to the new activity `MapTracking.class`. Note that M value is the coefficient of multiplication between carbon emissions and distance, simplified from the formulas and data table in Section 1.

```
...
Intent intent = new Intent();
intent.putExtra("M", M);  // assign M value to counterpart in
    MapTracking
intent.setClass(SettingActivity.this, MapTracking.class);
SettingActivity.this.startActivity(intent); // trigger MapTracking
    activity
...
```

4.2.4 MapTracking

MapTracking is a big class that extends MapActivity and implements LocationListener and OnInitListener. The body of MapTracking pertains to location service, drawing on overlays, TTS definition and instances, Shake detection interface, option menu issues, calculation of distance and carbon emission, and `onLocationChanged(Location)` method.

Class LocEntry is an important implementation in MapTracking (see Figure 15). We found all the subtasks mentioned above focus on the GPS location information. Location is a well defined class storing complete GPS information (latitude, longitude, altitude, accuracy, bearing, speed, time, service provider, etc.) on certain location. So we designed the class LocEntry storing both Location object received for a specific location and essential attributes used for CO2 statistics (the local timestamp, distance and CO2 emission differences). It is feasible to initialize a LocEntry object using LocEntry() method, but we use setXXX()-like methods to assign values to its attributes along the MapTracking process.

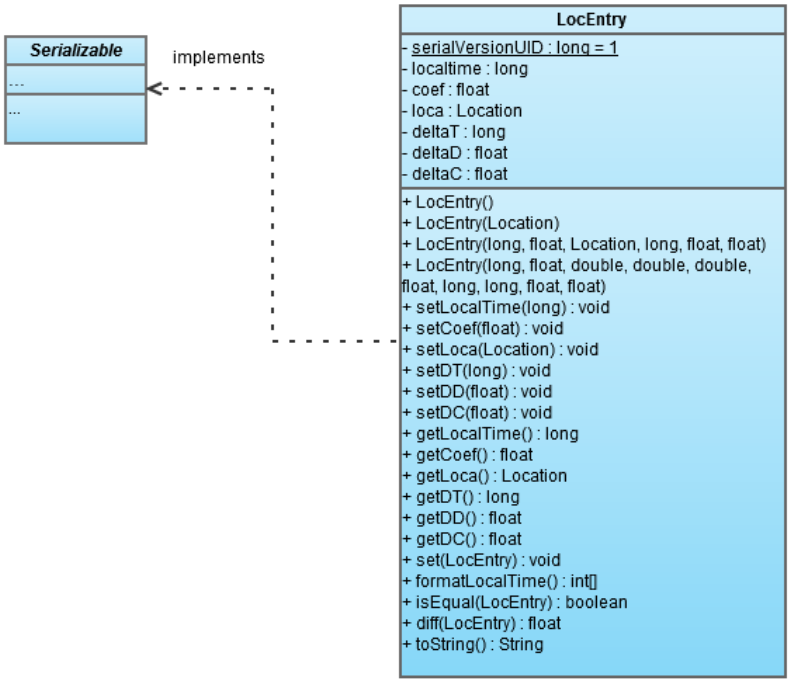

Figure 15: LocEntry class diagram for storing Location information and interacting with database.

We need mention the special methods in LocEntry as Figure 15 shows: set(LocEntry) is

used for getting assigned from another LocEntry object, and is used in exchanging contents between currentLocEntry and lastLocEntry; formatLocalTime() returns "hh:mm:ss" format of timestamps; isEqual(LocEntry) is a boolean test whether two LocEntry share the same timestamp, used for testing the first point in tracking process by comparing currentLocEntry and lastLocEntry; diff(LocEntry) returns the difference of distance, used for updating the attribute of "deltaD" of currentLocEntry.

