Software Design Document

Virtual Reality for Sensor Data Analysis

Project: Virtual Reality for Sensor Data Analysis
Author: Gero Birkhölzer, Johannes Blank, Alexej Gluschkow, Fabian Klopfer, Lisa-Maria Mayer
Last Change: July 15, 2017 , Version 2.0

Contents

1 Purpose

The software project in the summer term 2017 at the University of Constance focuses on the development of apps for mobile devices. In the course of the project an Android app is being developed which allows the user to explore sensor data in virtual reality.

Especially, this Software Design Document intends to describe the internal structure of the app as well as test cases for the requirements stated in the Requirements Specification.

1.1 Product Idea and Goal

The general idea of the product is to allow the user to record data about their environment and later explore the data in a three-dimensional scene via virtual reality. Therefore, the developed product will consist of two parts:

Firstly, the app itself. It's main goal is to connect to an external sensor device via Bluetooth and to process and save the data collected by the sensor (referred to as "app").
In order to view the saved data, the second part consists of a web application where a virtual reality scene is generated and the stored data are visualized (referred to as "web application").

These two parts will be connected in such a way that the user can open a browser with the according web application from within the app.

1.1.1 Use Case

A possible scenario for using the product would be the following:

The Faculty of Sports of the University of Constance would like to have the universitiy's gym renovated. For this, they need prove that the gym actually is in need of renovation.

One problem amongst others is the fact that the ventilation system doesn't work properly: During heavy use, the air temperature rises significantly.
With our product, they can track the temperatures inside the gym at different locations and present them to the people responsible for granting the renovation in a visully appealing way. Thereby, they can proof that the athletes' area becomes exceptionally warm.

1.2 Definitions

App When the app itself is mentioned, we refer to the application running on the smartphone that, as stated above, handles the recording of data and invokes a web browser with the web application.

Web application This term refers to the web site that can be invoked by the app, runs in a web browser, and provides the virtual reality display of the data gathered by the app.

Sensor (device) When referring to the sensor, we're talking about the sensor device (more clearly specified in section 2) containing several sensors.

Data With data we generally mean information that has been gathered by the sensor.

Virtual Reality (scene) This term describes the three-dimensional world in which the data will be displayed.

1.2.1 Abbreviations

TI Texas Instruments

VR Virtual Reality

3D three-dimensional

DB Database

App Application

BLE Bluetooth Low Energy

GATT Bluetooth Generic Attribute Profile

Characteristic Bluetooth Generic Attribute Profile Characteristic

1.2.2 Glossary

Stereoscopic 3D The impression of 3D is created by rendering different pictures for every eye of the viewer.

Virtual reality By using a headset in which the smart phone can be integrated, the user can view the three-dimensional world in stereoscopic 3D and thereby experiences the feeling of being fully immersed in the scene.

Augmented reality Displaying 3D objects in a real-world surrounding while providing an immersive experience like virtual reality.

Gyroscope sensor Sensor for measuring orientation in space.

Web application Web site that offers functionalities similar to those of "normal" desktop or mobile applications but runs in a web browser.

Service From AndroidDoc: "A Service is an application component that can perform long-running operations in the background, and it does not provide a user interface".

GUI From AndroidDoc: "They (Activities) serve as the entry point for a user's interaction with an app, and are also central to how a user navigates within an app (as with the Back button) or between apps (as with the Recents button)".

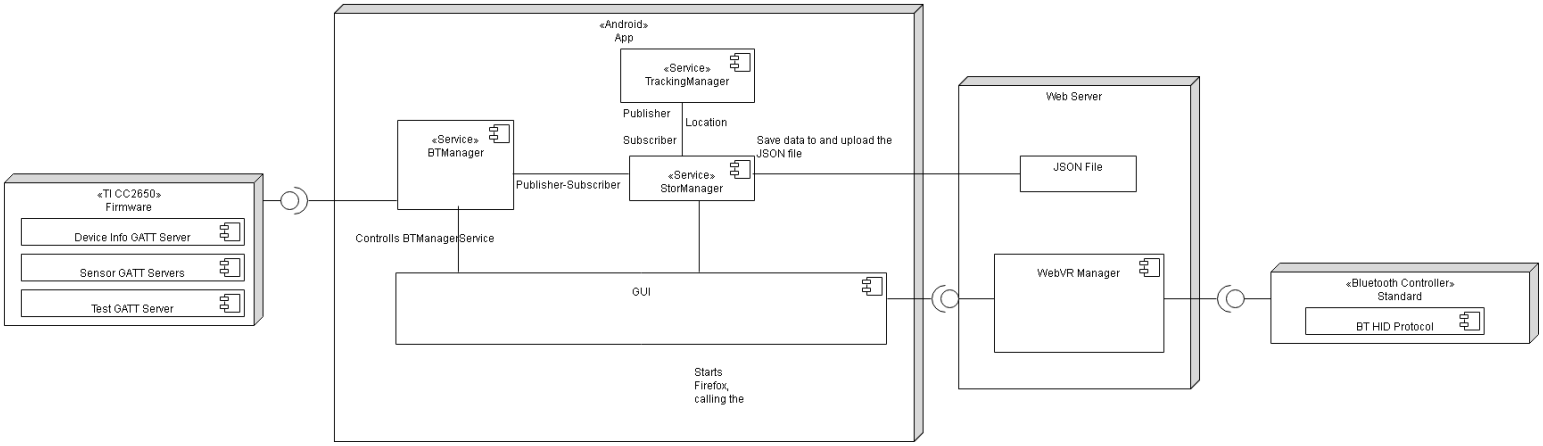
1.2.3 References

Requirements Specification Version 2.0 from May 15th, 2017.

2 Proposed Architecture

A better zoomable representation of these diagrams can be found in the github repository of this project in /doc/pflichtenheft/pics, where also the xml sources are.

