Projet: Pedestrian Dead Reckoning for Android

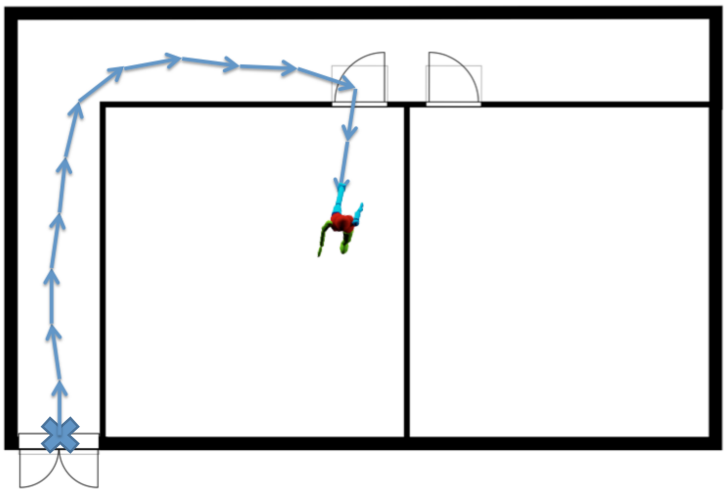
Navigation à l'estime pour piéton pour Android

Thibaud Michel
University of Grenoble Alpes

10 mars 2016

1 Introduction

Un des problèmes principaux de la localisation sur mobile aujourd'hui est le suivi de la personne là où le signal GPS ne fonctionne pas. C'est notamment le cas à l'intérieur des bâtiments ou des zones couvertes (tunnels, grottes...). Nous aimerions par exemple connaître la position d'une personne à l'intérieur d'un musée pour lui proposer du contenu spécifique, ou encore proposer à un mal voyant un chemin en fonction de l'endroit où il se trouve et de là où il va. Pour répondre à ce problème, nous avons décidé de ne pas ajouter de nouvelles infrastructures dédiées à la localisation comme le WiFi, les iBeacons, des émetteur/récepteur radio. Ces systèmes là sont utilisés quand la précision est la priorité de l'application. Ce projet s'orientera vers une autre approche qui se nomme Pedestrian Dead Reckoning (ou navigation à l'estime pour piéton en français). C'est une méthode de navigation qui consiste à déduire la position de la personne en fonction de la distance parcourue depuis sa dernière position connue. Vous aurez besoin de ces fichiers qui vous serviront de base pour votre projet. Ce projet est à faire par groupe de 2 personnes, éventuellement 3 mais les attentes seront plus fortes.



2 GPX Viewer (3h30)

Pour commencer, nous réaliserons un visualiseur de document GPX afin d'afficher nos futures traces réalisées avec le PDR. Le GPX est un format de fichier permettant l'échange de coordonnées géolocalisées sous forme de point de cheminement (waypoint), trace (track) ou itinéraire (route). Le document GPX a son propre schéma XML, mais dans ce projet nous ne nous intéresserons qu'à la partie "Track" du format. Il s'agit d'un arbre avec 4 niveaux hierarchiques : GPX > Track > TrackSeg > TrackPoint. Un "parser" vous est déjà fourni pour charger un fichier GPX.

2.1 GPX Structure (30 min)

Question 2.1 *Dans un premier temps nous allons tester le fonctionnement du parser avec le fichier simple.gpx en entrée. Ce fichier se trouve dans le répertoire "assets". Placez vous dans la méthode onCreate et ouvrez votre fichier gpx grâce à la ligne suivante :*

```
InputStream is = getAssets().open("simple.gpx");
```

Appelez votre méthode parse() de la classe GPX puis vérifiez le contenu de la structure en utilisant la fonction de log : Log.v().