To thoroughly understand the structure and relations within MapTracking class, we need refer to the lifecycle model of Activity, with MapActivity supports the same model (controlled by Android system). Upon starting Mapview, it seems complex since we implemented the array of instructions into separate methods. However, we can figure out the order of initiation following the line of execution, i.e., onCreate()→onResume(). The execution flow is as below:

1. onCreate(): set database handler and timer handler, initialize currentLocEntry and lastLocEntry with null Location.
2. onCreate()→mapControl() method: configure mapView, initiate mapController, and "`locationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE);`"
3. Back to onCreate(): get coefficient "M" value bundled with color and icon information from SettingActivity, setting TTS, prepare vibrator service, and start a shakeListener instance.
4. onResume()→init(): get the last known point as seed location with "`currentLocation=locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER)`", used for initializing a LocEntry object.
5. init()→initMyLocation(): set up an instance of MyLocationOverlay, then runOnFirstFix(new Runnable()) centers on the first point on myLocationLay using animateTo method, and add myLocationLay to mapView.
6. Back to init(): set flag indicating the initialization is done.
7. Back to onResume(): set location listener in `locationManager.requestLocationUpdates(LocaGPS_PROVIDER,2000,6,MapTracking.this)`, the updating period is 2000ms. If GPS is down, use coarse location service instead.
8. onLocationChanged(Location): triggered by LocationListener. Since MapTracking implements LocationListener, there is no need to explicitly register the listener. Get the first location, converse the coordinate, draw the route from lastLocation to currentLocation on overlay which is then added to mapView, calculates the distance and carbon emission value based on "M" value, update the visual reading in timer task, add currentLocEntry to the database in an AsyncTask, and check carbon emission threshold for TTS output. Each time onLocationChanged(Location)is triggered, lastLocEntry and currentLocEntry are shifted enabling the next step of drawing route.

Here we need to specify the use of AsyncTasks. We previously defined AddEntryTask and UpdateEntryTask extending AsyncTask class, and implemented addLocEntry() and updateLocEntry() respectively in doInBackground() method. Every time the current location

is changed, the currentLocEntry is add to the database; and when transfer of transportation mode is triggered, the coefficient of current entry in database is updated. Both async-tasks are done in the background threads, avoiding the block of the main process (current activity) when writing the database.

4.2.5 Text-to-Speech

The Text-to-Speech (TTS) engine in Android is not widely used, but quite suitable to augment the usability of our application. The main concern of adopting TTS engine is safety in transportation: we need speech feedback triggered by shaking to provide the undistracting experience esp. when driving a car, and TTS engine is much more customizable than static audio records. And we use TTS to provide speech output for both shaking triggered alerts as well as the active alerts upon exceeding the threshold.

Given references [1], it's generally easy job to insert TTS methods into MapTracking class. Both active and passive speech feedbacks calls for TTS methods in MapTracking. Here is the process: firstly check whether the TTS data is installed or not, as TTSIntent and onActivityResult do; then onInit method is invoked to initial TTS engine by setting its language; finally we can use TTS.speak method to transfer strings to speech.

```
    Intent TTSIntent = new Intent();
    TTSIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
    startActivityForResult(TTSIntent, MY_DATA_CHECK_CODE);
...
protected void onActivityResult(int requestCode, int resultCode, Intent
    data)
...
public void onInit(int initStatus)
...
public void speaking(String speech) {
        TTS.speak(speech, TextToSpeech.QUEUE_FLUSH, null);
    }
...
speaking("Your current distance is" + String.valueOf(NEWSumDistance) + "
    meters" + "And your current CO2 emission is" + String.valueOf(NEWCO2M
    ) + "grams");
...
```

4.2.6 ShakeListener

ShakeListener class has no corresponding layout files, and is initialized with methods invoked inside the MapTracking class when shaking event is recognized, so an interface is implemented as OnShakeListener, and the constructor has an input value Context, convenient for passing Context variable in MapTracking. According to the lifecycle model of Activity, the constructor and onResume() are responsible for setting sensors and registering listeners: retrieve a SensorManager for accessing sensors, then get default sensor of type ACCELEROMETE, and finally Register ShakeListener for given sensor.

```
...
```

```
public ShakeListener(Context context) {
        this.context = context;
        sensorManager = (SensorManager)context.getSystemService(Context.
            SENSOR_SERVICE);
      onResume();
    }

public void setOnShakeListener(OnShakeListener listener) {
      onShakeListener = listener;
    }
@Override
protected void onResume() {
      sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
          ;
      if(sensor != null) {
         sensorManager.registerListener(this, sensor, SensorManager.
             SENSOR_DELAY_GAME);
      }
    }
...
```

