BeSports

Localisation for sportsmen

Indoor localisation – Rapport de projet

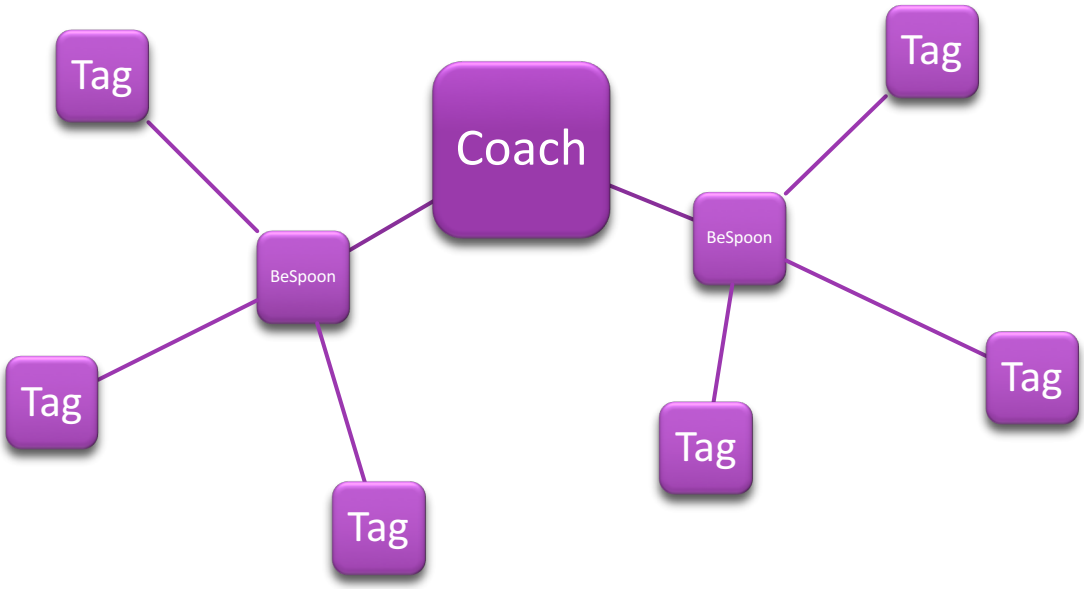
Nicolas Duponchel - Raphaël Courivaud - Alexandre Aubry

Claudia Cao - ChanBora Sim

Table des matières

Introduction

Le projet réalisé a pour but principal de localiser en temps réel des sportifs sur un terrain de sport. Nous créons alors une solution visant à assister le coach sportif dans ses entrainements. Les outils utilisés dans ce projet se résument à la technologie *Android* pour la partie application mobile et calculs algorithmiques, puis une nouvelle technologie mise en place par des anciens ingénieurs de l'ESIEE Paris nous servira pour la localisation. Celle-ci s'appelle l'*Ultra Wild Bande*, et consiste à communiquer entre deux appareils (smartphone Android et capteurs appelés *Tag BeSpoon* dans notre cas) via des fréquences très basses.



Le principe de fonctionnement est le suivant :

- Les Tags BeSpoon sont positionnés autour du terrain de sport. Chaque joueur possède un téléphone BeSpoon sur lequel est installée une application que nous développons.
- Chaque joueur configure son application avec ses données et renseigne les positions des Tags. Chacun de ces téléphones sera capable de se situer dans l'espace grâce à la connexion avec les Tags.
- Ces positions sont transmises à l'application Coach via le système DDS. Pour finir cette application sera capable d'afficher les positions de tous les joueurs sur le terrain en temps réel, d'afficher leurs vitesses, hauteur de saut etc.

Nous avons alors séparé le travail en 4 parties :

- Algorithmes de localisation et vitesse - (Claudia et Chamborra)
- Connexion DDS - (Alexandre)
- Application Joueur - (Raphaël)
- Application Coach - (Nicolas)