2.2 Trace Viewer (3h)

Le meilleur moyen de visualiser une trace est de l'afficher au dessus d'un fond de carte. Sur Android, Google nous propose un accès facile à son API Maps, je vous conseille de l'utiliser. La première installation est un peu fastidieuse mais l'utilisation de la librairie reste assez facile. Voici le lien vers la documentation pour la mise en place. Libre à vous d'utiliser un autre système de cartographie si vous préferez (MapsForge, MapBox)

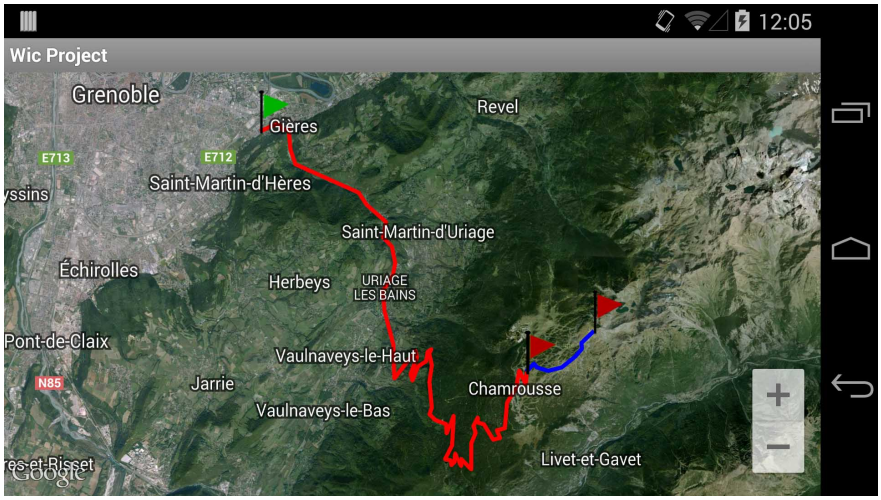


FIGURE 1 – Exemple d'un Viewer GPX

Question 2.2 *Commencez par créer une nouvelle activity qui aura comme rendu seulement une vue avec la carte Google*

Question 2.3 *Centrez cette carte sur Grenoble avec un niveau de zoom suffisant en utilisant le système de Camera. Pour retrouver les coordonnées latitude et longitude ainsi que le niveau de zoom d'une carte vous pouvez vous aider du site maps.google.com et regarder l'url.*

Question 2.4 *Créez une suite de segments en surcouche de la carte GoogleMap grâce à la classe Polyline. Pour vos tests utilisez les points suivant : [1](lat : 45.189 lon :5.704), [2](45.188, 5.725), [3](45.191, 5.733). Vous verrez alors une ligne traverser Grenoble. Vous pouvez vous amuser en changeant sa couleur et sa largeur et en créant une deuxième Polyline.*

Question 2.5 *Créez une méthode qui prend en entrée votre structure GPX et qui affiche les traces en surcouche du fond de carte Google grâce aux polylines. Nous créerons une nouvelle polyline pour chaque TrackSeg.*

Question 2.6 *(Optionnel) Centrez la carte de manière à afficher toute les traces du document GPX à l'écran. Pour cela vous pourrez utiliser CameraUpdateFactory.newLatLngBounds(). Pour vous aider, créez une méthode getLatLngBounds() dans la classe GPX. Regardez du côté de cette méthode : LatLngBounds.Builder, 2-3 lignes suffiront.*

Question 2.7 *En utilisant les intent-filter, votre visualiseur sera capable de prendre en entrée n'importe quel fichiers de type GPX contenus dans votre téléphone de façon implicite. Vous pouvez pour cela utiliser une application tierce pour lister les documents de votre téléphone comme ES Explorer*

3 Pedestrian Dead Reckoning (9h)

Dans cette partie, nous allons utiliser les différents capteurs du téléphone pour estimer la position de la personne. Le PDR (ou navigation à l'estime) est un système basé sur le positionnement relatif, c'est à dire que la précision de la nouvelle position dépend aussi de la précision de l'ancienne, contrairement au GPS qui fournit des positions (latitude et longitude) qui sont absolues. La navigation à l'estime est simplement basée sur la formule suivante, la seule difficulté est de trouver une valeur pour les différents paramètres.

```
Nouvelle position = calcul_nouvelle_position(Position courante, Taille du pas, Angle du pas)
```