Afterwards, `onSensorChanged(SensorEventevent)` as the kernel part of shaking-detection algorithm is simple and concise, if one understands the coordinate of the accelerometer. We consider shaking behavior back and forth as a sharp changing of acceleration in direction, meaning an obvious difference between two consecutive sampling points in a data array. So how can we present the difference? Each time upon update, we get the delta values (dX, dY, dZ) from buffered acceleration for each component in 3-dimensional coordinate. Then a new delta value is calculated with the formula

$$dA = \sqrt{dX^2 + dY^2 + dZ^2}$$

Finally, we compare dA with threshold DELTA_THRESHOLD: if delta value is bigger, i.e., the sharp change of acceleration is detected, the interface `onShakeListener` will invoke `onShake()` method. `onShake()` method is defined in MapTracking class, containing both visual feedback method show() in Toast and TTS speech feedback instance speaking(..).

After illustrating the general algorithm of shaking-detection, we need to find a suitable value for threshold. This is done by inserting a debug logging instruction `"Log.d("sensor", "Currentdeltaacceleration:"+String.valueOf(dA));"`, which logs debug messages with suggested segments. Here is a snapshot of debugging messages. From the array of messages below, we can well decide the level of threshold, neither too sensitive nor too negative.

```
...
11-26 21:51:32.736: D/sensor(6512): Current delta acceleration
    :0.06864643112160147
11-26 21:51:32.807: D/sensor(6512): Current delta acceleration
    :0.2843923526751469
11-26 21:51:32.924: D/sensor(6512): Current delta acceleration
    :0.3633754156980968
11-26 21:51:33.057: D/sensor(6512): Current delta acceleration
    :1.5720031140032105
```

```
11-26 21:51:33.182: D/sensor(6512): Current delta acceleration
    :2.3172346566894335
11-26 21:51:33.307: D/sensor(6512): Current delta acceleration
    :1.722512583029177
11-26 21:51:33.432: D/sensor(6512): Current delta acceleration
    :2.443607601299943
11-26 21:51:33.572: D/sensor(6512): Current delta acceleration
    :0.6311399703869192
11-26 21:51:33.682: D/sensor(6512): Current delta acceleration
    :0.9440909129933476
11-26 21:51:33.807: D/sensor(6512): Current delta acceleration
    :11.139029240113762
11-26 21:51:33.939: D/sensor(6512): Current delta acceleration
    :9.153004354226873
11-26 21:51:34.057: D/sensor(6512): Current delta acceleration
    :54.30645823702739
11-26 21:51:34.135: D/sensor(6512): Current delta acceleration
    :66.07283209175311
11-26 21:51:34.252: D/sensor(6512): Current delta acceleration
    :4.297596264038962
11-26 21:51:35.182: D/sensor(6512): Current delta acceleration
    :1.7857798482575815
11-26 21:51:35.307: D/sensor(6512): Current delta acceleration
    :0.5675159764246626
11-26 21:51:35.439: D/sensor(6512): Current delta acceleration
    :1.1475453966589682
11-26 21:51:35.564: D/sensor(6512): Current delta acceleration
    :0.23637896377159073
11-26 21:51:35.690: D/sensor(6512): Current delta acceleration
    :0.2249120911421012
11-26 21:51:35.814: D/sensor(6512): Current delta acceleration
    :0.8321217590676919
11-26 21:51:35.940: D/sensor(6512): Current delta acceleration
    :0.6446334064328296
...
```