Algorithmes de localisation

Trilatération

L'algorithme de trilatération permet de localiser un objet dans un plan en 2D ou 3D. Dans le cadre de notre projet de système de localisation indoor, nous avons eu besoin de faire la localisation en 3D d'un joueur sur un terrain (futsal, handball, basket-ball, …).
Nous avons implémenté l'algorithme de trilatération en java pour ensuite l'intégrer à nos différentes applications (Entraîneur et Joueur).
L'algorithme nécessitant l'utilisation de matrices, nous avons été contraints de trouver une solution qui s'est avérée être la librairie JAMA (Java Matrix). Cette bibliothèque inclut une classe Matrix avec tout un ensemble de fonctions telle que la transposée (transpose()) ou encore la multiplication de matrices de tailles différentes (times()). Dans le code, nous avons uniquement inclus les fichiers .java de JAMA dont nous avions besoin.
Le code d'algorithmie de la trilatération se trouve dans la classe Point, ou plus exactement ici :

```
        //              CREATION DE LA MATRICE A

        for(i=0; i<A.length; i++) {
            for(j=0; j<A[i].length; j++) {
                if (j == 0){
                    A[i][j] = 2*(tabPoint.get(i+1).x-tabPoint.get(0).x);
                }
                else if (j == 1){
                    A[i][j] = 2*(tabPoint.get(i+1).y-tabPoint.get(0).y);
                }
                else{
                    A[i][j] = 2*(tabPoint.get(i+1).z-tabPoint.get(0).z);
                }
            }
        }

        Matrix AMatrix = new Matrix (A);
        //System.out.println("A :");
        //AMatrix.print(3,2);

//              CREATION DE LA MATRICE b

for(i=0; i<b.length; i++) {
    Point zero = tabPoint.get(0);
    Point actif = tabPoint.get(i+1);
    b[i][0] = (Math.pow(tabDist.get(0),2.0)-Math.pow(tabDist.get(i+1),2.0)+
    Math.pow(actif.x,2.0)+Math.pow(actif.y,2.0)+Math.pow(actif.z,2.0)-
    Math.pow(zero.x,2.0)-Math.pow(zero.y,2.0)-Math.pow(zero.z,2.0));
}
```

```
//            CALCUL ALGO DE TRILATERATION

Matrix bMatrix = new Matrix(b);
//System.out.println("b :");
//bMatrix.print(1,2);

/* Formule : (AMT*AM)^-1*AMT*b */
Matrix AMatrixTranspose = AMatrix.transpose();
//System.out.println("AMT :");
//AMatrixTranspose.print(3,2);

Matrix AMTxAM = AMatrixTranspose.times(AMatrix);
//System.out.println("AMTxA :");
//AMTxAM.print(3,2);

//Log.d(TAG, AMTxAM.toString());
Matrix inverseAMTxAM = AMTxAM.inverse();
//System.out.println("(AMTxA)^-1");
//inverseAMTxAM.print(2, 6);

Matrix mult1 = inverseAMTxAM.times(AMatrixTranspose);
//System.out.println("(AMT*AM)^-1*AMT :");
//mult1.print(3,2);

Matrix result = mult1.times(bMatrix);
//System.out.println("(AMT*AM)^-1*AMT*b :");
//result.print(3,2);

int resultCol = result.getColumnDimension();
int resultRow = result.getRowDimension();

    //             COORDONNEES DU POINT

    this.x = result.get(resultRow-3,resultCol-1);
    this.y = result.get(resultRow-2,resultCol-1);
    this.z = result.get(resultRow-1,resultCol-1);
```

Les prérequis pour lancer le calcul sont les suivants : il faut avoir initialisé les points correspondants à l'emplacement des balises en donnant leurs coordonnées X, Y, Z par rapport au terrain. Il faut également créer une ArrayList contenant les points précédents et une autre ArrayList contenant les distances de chaque balise au joueur dans le même ordre.

ArrayList1 contient A, B, C, D

ArrayList2 contient DistAJoueur, DistBJoueur, DistCJoueur, DistDJoueur

Étant donné que la trilatération demande au minimum 4 balises (dont une sur un plan différent des 3 autres) pour pouvoir calculer la position du joueur en 3D et que nous étions en possession de 8 balises, les ArrayList ont été les outils les plus adaptés pour pouvoir inclure 4 balises ou plus dans le code.

Nous créons ensuite les deux matrices A et b en fonction du nombre de Points contenu dans l'ArrayList1. Nous lançons alors l'algorithme de trilatération.

Lorsque celui ci se termine, nous mettons les éléments de la matrice résultante dans les attributs correspondants du Point créé.