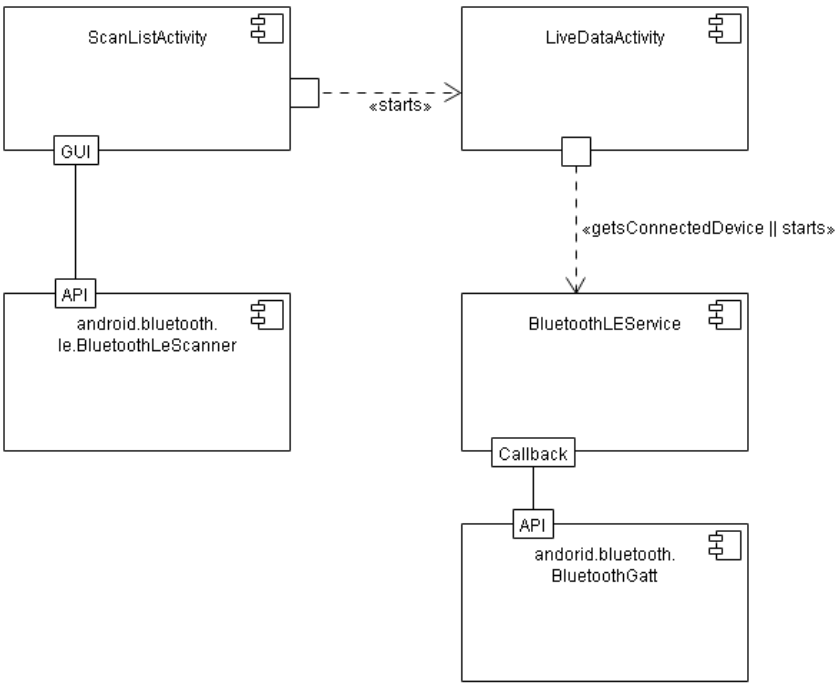
2.1 Overview



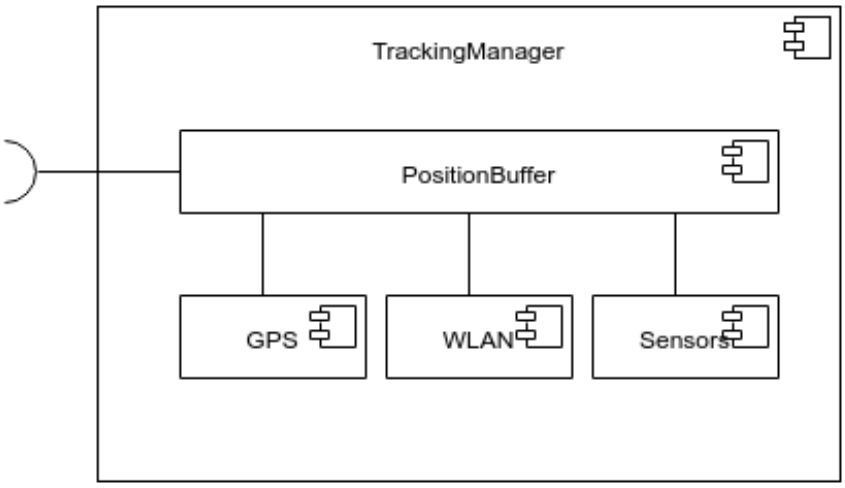
2.2 Component Decomposition

2.2.1 Services

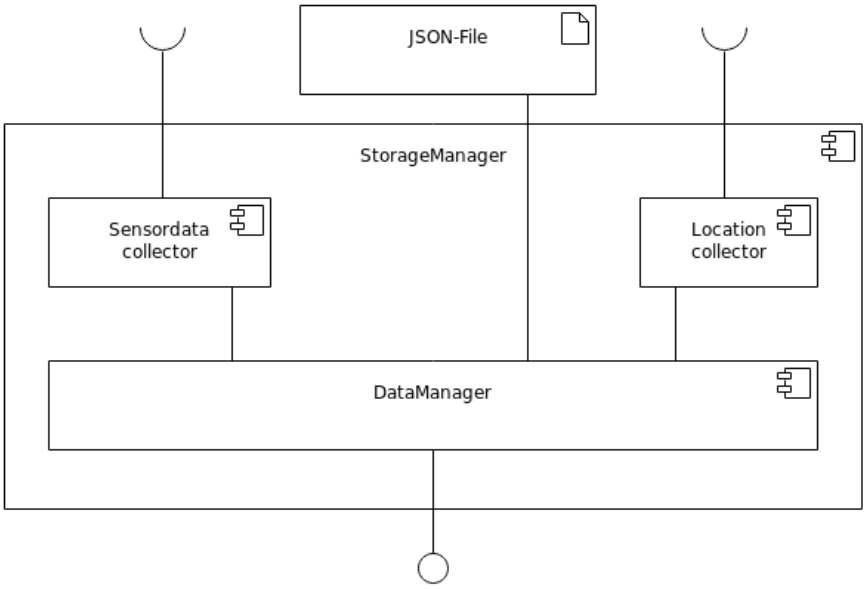
- **BluetoothLEService:** Uses the android.bluetooth and especially the android.bluetooth.le libraries to connect to a TI CC2650 MCU (sensor device), fetch the sensor data from the sensor device, to convert those values and broadcast them.



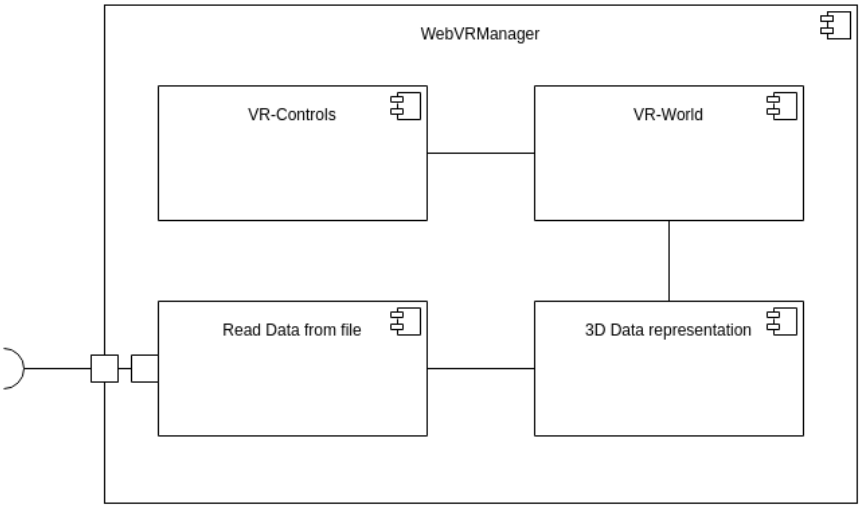
- **TrackingManager:** Handles the tracking of the (current) location where the data is recorded. The current position is determined by GPS and enhanced by the smartphone wifi data.