4.2.7 Database & Data Visualization

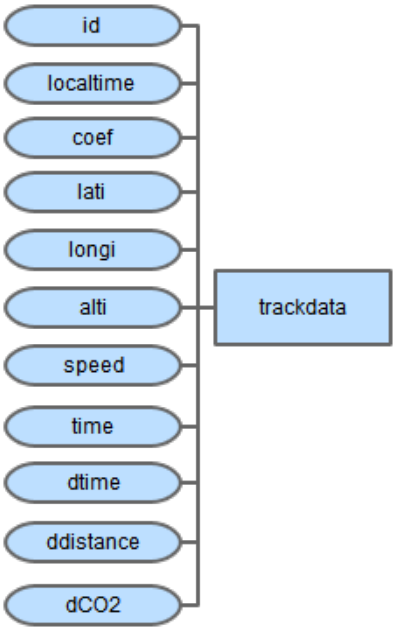

Figure 16: Entity-relational model of database mytrack.sqlite.

The E-R model in Figure 16 presents the structure of the database for logging (refer to `DBHandler.java`). Actually, the database handler class is designed for easily manipulating LocEntry objects in the database "mytrack.sqlite". There are common instructions "ACUD" (i.e. add, create, update, delete) interacting with Sqlite databases. We also specially implemented class ArrayPair to acquire and format the columns of "localtime" and "deltaC/deltaT" into two Number arrays, which were further used in data visualization in SummaryActivity.

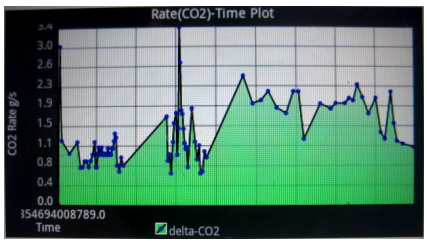

Figure 17: CO2 emission rate chart based on database, plot with AndroidPlot library.

In SummaryActivity, We used AndroidPlot library to implement CO2 emission rate chart. As we can see from Figure 17, the y-axis is "deltaC/deltaT", i.e. the CO2 emission rate, and the x-axis is long integers indicating absolute timestamps in UNIX time. The implementation details refer to the docs in AndroidPlot library (`http://androidplot.`

com/docs/how-to-plot-against-time/). There are also text statistics on total & average values of distance & CO2 emissions.

4.3 Debugging & Testing

As we has mentioned, Emulator does support for debugging a virtual device which imitates the response of target real device. However, only manually input data sets are accepted which makes debugging sensor-assisted application on an emulator unpractical. We recommend using wired smartphone in debugging mode to test the response under real circumstances. Below is the snapshot of an instance of debugging logs. We use the static method "Log.d(tag,msg)" to generate debugging log messages which are routed to debugging output.

```
...
11-26 10:23:47.963: D/CO2M(29284): the sum NEWCO2M:3.0
11-26 10:23:47.963: D/distance(29284): the current getdistance:21.449259
11-26 10:23:47.963: D/distance(29284): the sum getdistance:21.449259
11-26 10:23:47.971: D/location(29284): the lastlocation is:Location[
    mProvider=gps,mTime=1353918176417,mLatitude=60.26708006858651,
    mLongitude=24.984626770018806,mHasAltitude=true,mAltitude=32.0,
    mHasSpeed=true,mSpeed=0.0,mHasBearing=true,mBearing=259.0,
    mHasAccuracy=true,mAccuracy=98.34706,mExtras=Bundle[mParcelledData.
    dataSize=4]]
11-26 10:23:47.971: D/location(29284): the currentlocation is:Location[
    mProvider=gps,mTime=1353918178446,mLatitude=60.26709079742257,
    mLongitude=24.98501300811695,mHasAltitude=true,mAltitude=25.0,
    mHasSpeed=true,mSpeed=0.0,mHasBearing=true,mBearing=259.0,
    mHasAccuracy=true,mAccuracy=98.34706,mExtras=Bundle[mParcelledData.
    dataSize=4]]
11-26 10:23:50.057: D/dalvikvm(29284): GC_CONCURRENT freed 618K, 47%
    free 3674K/691 9K, external 3072K/3776K, paused 2ms+4ms
...
```

We have witnessed from emulator results that we have got very nice but theoretical results (esp. routes by GPS) through MapTracking and manual input data set. Nevertheless, it is indispensable to conduct "field test", i.e. testing and logging with GPS in the wild field. Since self-logging module is not available temporarily, we need take laptops with debugger & logger with target smartphone to experience real transportation conditions. Figure 18 shows the snapshots of our testing activity in a real test (around Kumpula campus), validating the features of CO2EmissionAlert in practical environment. The other two snapshots in Figure 19 present another experiment we conducted after the demo day, when we used a Motorola ME525 to test transferring between bus and walking, within a short distance near Kumpula.