Vitesse

Nous savons que la vitesse se calcule à partir de deux points pris à des instants différents.
La classe Vitesse se compose de 3 attributs : vitesse, lastTime et lastPoint.
L'attribut vitesse contient la vitesse en m/s du joueur, les attributs lastTime et lastPoint correspondent respectivement au dernier TimeStamp et au dernier Point relevé lors du dernier calcul de vitesse.
La fonction updateVitesse(Point p) a été pensé de sorte que l'attribut vitesse soit mis à jour à chaque mis à jour des valeurs du joueur dans l'application. Ainsi, une seule donnée de type Vitesse est créée au tout début.

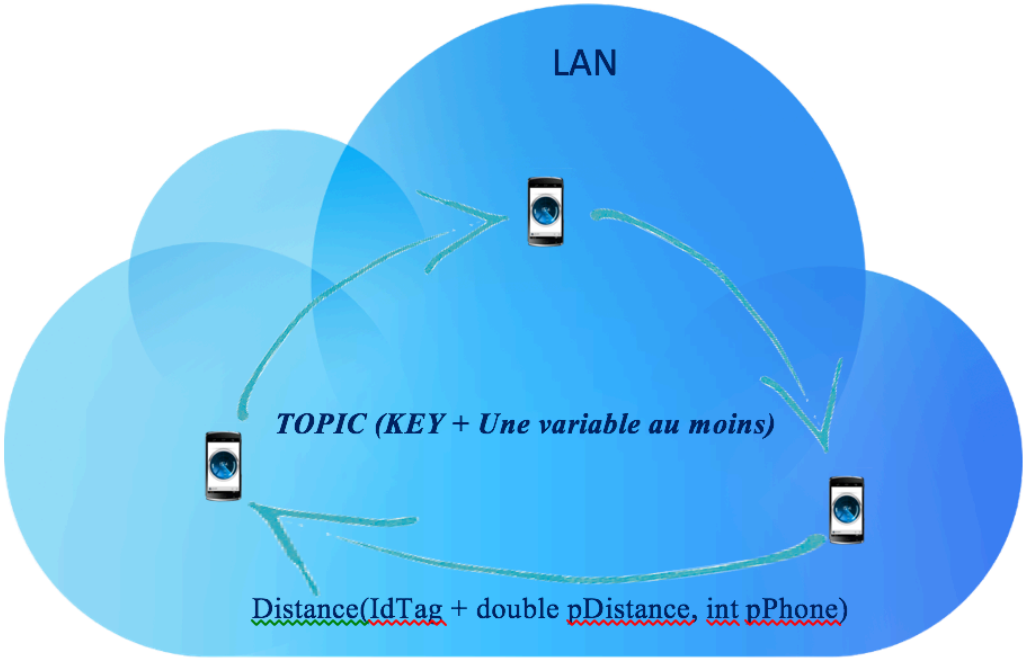
```
public void updateVitesse (Point p){

    if(this.lastTime == null || this.lastPoint==null){
        this.lastTime = new Date();
        this.lastPoint = p;
    }
    else{
        //On prend le point précédent qu'on garde dans p1 pour le calcul et on met le nouveau point dans l'attribut pour le calcul de vitesse suivant
        Point p1 = new Point(this.lastPoint.getX(), this.lastPoint.getY(),this.lastPoint.getZ());
        this.lastPoint = p;

        //Calcul du temps en milliseconde entre le point d'avant et le nouveau point
        long t0 = this.lastTime.getTime();
        this.lastTime = new Date();
        long t1 = this.lastTime.getTime();
        long tdiff = t1 - t0;

        //Calcul de la vitesse
        this.vitesse = (1000/tdiff) * Math.sqrt(Math.pow(this.lastPoint.getX() - p1.getX(),2.0)+ Math.pow(this.lastPoint.getY()-p1.getY(),2.0));

    }
}
```

Étant donné que le temps entre deux mesures est de moins d'une seconde, il nous a fallu effectuer une petite conversion des TimeStamps en millisecondes puis une reconversion de la vitesse obtenue en m/s.