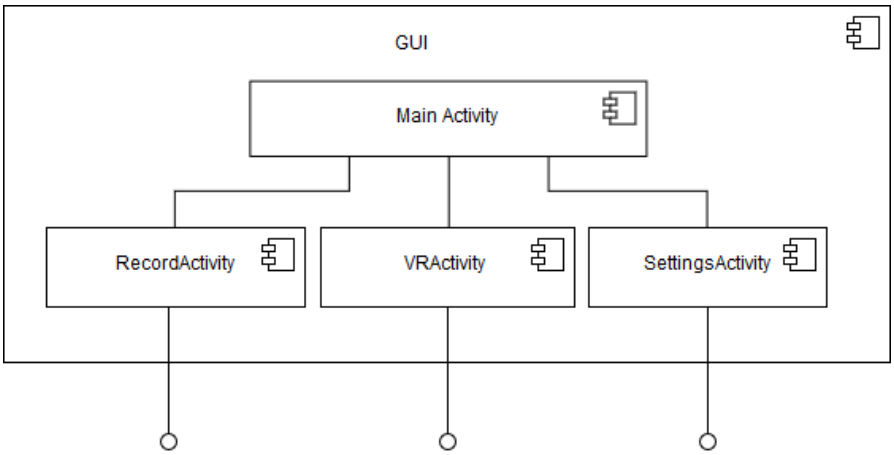
- **StorageManager:** Processes the data provided by the TrackingManager and the BluetoothManager. Uses a JSON file to store data and uploads them to a webserver where the web application can access them.



- **WebVRManager:** Handles the display of the virtual reality scene and the given data from the sensor.



2.2.2 GUI



- **SplashActivity** Provides the main startup screen as the main entry point.
- **MainActivity** Provides the main startup screen as the main entry point.
- **RecordActivity** Provides an user interface to record new data.
- **SessionActivity** Provides an user interface to record new data.
- **LiveDataActivity** Shows the data that are received in real time using the android databinding api
- **SettingsActivity** Provides a user interface to configure the connectivity to the sensor device and the tracking options.
- **ScanListActivity** Used to start the scan and to let the user initiate the connection routine(s)

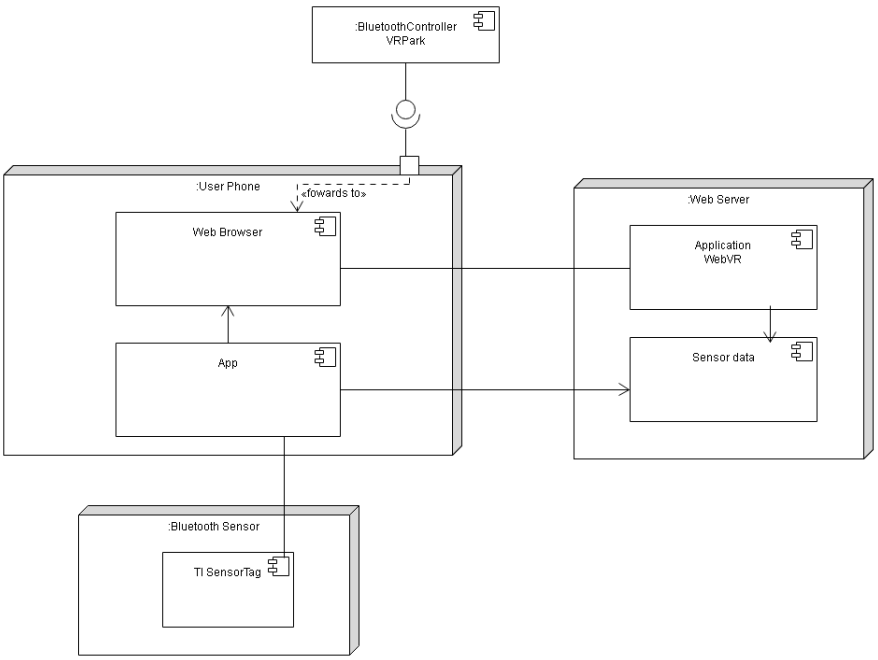
2.2.3 Additional Classes

- **TIUUIDs** GATT TI CC2650 Service UUIDs
- **Sensor** Parser and helper functions for each sensor because the BLE protocol implemented in the TI CC2650 delivers raw sensor output
- **SensorDataListAdapter, LeDeviceAdapter, ScanListItem, DataListItem** Adapters, data- and view holders for the databinding API

2.3 Hardware/ Software mapping

The product consists of two parts which both run on the smart phone:
Firstly, the app will connect to the bluetooth sensor, gather the data and store them on a web server.

Secondly, the web application will also run on the smart phone and will be executed in a web browser which can be started by the user from within the app. The web application will show a previously built scene and display the gathered sensor's data in virtual reality.



2.4 Global software control

2.4.1 Startup behavior

User: The user can start the app by pressing the icon on his screen. The app will show a splash screen while loading and then transition into the main menu, from where the user can start the other functions of the app.

2.4.2 Interfaces

The **BluetoothManager** provides a possibility to connect to a sensor and then record the data. It ensures that a connection is established and that the data are broadcast via a local BroadcastManager.