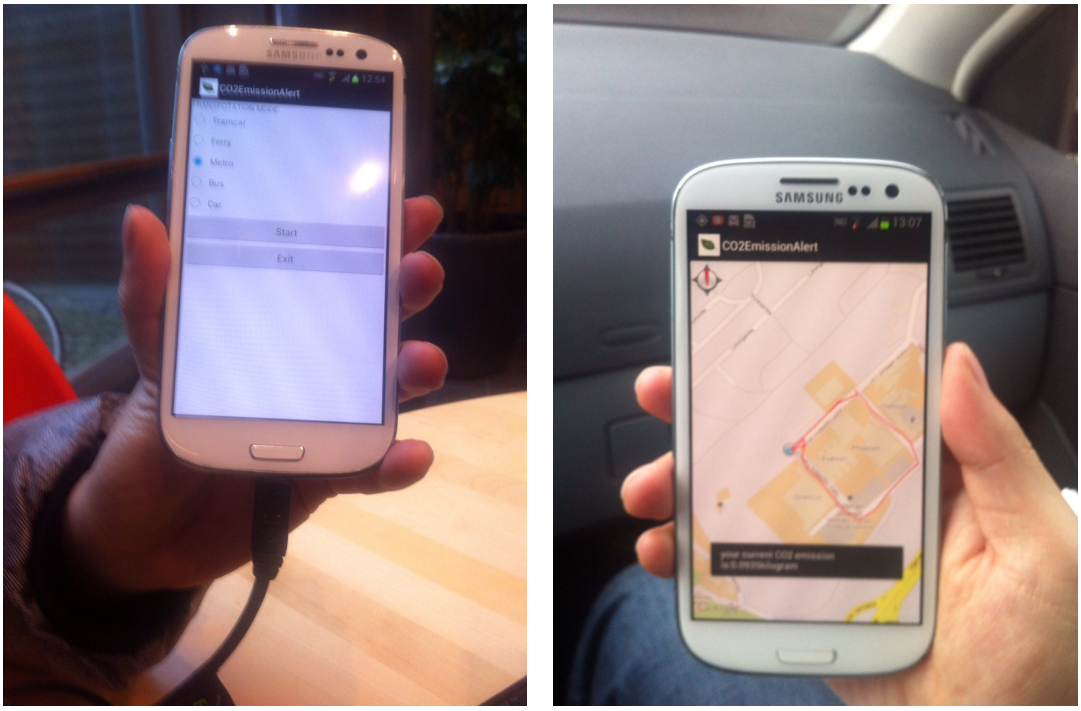

Figure 18: Snapshots on smartphone screen in real test1 (from a video demo with Samsung Galaxy S3).