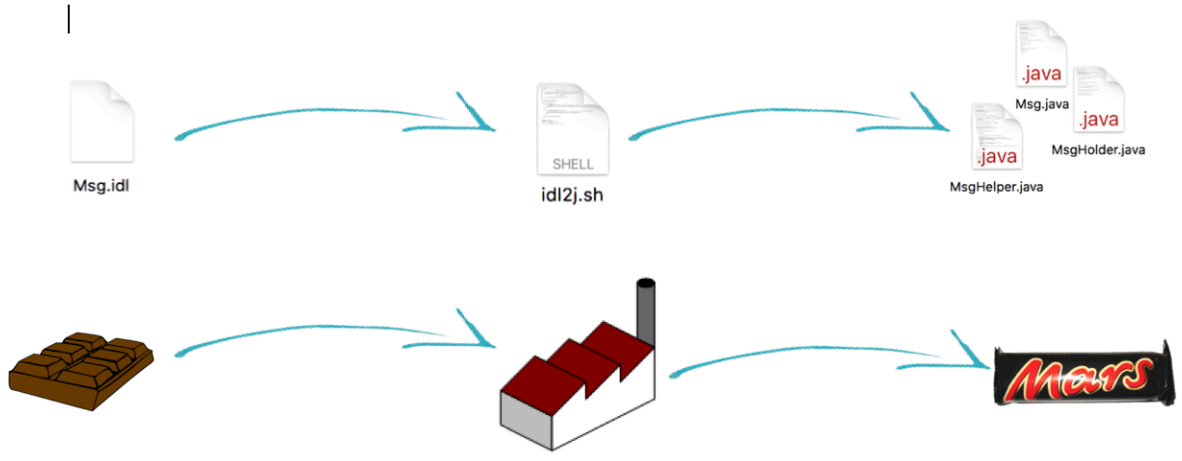
Connexion DDS

Dans le cadre du projet, nous avons besoin de trouver un une technologie nous permettant d'envoyer les données recueillies en temps réel. En effet, les spoonphones récupèrent les données des tags mais chacun de leur côté. Il nous faut donc pouvoir échanger les données entre les téléphones, pour finalement les envoyer à l'application du coach.

Principe



Création et mise en place d'une classe java



Class application

Dans chacune des deux applications (voir la suite du rapport), nous créons une classe application afin d'initialiser et de mettre en place tout le système DDS.

```
@Override
public void onCreate() {
    super.onCreate();

    // Configure DDS ServiceEnvironment class to be OpenSplice Mobile
    System.setProperty(ServiceEnvironment.IMPLEMENTATION_CLASS_NAME_PROPERTY,
        Config.DDS_BOOTSTRAP_CLASS);

    // Create a DDS ServiceEnvironment
    ServiceEnvironment env = ServiceEnvironment.createInstance(
        this.getClass().getClassLoader());

    // Get the DomainParticipantFactory singleton
    DomainParticipantFactory dpf =
        DomainParticipantFactory.getInstance(env);

    // Create a DomainParticipant on DomainID
    dp = dpf.createParticipant(Config.DDS_DOMAIN_ID);

    Topic<Msg> topic = dp.createTopic(Config.DDS_TOPIC_NAME, Msg.class);

    // Create a Publisher and a Subscriber (with default QoS)
    Publisher pub = dp.createPublisher();
    Subscriber sub = dp.createSubscriber();

    dw = pub.createDataWriter(topic);
    dr = sub.createDataReader(topic);
}
```

Writer & Reader

Nous avons maintenant accès au Writer (envoie de données) et au Reader (réception de données). Chacun d'eux sera exploité dans les deux applications dont les explications apparaissent dans les deux parties suivantes. Writer pour l'application joueur, et Reader pour l'application du coach.

Finalement, l'utilisation du système DDS s'est avérée efficace et performante. Cependant à la vue de l'utilisation que nous en avons fait, un serveur aurait aussi pu fonctionner. À l'avenir, comme nous voulons remplacer les 4 balises par des téléphones, et les téléphones par des Tags BeSpoon, le système DDS restera le plus adapté et le plus efficace, notamment avec l'utilisation de topics.

Application joueur

Introduction

Nous avons du créer deux applications différentes pour réaliser les deux opérations essentielles de notre système. Les deux applications sont vraiment extrêmement différentes c'est pour cela que nous avons décidé d'en créer deux avec des rôles bien différents. L'application coté Joueur se charge de récupérer en temps réel les informations (distances dans ce cas là) des différents Tags connectés au téléphone du joueur. Nous devons aussi mettre en place un système permettant de communiquer. L'application du joueur se chargera de poster les messages en question sur le serveur DDS. Ces messages sont créées directement par l'application après les calculs de toutes les statistiques du joueur. Ces statistiques sont calculé grâce à plusieurs algorithmes directement implémentés dans l'application. Nous avons, pour cela beaucoup structuré le code pour avoir un code clair et facile à comprendre mais aussi pour faciliter les mises à jour temps réel des informations du joueur.