The following listing contains the names of the intents and which extras they contain.

```
public enum Sensor {

    /**
     * Constructor and main fields;
     * there is also a constant byte code "ENABLE_CODE" = 0000 0001
     * @param name the name of the according sensor
```

```
         * @param service The UUID of the GATT Service of the sensor
         * @param data The UUID of the Characteristic of the sensor data
         * @param config The UUID of the configuration characteristic
         */
        Sensor(String name, UUID service, UUID data, UUID config);

        /**
         * Used to convert the received binary values into human readable
         * form
         *
         * @param the raw byte value of the received Bluetooth
         * characteristic
         */
        public float convert(byte[] value) {
         requires value != null
           ensures correctly conversion from byte to value in human readable
           form
        }

    }
```

Below, the interfaces of the ServiceConnection that is used to bind the service are listed.

```
    /**
     * This abstract class is used to implement {@link BluetoothGatt}
     * callbacks.
     */
    public class BluetoothLEService extends Service {

        /**
         * Methods overridden from the interface android.app.service
         */

         /**
           * Called by the system every time a client explicitly starts
           * the service by calling
           * {@link android.content.Context#startService}, providing the
           * arguments it supplied and a unique integer token
           * representing the start request.  Do not call this method
           * directly.
           *
           * <p>For backwards compatibility, the default implementation
           *mcalls {@link #onStart} and returns either
```

```
     * {@link #START_STICKY} or {@link #START_STICKY_COMPATIBILITY}.
     *
     * <p>If you need your application to run on platform versions
     * prior to API level 5, you can use the following model to
     * handle the older {@link #onStart} callback in that case.
     * The <code>handleCommand</code> method is implemented by
     * you as appropriate:
     *
     * {@sample development/samples/ApiDemos/src/com/example/
     * android/apis/app/ForegroundService.java start_compatibility}
     *
     * Note that the system calls this on your service's main
     * thread. A service's main thread is the same thread where
     * UI operations take place for Activities running in the same
     * process. You should always avoid stalling the main thread's
     * event loop. When doing long-running operations, network
     *calls, or heavy  disk I/O, you should kick off a new thread,
     * or use {@link android.os.AsyncTask}.</p>
     *
     * @param intent The Intent supplied to {@link
         android.content.Context#startService},
     * as given.  This may be null if the service is being
     * restarted after its process has gone away, and it
     * had previously returned anything except {@link
      #START_STICKY_COMPATIBILITY}.
     * @param flags Additional data about this start request.
     * Currently either 0, {@link #START_FLAG_REDELIVERY},
     * or {@link #START_FLAG_RETRY}.
     * @param startId A unique integer representing this specific
     * request to  start.  Use with {@link #stopSelfResult(int)}.
     *
     * @return The return value indicates what semantics the system
     * should use for the service's current started state.
     * It may be one of the constants associated with the
     * {@link #START_CONTINUATION_MASK} bits.
     *
     * @see #stopSelfResult(int)
     */
    public int onStartCommand(Intent, int, int);

    /**
     * Called by the system to notify a Service that it is no longer
     * used and is being removed.  The service should clean up any
     * resources it holds (threads, registered receivers, etc)
```

```
     * at this point. Upon return, there will be no more calls
     * in to this Service object and it is effectively dead.
     * Do not call this method directly.
     */
    public void onDestroy();

    /**
      * Return the communication channel to the service.
      * May return null if clients can not bind to the service.
      * The returned {@link android.os.IBinder} is usually
      * for a complex interface that has been described using
      * aidl.
      *
      * <p><em>Note that unlike other application components, calls
      * on to the IBinder interface returned here may not happen on
      * the main thread of the process</em>. More information about
      * the main thread can be found in Processes and Threads.</p>
      *
      * @param intent The Intent that was used to bind to this
      * service, as given to {@link android.content.Context
      * #bindService Context.bindService}. Note that any extras
      * that were included with the Intent at that point will
      * <em>not</em> be seen here.
      *
      * @return Return an IBinder through which clients can call on
      * to the service.
      */
    public void onBind(Intent);

    /**
     * Called when all clients have disconnected from a particular
     * interface published by the service. The default
     * implementation does nothing and returns false.
     *
     * @param intent The Intent that was used to bind to this service
     * as given to {@link android.content.Context#bindService
     * Context.bindService}. Note that any extras that were included
     * with the Intent at that point will <em>not</em> be seen here.
     *
     * @return Return true if you would like to have the service's
     * {@link #onRebind} method later called when new clients bind
     * to it.
     */
    public boolean onUnbind(Intent);
```

```
public class BluetoothGattCallback {

  /**
   * Callback indicating when GATT client has connected/
   * disconnected to/from a remote GATT server.
   */
  public void onConnectionStateChange(BluetoothGatt gatt,
  *  int status, int newState) {
    ensures sending a broadcast containing the state information
  }

  /**
   * Callback invoked when the list of remote services,
   * characteristics and descriptors for the remote device
   * have been updated, ie new services have been discovered.
   *
   * @param gatt GATT client invoked {@link BluetoothGatt
   * #discoverServices}
   * @param status {@link BluetoothGatt#GATT_SUCCESS} if the
   * remote device has been explored successfully.
   */
  public void onServicesDiscovered(BluetoothGatt gatt,
  * int status) {
  }

  /**
   * Callback reporting the result of a characteristic read
   * operation.
   *
   * @param gatt GATT client invoked {@link BluetoothGatt
   * #readCharacteristic}
   * @param characteristic Characteristic that was read from the
   * associated remote device.
   * @param status {@link BluetoothGatt#GATT_SUCCESS} if the read
   * operation was completed successfully.
   */
  public void onCharacteristicRead(BluetoothGatt gatt,
  BluetoothGattCharacteristic characteristic,  int status) {
  }

  /**
   * Callback indicating the result of a characteristic write
```

```
     * operation.
     *
     * <p>If this callback is invoked while a reliable write
     * transaction is  in progress, the value of the characteristic
     * represents the value reported by the remote device.
     * An application should compare this value to the desired
     * value to be written. If the values don't match,
     * the application must abort the reliable write transaction.
     *
     * @param gatt GATT client invoked {@link BluetoothGatt
     * #writeCharacteristic}
     * @param characteristic Characteristic that was written to
     * the associated  remote device.
     * @param status The result of the write operation
     *               {@link BluetoothGatt#GATT_SUCCESS} if the
     * operation succeeds.
     */
    public void onCharacteristicWrite(BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic, int status) {
    }

    /**
     * Callback triggered as a result of a remote characteristic
     * notification.
     *
     * @param gatt GATT client the characteristic is associated
     * with
     * @param characteristic Characteristic that has been updated
     * as a result of a remote notification event.
     */
    public void onCharacteristicChanged(BluetoothGatt gatt,
    BluetoothGattCharacteristic characteristic) {
    }


    /**
     * Callback indicating the result of a descriptor write operatio