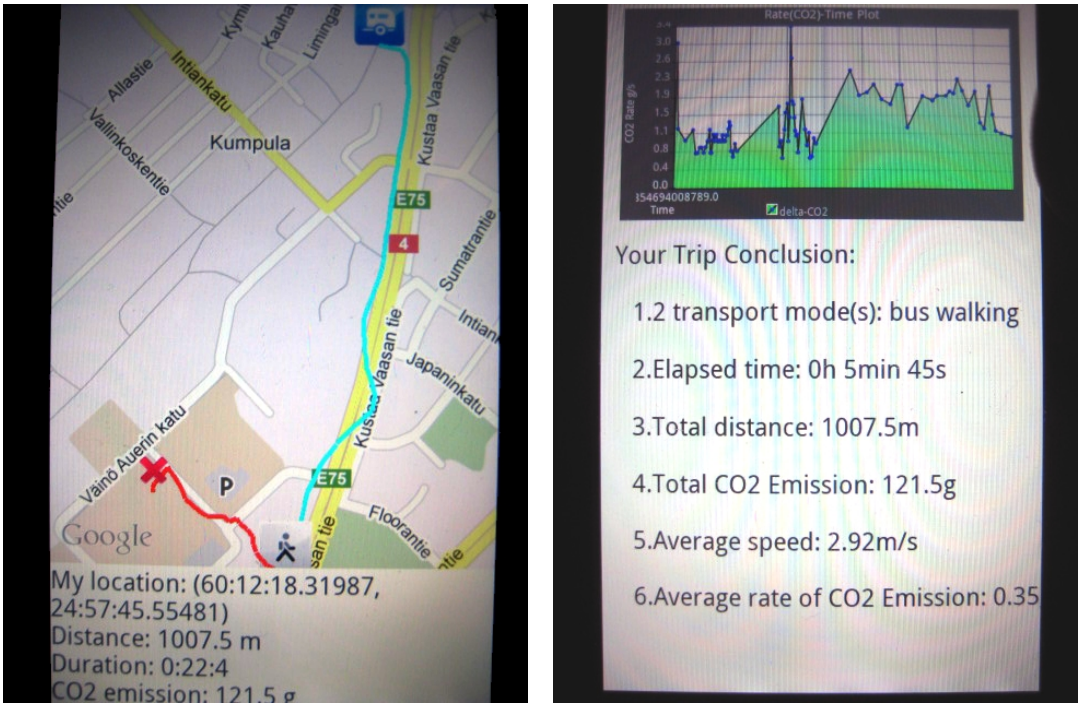

Figure 19: Snapshots on smartphone screen in real test2 (a simple case with Motorola ME525).

We had to admit that noise and fluctuation is obvious to some extent, thus the error range is expanded. Based on the logging data from the real test, actually, we are trying to adopt some sort of filters to attenuate the fluctuation eliminating sharp jump points. So far, the moving average filter is considered effective in theory, which buffers an queue of history data rapidly generated by GPS receiver and evaluates the average value on the updating queue. However, applying this algorithm in a 2-dimension plane cast many questions, calling for further references. Another phenomenon was the "transient state problem": the GPS returns random points within an area of 100+ meters when the smartphone was not in motion (most often when the smartphone was static, e.g. waiting for bus transfer at a bus stop). In this sense, we designed a sensor-assisted strategy which utilizes accelerometers and gyrometers to recognize static scenario, in which GPS data is blocked from overlay methods.

4.4 Facing Problems

We are all invoices for Android development, so there were big and small obstacles to get over. To demonstrate the category of problems (technical and collaboration) and our efforts in solving problems, we present two examples.

Problem I: The first one is a technical problem: when developing MapTracking methods, we met with the difficulty of how to draw the real-time trace of motion on MapView. There was no direct methods to implement such sort of curves.

Solution: We discussed this problem on a group meeting. Then we all agreed on drawing on any overlays over the MapView. To tackle the curve instead of straight line, each of us provides advice based on the references he found himself, and shares it with each other. As a result, we decide to adopt differential theory to our method. So a chain of straight lines form the estimation of the curve trace. And it's much easier to plot straight lines between two ends on an overlay. But a further problem is: how to retrieve the first location on the map, since no "lastLocation" exists beforehand. So we tried to shift the last point and current point for each update, in which way the plot route moves forward step by step.

Problem II: About collaboration. Since we all lack of the experience of collaborative group working in a project, we met with little but many difficulties in the process. Communication is really important in the groupwork. The lesson was learned from the experience of developing objects following different conventions, resulting in troubling interfaces and the delay of schedule.

Solution: Group meeting can be anytime, just convenient for exchanging ideas and eliminating misunderstanding. Summary after the group meeting can be used to prompt control of schedules. Cooperation can be improved only by communication.