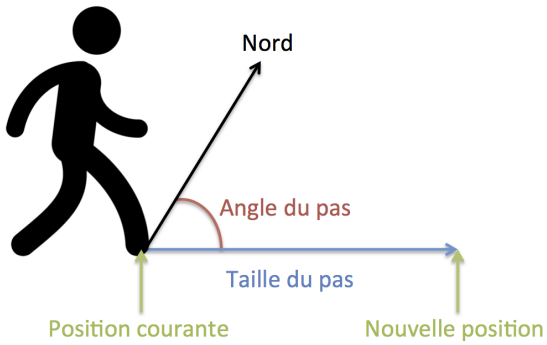


FIGURE 2 – Principe de la navigation à l'éstime

Question 3.1 *Créez un package* ***pdr*** *qui servira pour cette partie*

Question 3.2 *Créez une nouvelle Activity avec une carte Google Map qui servira (plus tard) à afficher le chemin parcouru.*

3.1 Step Detection (4h)

Nous commencerons par faire un système de détection de pas que l'on appelle plus communément un podomètre. Le moyen le plus simple pour réaliser un tel système est d'utiliser l'accélération linéaire du téléphone. Un premier problème se pose, selon l'endroit où l'on place le téléphone (main, poche, ...) les données récupérées par les capteurs sont différentes. Il faudrait analyser le modèle de marche d'une personne pour bien comprendre le problème. Pour ceux qui seraient intéressés je vous conseille d'aller lire la Thèse de Q. Ladetto. Pour le projet nous en retiendrons surtout le modèle de marche quand le téléphone est tenu dans la main.

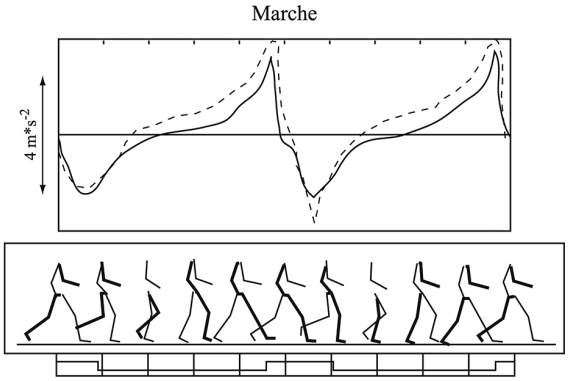


FIGURE 3 – Accélération linéaire en Z du centre de masse (courbe pleine) et la même accélération perçue par le smartphone (courbe discontinue) tenu dans la main

Afin de mieux comprendre les capteurs sous Android, je vous recommande de télécharger l'application Sensor Kinectics et de regarder ce qu'il se passe au niveau de l'accélération linéaire quand vous marchez. Regardez principalement l'axe Y sur lequel nous allons travailler.

Question 3.3 *Créez une classe* ***StepDetectionHandler*** *dont le constructeur prend en paramètre un SensorManager et 2 méthodes start() et stop() qui s'abonneront/désabonneront au "capteur" d'accélération linéaire. Aidez-vous du document annexe sur les capteurs.*

Pour la suite vous pourrez vous aider de Log.d() pour afficher les valeurs des capteurs dans logcat.

Question 3.4 *Dans la méthode SensorListener.onSensorChanged(), extraire l'accélération linéaire en Y.*

Choisissez un seuil au dessus duquel on sera quasiment certain qu'une personne est entrain de faire un pas, vous pourrez vous aider de l'application Kinectics précédement téléchargée.