     *
     * @param gatt GATT client invoked {@link BluetoothGatt
     * #writeDescriptor}
     * @param descriptor Descriptor that was writte to the
     * associated remote device.
     * @param status The result of the write operation
     * {@link BluetoothGatt#GATT_SUCCESS} if the operation
```

```
     * succeeds.
     */
    public void onDescriptorWrite(BluetoothGatt gatt,
    BluetoothGattDescriptor descriptor, int status) {
    }

    /**
     * Callback reporting the RSSI for a remote device connection.
     *
     * This callback is triggered in response to the
     * readRemoteRssi function.
     *
     * @param gatt GATT client invoked {@link BluetoothGatt
     * #readRemoteRssi}
     * @param rssi The RSSI value for the remote device
     * @param status {@link BluetoothGatt#GATT_SUCCESS} if the
     * RSSI was read successfully
     */
    public void onReadRemoteRssi(BluetoothGatt gatt, int rssi,
    int status) {
    }

    /**
     * Broadcasts the converted Bluetooth characteristic that was
     * read from the sensor.
     *
     * @param characteristic Characteristic that has been received
     * from a sensor.
     */
    private void broadcastCharacteristic
    (BluetoothGattCharacteristic characteristic) {
    }
}

/**
 * connects through the gatt callback to a  TI CC2650 MCU using
 *  its MAC-address
 *
 * @param the MAC address of a TI CC2650 MCU
 */
public boolean connect(String address) {
  requires address != null
    ensures established connection to the device
}
```

```
public void disconnect() {
  ensures disconnection from a connected TI CC2650 MCU
}

/**
 * Basic sensor control: turn power and notifications on and off.
 *
 * @param s
 * Sensor enum: IR_TEMPERATURE || BAROMETER || LUXMETER ||
 * HUMIDITY
 * @param power         true = on, false=off
 * @param notification  true = enabled, false=off
 */
private void controlSensor(Sensor s, boolean power,
boolean notification) {
 }

/**
 * Wrapper for a blocking write: Either execute if the Queue is
 * empty and no write is currently going on or enqueue the
 * write task.
 *
 * @param o Either a BluetoothGATTCharacteristic (e.g. the
 * config characteristic to turn the sensor on and off) or a
 * BluetoothGattDescriptor (e.g. the Client Characteristic
 * Config descriptor, to en-/disable the notifications).
 */
private synchronized void write(Object o) {
}

/**
 * Blocking write; Sets isWriting to true (onWriteCharacteristic/
 * Descriptor settings it to false when finished)
 *
 * @param o Either a BluetoothGATTCharacteristic (e.g. the config
 * characteristic to turn the sensor on and off) or a
 * BluetoothGattDescriptor (e.g. the Client Characteristic
 * Config descriptor, to en-/disable the notifications).
 */
private synchronized void doWrite(Object o) {
}

/**
```

```
         * Function that is called if a write finishes or has illegal
         * arguments
         */
        private synchronized void nextWrite() {
        }

}

public class ScanListActivity extends AppCompatActivity {

    /**
     * Methods overridden from the interface import android.support.v7.a
     */

    /**
      * Called when the activity is starting.  This is where most
      * initialization should go: calling {@link #setContentView(int)}
      * to inflate the activity's UI, using {@link #findViewById}
      * to programmatically interact with widgets in the UI, calling
      * {@link #managedQuery(android.net.Uri , String[], String,
      * String[], String)} to retrieve  cursors for data being
      * displayed, etc.
      *
      * <p>You can call {@link #finish} from within this function, in
      * which case onDestroy() will be immediately called without any
      * of the rest of the activity lifecycle ({@link #onStart},
      * {@link #onResume}, {@link #onPause}, etc) executing.
      *
      * <p><em>Derived classes must call through to the super class's
      * implementation of this method.  If they do not, an exception
      * will be  thrown.</em></p>
      *
      * @param savedInstanceState If the activity is being
      * re-initialized after previously being shut down then
      * this Bundle contains the data it most recently supplied
      * in {@link #onSaveInstanceState}.
      * <b><i>Note: Otherwise it is null.</i></b>
      *
      */
    protected void onCreate(Bundle savedInstanceState) {
    }

    /**
     * Dispatch onResume() to fragments.  Note that for better
```

```
     * inter-operation with older versions of the platform, at
     * the point of this call the fragments attached to the activity
     * are <em>not</em> resumed.  This means that in some
     * cases the previous state may still be saved, not allowing
     * fragment transactions that modify the state.  To correctly
     * interact with fragments in their proper state, you should
     * instead override {@link #onResumeFragments()}.
     */
    protected void onResume() {
    }

    /**
      * Called as part of the activity lifecycle when an activity is
      * going into  the background, but has not (yet) been killed.
      * The counterpart to {@link #onResume}.
      *
      * <p>When activity B is launched in front of activity A, this
      * callback will be invoked on A.  B will not be created until
      * A's {@link #onPause} returns, so be sure to not do
      * anything lengthy here.
      *
      * <p>This callback is mostly used for saving any persistent
      * state the activity is editing, to present a "edit in place"
      * model to the user and making sure nothing is lost if there
      * are not enough resources to start the new activity without
      * first killing this one.  This is also a good place to do things
      * like stop animations and other things that consume a
      * noticeable amount of CPU in order to make the switch to
      * the next activity as fast as possible, or to close resources
      * that are exclusive access such as the camera.
      *
      * <p>In situations where the system needs more memory it may kill
      * paused processes to reclaim resources.  Because of this, you
      * should be sure that all of your state is saved by the time you
      * return from this function.  In general
      * {@link #onSaveInstanceState} is used to save per-instance
      * state in the activity and this method is used to store
      * global persistent data (in content providers, files, etc.)
      *
      * <p>After receiving this call you will usually receive a
      * following call  to {@link #onStop} (after the next activity
      * has been resumed and displayed), however in some cases
      * there will be a direct call back to {@link #onResume} without
      * going through the stopped state.
```

```
     *
     * <p><em>Derived classes must call through to the super class's
     * implementation of this method.  If they do not, an exception
     * will be thrown.</em></p>
     */
    protected void onPause() {
    }

    /**
     * Bluetooth LE scan callbacks. Scan results are reported using
     * these callbacks.
     *
     * @see BluetoothLeScanner#startScan
     */
    public abstract class ScanCallback {
        /**
         * Callback when a BLE advertisement has been found.
         *
         * @param callbackType Determines how this callback was
         * triggered. Could be one of
         *            {@link ScanSettings#CALLBACK_TYPE_ALL_MATCHES},
         *            {@link ScanSettings#CALLBACK_TYPE_FIRST_MATCH} or
         *            {@link ScanSettings#CALLBACK_TYPE_MATCH_LOST}
         * @param result A Bluetooth LE scan result.
         */
        public void onScanResult(int callbackType, ScanResult result) {
        }

        /**
         * Callback when batch results are delivered.
         *
         * @param results List of scan results that are previously
         * scanned.
         */
        public void onBatchScanResults(List<ScanResult> results) {
        }

        /**
         * Callback when scan could not be started.
         *
         * @param errorCode Error code (one of SCAN_FAILED_*) for scan
         */
        public void onScanFailed(int errorCode) {
        }
```

```
        }

        /**
         * @param enable if true the scan will start and stop after 5
         * seconds
         *               if false the scan will stop immediately
         */
        private void scanLeDevice(final boolean enable) {
        }

    }
```