4.5 Further Plan

In our GitHub repository `https://github.com/mrr7rhq/CO2Alert`, there are already versions of iteration results for CO2EmissionAlert, with project source files and ReadMe manual. The version 1.0 focused on MapTracking algorithms, the problems of real-time multi-modal feedback (visual toast and speech reading) and drawing routes are well solves. Version 1.1 made efforts to check through the algorithms in MapTracking and ShakeListener classes, added support for transportation transfer, keeping screen on function and option menu to reset/clear overlays and accumulators, and active emission alerting. The latest version 1.2 updated the UIs and added database and data visualization features.

In the planned version 2.0 in the future, we decide to add the following functions:

1. The feature of automatically detecting transportation environment. We can combine GuessDroid project presented on demo day to implement the machine learning methods.

2. The feature of social sharing of carbon records after the summary view. We can add a remote server to store massive data of carbon records enabling Facebook interactions.

3. The functionality of sensor-assisted GPS sensing, which utilizes accelerometers and gyrometers to detect static scenario (no motion), in which GPS data can be blocked from drawing overlays. This strategy is mentioned in the discussion after practical test, and seems easier for implementing than the moving filter method.

5 Discussion

In this project, we had made efforts in designing and developing the Android geo-location application, by applying multiple interface technologies to the project. We had completed the complete functions of active and passive speech feedback, the visual indicator on mapView along the route, satisfactory tracking results proved in the real test. We have met with big or small troubles as mentioned above, most of which calls for everyone's efforts and the cooperation and collaboration. It's nice to witness our project approaching to the final destination – a sign of success.

Besides the application itself, each of us have learned a lot from the project: the knowledge of developing interface related applications, and the knowledge of collaboration and communication in group development. We decide that version control is also essential esp. for rapid iteration strategy. At first we had built the project files on local version control system, but soon we found the feasibility of GitHub, where we are moving project files to. Another issue that weighs is time scheduling: it can be more than trouble delaying the schedule in a group work, since every participants are influenced. Anyway, it had not turn out a disaster for us.

There is still a problem of updating requirements: it's the reason for adding new features – we can often get better ideas on "what if ...". In this sense, new features like the summary view were actually generated from one of the group meetings.

6 Conclusion

We developed CO2EmissionAlert, a lightweight Android application to inspect daily carbon footprints especially for transportation process. It features real-time tracking on map, multi-modal feedback with visual and speech alerts, active alert upon threshold, and shake-triggered feedback. Analysis of the logic and algorithms enlighten us on development of interactive applications using interface technologies.

In this report, we present mainly the features and methods in designing and developing CO2EmissionAlert project. Besides, evaluation and reflection on our project provides new advice on how to improve similar projects and behavior of ourselves. In our demonstrative experiments, the implemented modules (e.g., MapTracking, SummaryActivity) of CO2EmissionAlert have worked to prove a successful project.

It was a successful project not only because of the application itself, but the knowledge and experiences we have learned from the groupwork. We met with problems on development and collaboration, which were conquered finally in various methods. It was truly valuable experience attending the Interface Technology course.

References

[1] Android SDK: Using the Text to Speech Engine. `http://mobile.tutsplus.com/tutorials/android/`

android-sdk-using-the-text-to-speech-engine/.

[2] Co2 Emissions - Wikiprogress.org. http://www.wikiprogress.org/index.php/Co2_Emissions.

[3] CO2GO. http://senseable.mit.edu/co2go/.

[4] Andrew Austin. Eco Driving Tutor Application for Android. *lancs.ac.uk*.

[5] Bojin Liu, Dipak Ghosal, Yachao Dong, Chen-Nee Chuah, and Michael Zhang. CarbonRecorder: A Mobile-Social Vehicular Carbon Emission Tracking Application Suite. In *2011 IEEE Vehicular Technology Conference (VTC Fall)*, pages 1–2. IEEE, September 2011.

[6] Vincenzo Manzoni, Diego Maniloff, Kristian Kloeckl, and C Ratti. Transportation mode identification and real-time CO2 emission estimation using smartphones. Technical report, 2010.

[7] C RATTI and K KLOECKL. SYSTEM FOR DETERMINING CO2 EMISSIONS. http://patentscope.wipo.int/search/en/WO2012094465, 2012.