Question 3.5 *Toujours dans onSensorChanged(), seulement avec les valeurs de z, trouvez une façon de détecter le moment où l'accélération linéaire passe au dessus de ce seuil, on considérera qu'à ce moment là, la personne a fait un pas. Vous pouvez tester votre application en simulant un pas à la main.*

Question 3.6 *Créez un listener StepDetectionListener avec la méthode : void newStep(float stepSize). On considerera que la taille du pas est de 0.8m. Dans votre Activity, implémentez le système d'abonnement associé au listener.*

Question 3.7 *(Optionnel) La valeur de l'accélération linéaire en Z peut-être affinée en établissant une moyenne sur les 5 dernières valeurs par exemple.*

3.2 Device Attitude (1h)

Nous allons maintenant essayer de déterminer l'angle du pas effectué. Cet angle va être calculé par rapport au Nord magnétique de la terre, c'est à dire que si le téléphone est orienté en direction du Nord magnétique, la valeur retourné est 0°, à l'Est 90°... Idéalement il faudrait utiliser le Magnétomètre du téléphone, cependant il n'est pas fiable à l'intérieur des bâtiments à cause des variations magnétiques produites par les différentes machines. On pourrait aussi utiliser le Gyroscope qui permet à l'aide d'une intégration de connaître la rotation du téléphone sur un temps donné, malheureusement cette valeur est relative à la position de départ du téléphone. De plus, elle n'est pas fiable dans le temps (drift). La meilleure solution est de fusionner les capteurs, plusieurs algorithmes existent déjà et sont assez complexe. Afin de ne pas les recréer, Android nous propose d'utiliser des capteurs composites.

Question 3.8 *Créez une classe* ***DeviceAttitudeHandler*** *de la même manière dont vous avez utilisé l'accéléromètre dans la partie précédente (toujours avec les méthodes start() et stop()). Utilisez cette fois-ci le capteur Rotation Vector.*

Le problème est que les valeurs retournées par event.values dans SensorListener.onSensorChanged() ne sont pas exploitables directement pour ce que nous voulons faire. Il va falloir qu'on change de repère et qu'on récupère le vecteur de rotation comme sur la figure ci-dessous.

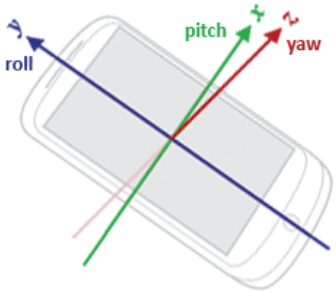


FIGURE 4 – Orientation du Smartphone sur les axes yaw, pith et roll

Voici une petite explication sur ces 3 valeurs, les unités sont en radians :

— **Yaw** $[-\pi; -\pi[$ correspond à ce qu'on appelle l'azimuth ou le bearing. C'est la détermination de l'angle que fait, dans le plan horizontal, la ligne du téléphone vers le nord géographique. 0 radian correspond au Nord, $\frac{\pi}{2}$ radian à l'Est... C'est cette valeur que l'on utilisera pour le PDR.
— **Pitch** $[-\frac{\pi}{2}; -\frac{\pi}{2}[$ correspond à la rotation du téléphone autour de l'axe x. On peut par exemple l'utiliser en réalité augmentée pour afficher des objets selon leur hauteur ou altitude.
— **Roll** $[-\pi; -\pi[$ correspond à la rotation du téléphone autour de l'axe y.

Question 3.9 *Utilisez les données du capteur (de type "Software") Rotation Vector. Ce capteur ne vous renvoie pas directement ces 3 valeurs, ce serait trop simple... Utilisez le bout de code ci-dessous pour récupérer les valeurs :*

```
@Override
public void onSensorChanged(SensorEvent event)
{
    // It is good practice to check that we received the proper sensor event
    if (event.sensor.getType() == Sensor.TYPE_ROTATION_VECTOR)
    {
        float[] rotMat = new float[9];
        float[] values = new flaot[3];

        SensorManager.getRotationMatrixFromVector(rotMat, event.values);
        SensorManager.getOrientation(rotMat, values);

        float yaw = orientationVals[0];
        float pitch = orientationVals[1];
        float roll = orientationVals[2];

    }
}
```