Création de session

La première activité apparaissant à l'écran est une activité composé d'un formulaire très simple permettant de récupérer les informations essentielles permettant l'affichage des données sur l'application du coach.

```
Bundle extras = getIntent().getExtras();
String vPlayerName = "";
String vPlayerNum = "";
String vPlayerTeam = "";
if (extras != null) {
    vPlayerName = extras.getString("name");
    vPlayerNum = extras.getString("number");
    this.aLargeur = extras.getDouble("largeur");
    this.aLongueur = extras.getDouble("longueur");
    vPlayerTeam = "" + extras.getInt("team");
}
```

Informations du joueur

Dans un premier temps nous récupérons les informations relatives au joueurs. Il nous faut créer un identifiant unique pour tous les joueurs présents sur le terrain. On choisit alors de récupérer le nom du joueur son numéro mais aussi sont équipe qui est indispensable pour différentier les joueurs sur l'application du coach. On utilise un entier pour déterminer l'équipe du joueur. Le 1 si il fait partie de l'équipe à domicile et le 2 si il fait partie de l'équipe à l'extérieur. Toutes ces informations permettent de déterminer uniquement chaque joueur. Même si un joueur à le même nom et le même numéro que son adversaire il sera différencier par son équipe. Enfin si deux joueurs de la même équipe ont le même nom ils seront différenciés par leur numéro.

Informations du terrain

Pour avoir un affichage le plus proche du réel, on ne peut pas se contenter d'avoir une longueur unique. Ils faut que nous initialisions les longueurs du terrain avec le terrain réel. Il faut que chaque joueur entre la taille du terrain.

Dans un prochain temps, si nous continuons ce projet. Nous pourrons ajouter un topic DDS pour pouvoir transporter les informations vers tous les téléphones directement depuis l'application du coach.

Nous initialisons donc les points du terrain avec la longueur et largeur réel. Ce qui nous permet d'avoir un affichage optimal de la position du joueur. Ces informations doivent être entrée dans l'algorithme de trilatération pour calculer la position.

```
Point aPointTL = new Point();
Point aPointTR = new Point(this.aLargeur*100,0.0,0.0);
Point aPointBL = new Point(0.0,this.aLongueur*100,0.0);
Point aPointBR = new Point(this.aLargeur*100,this.aLongueur*100,0.5);

ArrayList<Point> pTabPoint = new ArrayList<Point>();
pTabPoint.add(aPointTL);
pTabPoint.add(aPointBL);
pTabPoint.add(aPointBR);
pTabPoint.add(aPointTR);
```

Sécurisation

Tous formulaires peuvent faire buguer une application entière si les informations attendues par le programmeur ne sont pas les bonnes. Il existe plusieurs types de sécurisation. Nous pouvons dans un premier temps sécuriser les données entrées par l'utilisateur en forçant les EditText à ne prendre en paramètre qu'un certain type de données (Texte, Nombres entiers, Nombres décimaux, mots de passe). Dans un second temps on doit vérifier que l'utilisateur rentre bien toutes les informations nécessaires. Pour cela il faut faire de nombreux tests afin de voir quels sont les champs corrects et les champs vides. En fonction des champs qui ne sont pas remplis on peut demander explicitement à l'utilisateur de les remplir, sinon il ne pourra pas aller plus loin dans l'application. Enfin quand toutes les vérifications ont été effectuées ont peut passer à l'activité suivante.

Connexion des Tags

Maintenant que nous avons passé la partie initialisation. Nous pouvons nous intéresser aux informations plus intéressantes. Nous devons maintenant connecter les différents Tags aux téléphones. Nous avons eu de nombreux problèmes de connections. Le premier problème était due à Android Studio qui ne devait pas bien compiler les différentes librairies et fichier .aidl nécessaires pour la détection des Tags par l'application. Les classes étaient bien reconnu mais le service ne s'initialisait pas correctement. Les Tags ne pouvaient alors pas être reconnus par notre application. Après le passage sur Éclipse conseillé par M. Hamouche le même code se comportait très bien et permettait de reconnaître très facilement les tags et de récupérer les informations d'identifiant et de distances dont nous avions besoin pour la suite du projet.