The **StorageManager** provides a possibility to store the data from the sensor together with the current position. It also can store the data in a .json file and upload them to a webserver.

```
InterfaceStorageManager {
  public void setSensor(integer internalSensorID){
    activeSensor = internalSensorID;
  }
  //tells the storage manager to get data from the active sensor and
  //the tracing manager, mark them with a timestamp and store the data
  //in a DataSet.
  public void measureNow(){
  }
  //get the latest measured DataSet of the active sensor
  public DataSet getLiveData() {
    return DataSet;
  }
  //get a DataSet which contains data of the activeSensor and position
  //at the specified time. If no data was stored at that time, an empty
  //DataSet will be returned.
  public DataSet getDataFrom(int time) {
    return DataSet;
  }
  //get multiple DataSets which contain all measured Data from the
  //active Sensor between 'time' and now.
  public DataSet[] getDataSince(int time) {
    return DataSet[];
  }
  // Upload the data set to a webserver
  // a .json file from time till now.
  public void uploadData(int time) {
    //upload the data to a server so webvr can later use it.
  }
```

```
}
```

The **TrackingManager** gets the current location of the smartphone and provides it to the rest of the App.

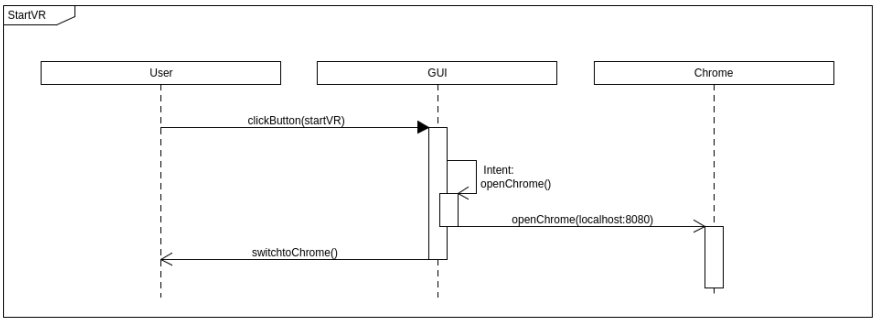
```
Interface TrackingManager {
  //get the current location of the device
  public Location p getCurrentPosition() {
    ensures iff lockCustomPos
      p == customPos;
      else
        p == currentPosition;
  }
}
```

The **VRBox** gets the data from a webserver and loads it into the VR-World.

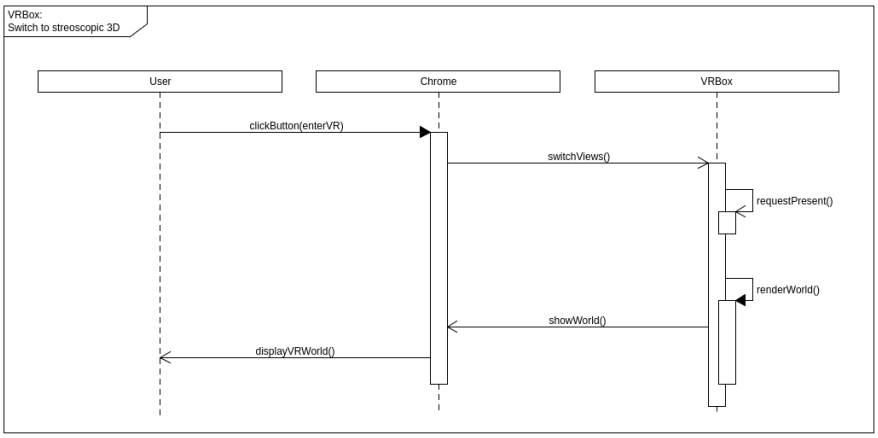
```
Interface WebVR {
  //Load the data from the webserver to display in VR.
  //The data will be a .json file which will be parsed
  // by in javascript by the web page.
  function loadData(file){
  }
}
```