Question 3.10 *Vérifier les valeurs de yaw (nord = 0°, est = 90°...). N'oubliez pas les conversions degrés radians.*

Question 3.11 *Créez un getter pour récupérer la valeur de yaw à l'exterieur de la classe DeviceAttitudeHandler.*

3.3 Step Positioning (1h)

Maintenant que nous avons la taille et l'angle du pas il ne nous reste plus qu'à calculer la nouvelle position en fonction de la position courante. C'est à dire **calcul_nouvelle_position** dans la formule suivante :

```
Nouvelle position = calcul_nouvelle_position(Position courante, Taille du pas, Angle du pas)
```

Si nous nous plaçons dans un repère orthonormé il est facile de calculer la nouvelle position.

```
Nouvelle position x = Position courante x + Taille du pas * cos(Angle du pas)
Nouvelle position y = Position courante y + Taille du pas * sin(Angle du pas)
```

Cependant, ce n'est pas nouveau, la terre n'est pas plate et le système de latitude/longitude que l'on utilise peut être assimilé à une sphère ou encore une ellipsoïde.

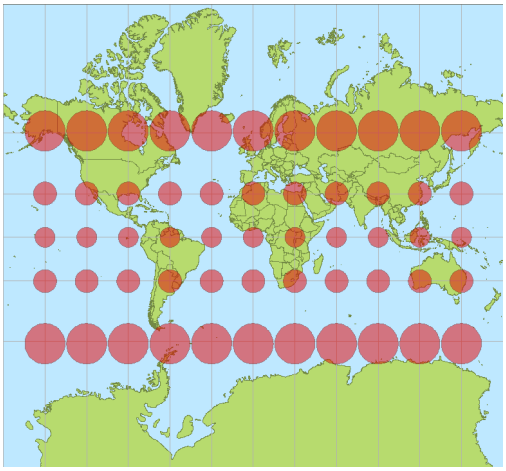


FIGURE 5 – Distortion de la projection de Mercator. Tous les cercles rouges représentent la même aire à la surface de la terre.

Question 3.12 *Créez une classe* ***StepPositioningHandler*** *avec un attribut* ***mCurrentLocation*** *ainsi que les getter et setter associés.*

Question 3.13 *Créez une méthode* ***computeNextStep(float stepSize, float bearing)*** *qui prend en entrée la taille et l'angle du pas puis qui retourne la nouvelle position. Ce document vous aidera à calculer le point de destination en utilisant un arc du cercle. Attention aux unités et conversions rad/deg. Dans l'article la latitude et longitude sont en radians et dans votre code, en degrés. Attention aux conversions !*

3.4 Google Map Tracer (1h)

Afin de représenter la trace de la personne à l'écran en temps réel, nous allons ré-utiliser GoogleMap comme à la partie précédente. Cette fois-ci nous n'allons pas utiliser le format GPX mais simplement créer une polyline qu'on mettra à jour à chaque nouvelle position.

Question 3.14 *Créez une classe* ***GoogleMapTracer*** *dans le package* ***viewer*** *dont le constructeur prend en praramètre GoogleMap.*

Question 3.15 *Créez une méthode* ***newSegment()*** *qui commencera une nouvelle polyline et une méthode* ***newPoint(LatLng point)*** *qui affichera un point supplémentaire à la polyline courante.*

Question 3.16 *(Optionnel) Ajoutez un marker au début et à la fin d'une polyline.*

3.5 PDR Application (2h)

Vous avez tout en main pour faire l'application qui visualise le déplacement d'un piéton. Il ne reste plus qu'à réutiliser tout ce qui a été fait précédemment.

Question 3.17 *Commencez par créer une nouvelle Activity. On y affichera la carte GoogleMap en plein écran comme on l'a fait pour le GPX Viewer.*

Depuis le début il manque une donnée importante que l'on n'a pas définie. Nous n'avons pas de point de départ et c'est normal, le système PDR est relatif. Afin de rendre l'application flexible et éviter d'entrer manuellement une valeur latitude/longitude, nous allons réutiliser la carte précédement créée pour choisir un point de départ.

Question 3.18 *Abonnez vous au listener de GoogleMap.setOnMapClickListener(). A chaque nouveau clic sur la carte vous commencerez un nouveau segment et mettrez à jour la position courante dans* ***StepPositioningHandler****.*

Question 3.19 *Abonnez vous au listener de Step Detection Handler pour savoir à quel moment l'utilisateur fait un pas. Faites le nécessaire en utilisant Device Attitude Handler et Step Positioning Handler pour récupérer la nouvelle position et l'afficher avec le Google Map Tracer*

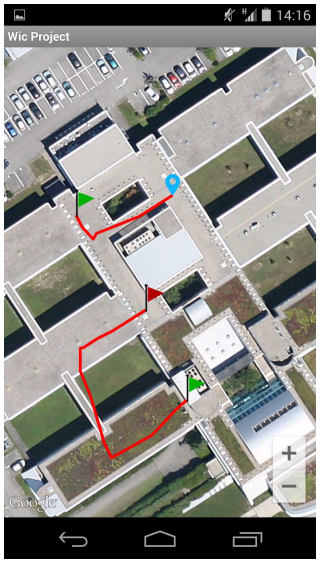


FIGURE 6 – Un exemple de l'Application PDR.

4 Extras

Cette partie est facultative mais elle permettra de rendre l'application un peu plus complète. Libre à vous d'implémenter ce que vous voulez. Voici une liste de propositions qui ne doivent pas forcément être traitées dans l'ordre.

4.1 Exporter la trace en GPX (1h30)

Vous avez un Visualiseur de GPX et un PDR qui affiche sa propre trace. Il serait intéressant de pouvoir sauvegarder les traces du PDR dans un fichier GPX afin de le revoir plus tard. Pour cela on peut utiliser XMLSerializer ou DocumentBuilder qui permettent de générer du XML. Voici un exemple d'utilisation de chacun des sérializeurs.

Question 4.1 *Créez une méthode qui prend en entrée une structure GPX et un File puis qui stocke les informations dans un fichier.*

4.2 Suivre la position de l'utilisateur (15 min)

Question 4.2 *Faites en sorte que la caméra de la GoogleMap suive la position de l'utilisateur à chaque nouveau pas.*

4.3 Centrer la carte sur la dernière position de l'utilisateur (15 min)

Question 4.3 *A l'aide du LocationManager, récupérez (si elle existe) la dernière position connue de l'utilisateur. Utilisez là pour centrer la carte à l'ouverture de l'Activity*

4.4 Icône de l'utilisateur (15min + 45min)

Pour le moment quand vous avancez dans l'application PDR, vous voyez seulement une trace s'agrandir à l'endroit où se trouve l'utilisateur.

Question 4.4 *Créez un marker dans Google Map Tracer qu'on déplacera à la position courante de l'utilisateur.*

Grâce au Device Attitude Handler nous avons accès à l'azimuth en temps réel du téléphone. On va s'en servir pour afficher un marker (flèche) sur lequel on effectuera une rotation.

Question 4.5 *Créez un listener dans la classe Device Attitude Handler pour que le Google Map Tracer puisse recevoir les notifications.*

Question 4.6 *Modifiez le marker précédent par une flèche et lui appliquer une rotation quand le téléphone aura changé de direction.*

Pour information, cette dernière partie peut être faite avec l'interface GoogleMap.OnMyLocationChangeListener, mais son implémentation est un peu complexe pour le temps imparti.