2.4.3 Sequences

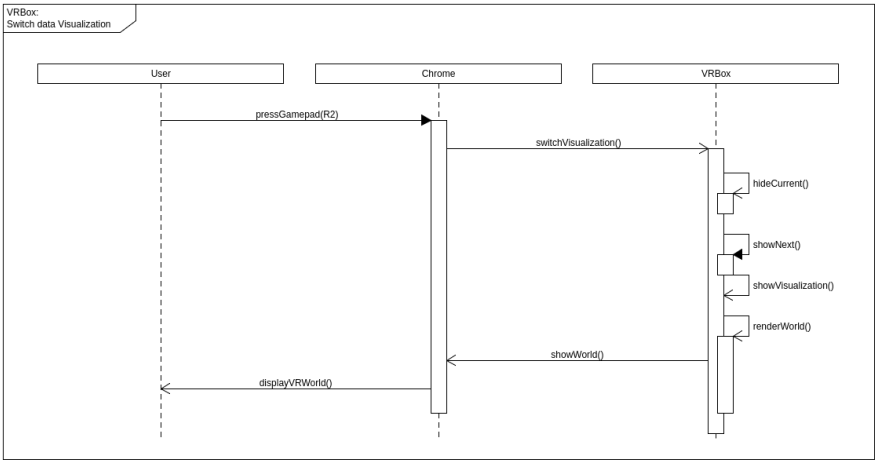
1) Start VR view: When the user clicks a button to start the VR view, a web browser is opened via an intent.



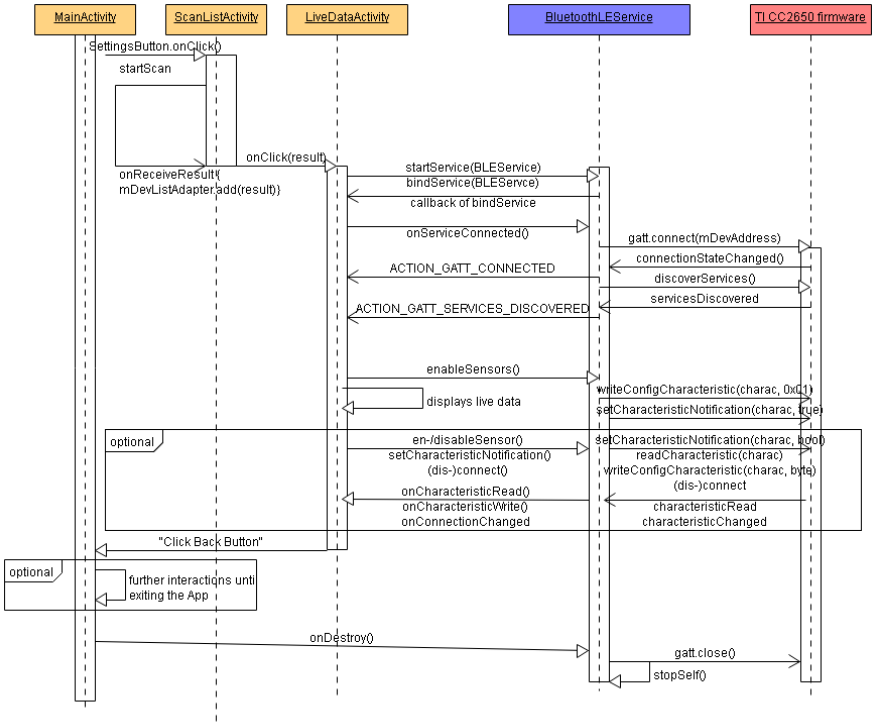
2) Switch to stereoscopic 3D: When the user enters the VR view, he can choose to switch the view and thereby switch to a stereoscopic view or switch back to non-stereoscopic view, respectively.



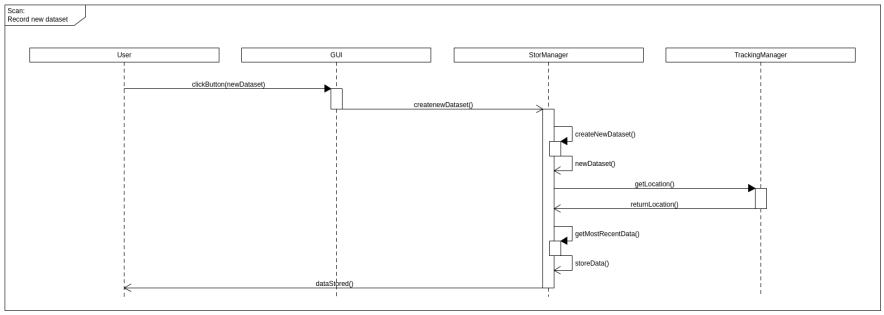
3) Switch data visualization: When the user opens the web site with the VR view, he can switch the visualization. The first view is a plane composed from the location tracking together with the data from the sensor, the second visualization is a representation of the locations as dots and the data as numbers hovering over these dots.



4) Scan and Connect: Scan for Bluetooth low energy devices and connect to one of them by clicking on the correct result ("TI CC2650"). After beeing connected one can edit the configuration of the sensors on the CC2650:
dis-/enable a sensor, (de-)activate the notifications for a measurement characteristic, read values, disconnect from the device. After that the user may start a new data scan record or start the VR View.



6) Record new data set: When the user clicks the button to record a new data set, the StorageManager will provide the data and get the repective location from the TrackingManager to calibrate the initial location for the recording session.



3 User interface

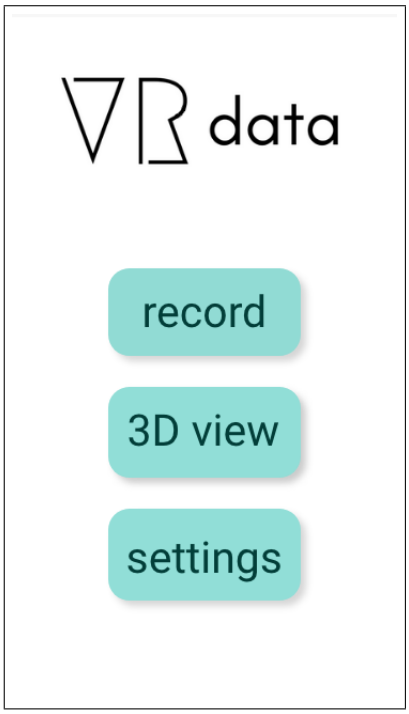
The following will explain the structure of the user interface.