5 Annexes

5.1 Les Listeners

En programmation évenementielle, il existe plusieurs Design Pattern différents pour recevoir des informations provenants de d'autres entités, les listeners en font parti. Sur Android vous en avez peut être déjà vu avec les boutons d'une interface graphique, lors d'un clic de l'utilisateur un évènement est renvoyé. L'idée est de s'abonner à une entité puis de recevoir une notification à un moment qui n'était pas prévisible. Nous pouvons nous aussi utiliser ce même principe pour surveiller les capteurs (par exemple). Voici ci-dessous un exemple avec un capteur de température.

```
public class TemperatureHandler implements SensorEventListener {

        @Override
        public final void onSensorChanged(SensorEvent event) {

                float temperature = event.values[0];

                if(temperature > 30 && mTemperatureListener != null) {
                        mTemperatureListener.onHightTemperatureDetected(event.values[0]);
                }
        }


        private TemperatureListener mTemperatureListener;

        public void setTemperatureListener(TemperatureListener listener) {
                mTemperatureListener = listener;
        }

        public interface TemperatureListener {
                public void onHightTemperatureDetected(float temperature);
        }
}
```

Dès que le capteur de température dépasse 30°, l'objet abonné reçoit alors une notification grâce à la méthode **onHightTemperatureDetected()**

```
public class TemperatureActivity extends Activity {

        @Override
        public final void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);

                TemperatureHandler mTemperatureHandler = new TemperatureHandler();
                mTemperatureHandler.setListener(mTemperatureListener);
        }

        private TemperatureListener mTemperatureListener = new TemperatureListener() {

                public void onHightTemperatureDetected(float temperature) {
                        Toast.makeText(getApplicationContext(),
                                "High temperature detected: " + temperature,
                                Toast.LENGTH_SHORT).show();
                }
        };
}
```

5.2 Les Capteurs

La plupart des appareils Android ont des capteurs embarqués qui mesurent les mouvements, l'orientation et l'environnement. Ces capteurs fournissent des données brutes avec une bonne précision. Par exemple, dans un jeu on va utiliser le capteur de gravité et l'accéléromètre pour reconnaître des mouvements complexes de l'utilisateur : inclinaison, secousse, rotation ou encore dans une application de voyage l'application utilisera le capteur magnétique et l'accéléromètre pour les utiliser comme boussole. La plateforme Android supporte 3 catégories de capteurs, ils peuvent êtres bruts ou composés (fusion) :

- **Capteurs de Mouvement** : Accéléromètre, Gravité, Gyroscope...
- **Capteurs d'Environnement** : Température extérieure, Pression, Luminosité...
- **Capteurs de Position** : Magnétique, Proximité, Orientation...

On peut accéder à tous ces capteurs grâce au Sensor Framework d'Android. On s'abonne au capteur que l'on veut utiliser grâce à un système de Listener (SensorEventListener). Les données reçues sont au format SensorEvent, on a accès à la précision de la donnée, le capteur duquel elle provient, le timestamp de l'évènement et un vecteur contenant les valeurs en fonction du capteur.

Les capteurs sont souvent très pratiques dans les applications mais il ne faut pas oublier qu'ils sont très gourmands en énergie ! Il est alors important de les utiliser dans les bonnes situations et surtout de s'y abonner seulement quand on en a besoin. Voici un exemple :

```
1  public class SensorActivity extends Activity implements SensorEventListener {
2      private SensorManager mSensorManager;
3      private Sensor mLight;
4
5      @Override
6      public final void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          setContentView(R.layout.main);
9
10         mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
11         mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
12     }
13
14     @Override
15     public final void onAccuracyChanged(Sensor sensor, int accuracy) {
16         // Do something here if sensor accuracy changes.
17     }
18
19     @Override
20     public final void onSensorChanged(SensorEvent event) {
21         // The light sensor returns a single value.
22         // Many sensors return 3 values, one for each axis.
23         float lux = event.values[0];
24         // Do something with this sensor value.
25     }
26
27     @Override
28     protected void onResume() {
29         super.onResume();
30         mSensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
31     }
32
33     @Override
34     protected void onPause() {
35         super.onPause();
36         mSensorManager.unregisterListener(this);
37     }
38 }
```