3.1 Splash screen

When the app is launched and until it is fully loaded, a splash screen shows the app's logo.

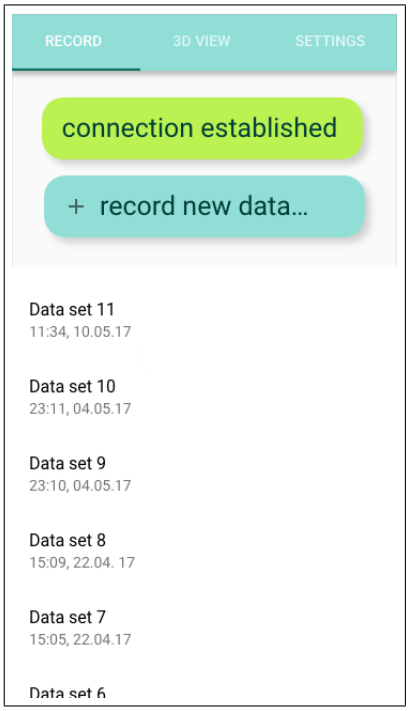
3.2 Start screen

When the app is fully loaded, the main menu is shown. It provides an overview of the three main functionalities of the app. This way, the user can choose directly what they want to do first: Either to record new data (and afterwards view them via the web application which can be invocated from here), to view the live data or to enter the settings screen where the tracking can be configured and the sensor device can be scanned for and connected to.



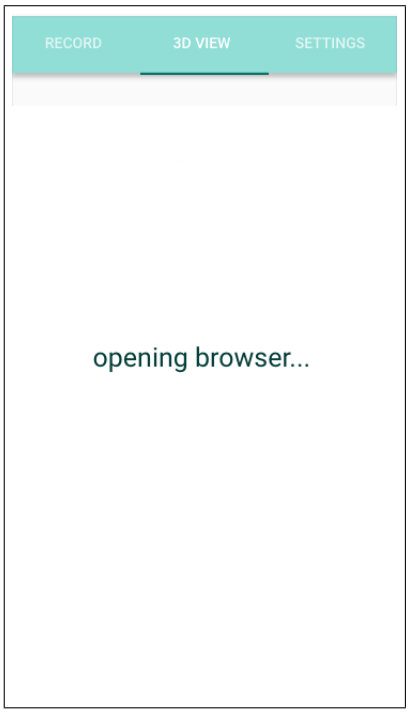
3.3 Record Data

Here, the user can start a new recording session and view the data that have been collected so far.



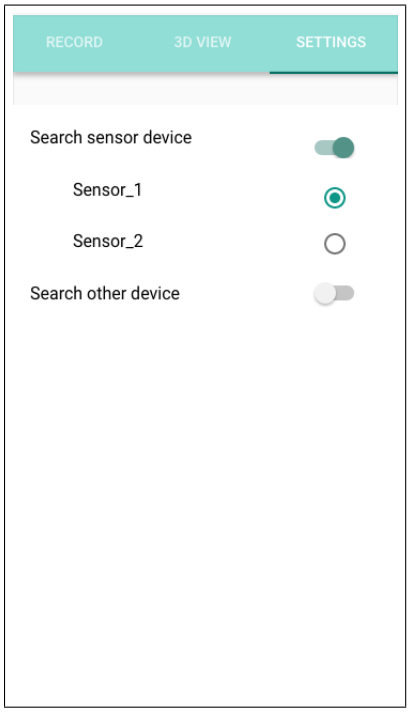
3.4 3D View

When the 3D view is chosen, the web application is started automatically.



3.5 Settings

Here, the tracking can be configured and the sensor device can be searched for and also be configured.



4 Test Cases

/T0300/ *Look around:* While in normal 3D mode the tester shall click the screen and drag first up to move the camera up. Then move down to move the camera down, then at last left and then right, all the time the camera must follow the movement of the finger. After this the tester shall tilt the phone up to move the camera up, then tilt it down, left and right. The camera shall follow the tilt direction of the phone all the time with no delay.

This test shall be repeated in stereoscopic 3D view. While the clicking and dragging shall not work, the tilting of the phone shall be the only way to pan the camera.

/T0310/ *Move inside the virtual reality scene:* While in normal 3D mode the tester shall tilt the joystick on the controller forward and the camera shall move forward. By tilting the joystick backward the camera shall move back, by tilting left the camera shall move left and by tilting right it shall move right. The camera shall always follow the view point, so forward is always in the center of the camera.

This test shall be again repeated in stereoscopic 3D view and all functions shall work the same.

/T0320/ *Searching, connecting and disconnecting devices:* The tester shall search a sensor device by pressing the "scan" button in the settings menu. All devices nearby shall be shown in a list with distinguishable entries. By tapping on a list entry a connection to the device shall be established. By tapping again on the list entry the connection shall be terminated.

/T0330/ *Displaying Data:* While in normal 3d mode the tester shall be able to see the data collected from the sensor. The tester shall be able to see the date relative to its stored position.
The test shall be again repeated in stereoscopic 3D view and shall work the same.

/T0340/ *Transferring Bluetooth data:* When the connection to a sensor device is established, the tester shall enter the data view wihin the app and see some representation of the transmitted data which allows him to check the connectivity and functionality to/ of the sensor device.

/T0350 *Storing Sensor data:* The tester shall be able to connect to a sensor and while connected the data received from the sensor shall be stored together with its current position, on the local file system. The tester shall open this file and check if the new data is in it.

/T0360 *Transferring sensor data to the Web Application:* The tester shall check if the data is transfered to the web application and shall check if the data is corresponds to the collected sensor data.