Les Tags

Bespoon propose plusieurs interfaces assez bien faites pour synchroniser et récupérer les informations des Tags. Ces bibliothèques permettent de gérer toutes les fonctions de l'UWB. Les Tags communiquent en interne avec le service UWB du téléphone pour calculer une distance. Une onde fait un aller retour avec le Tag depuis le téléphone pour calculer la distance. Cette distance est récupérée grâce à l'identifiant du Tag en question. Nous expliquerons plus tard comment nous gérons les évènements liés aux Tags.

UwbManager

L'UwbManager permet de gérer l'attachement et le détachement des différents Tags. Il implémente différentes méthodes permettant de gérer la synchronisation des Tags mais aussi l'UWB elle même. Il permet d'activer ou désactiver la radio (UWB) mais aussi d'avoir le statut de cette dernière. Il permet aussi d'avoir un accès à tous les Tags attachés. Il permet aussi de récupérer les informations concernant les Tags attachés.

UwbDevice`

Cette classe permet de gérer directement les informations des Tags. Récupérer l'adresse MAC de ceux ci et il définit aussi plusieurs listener important pour récupérer les informations dont nous avons besoin.

Connexions

Les Listeners

Nous devons créer deux Listeners pour récupérer les informations des Tags mais aussi les Tags eux même. Ils faut qu'on Listerner écoute la synchronisation des différents Tags c'est le UwbManager.UwbManagerListener et un autre qui écoute les modifications de distances de chaque Tag par rapport au téléphone c'est le UwbDevice.UwbDeviceListener. UwbManagerListener doit être implémenté sur le UwbManager. Ces deux Listeners implémentes des interfaces. Ce qui nous oblige à redéfinir plusieurs méthodes qui nous serons très utiles pour utiliser l'UWB. Dans l'UWBManagerListener nous devons redéfinir toutes les fonctions permettant d'attacher et de retirer les Tags par exemple ou alors les méthodes appelées lorsque le statut de l'UWB change.

Dans l'UWBDeviceListener nous devons redéfinir les méthodes appelées lorsque la distance change avec le Tag par exemple. Toutes ces fonctions sont essentielles et c'est sur celles ci que nous avons du travailler pour mettre en place l'infrastructure complexe de notre système.

Traitement des attachements et détachements.

Nous avons alors besoin de redéfinir les différentes fonctions de l'UWBManagerListener. Elles se trouvent dans la classe UWBManagerHelper qui implémente cette interface.

Dès qu'un Tag est détecté il est ajouté à une liste de Tag qui nous permet d'avoir une vision globale de tous les Tags présents.

```
public void onUwbDeviceAttached(UwbDevice device) {
    this.myActivity.mUwbAttachedDevices.add(device);
    if(!this.myActivity.myTagHandler.containsTAG(device)){
        this.myActivity.myTagHandler.addTag(new TAGDevice(device,this.myActivity.getDeviceIdentity(device)));
        device.registerListener(this.myActivity.getUwbDeviceListener());
        this.myActivity.nbAttachedDevice++;
        Toast.makeText(this.myActivity, this.myActivity.nbAttachedDevice +"device(s) Attached ", Toast.LENGTH_LONG).show();
        Log.i(TAG,(this.myActivity).getDeviceIdentity(device)+" attached");

        this.myActivity.updateTagInfos();
    }
}
```

Nous avons aussi défini une nouvelle classe myTagHandler. Cette classe implémente de nombreuses méthodes permettant de prendre en main tous les aspects de la récupération et ensuite de l'affichage des informations. Cette classe regroupe toutes les méthodes dont nous avons besoin. Nous avons créé cette classe par un souci de structuration et de compréhension du code. Cette classe ne comporte qu'un seul attribut qui est une liste de TagDevice. Nous avons aussi créer cette classe qui regroupe les méthodes concernant les Tags. Elle permet de regrouper le nom, l'ID du tag mais aussi sa distance dans un objet qui est beaucoup plus simple à utiliser et à prendre en main dans les autres méthodes du code.

Le joueur

Encore dans un souci de structuration du code nous avons créé une classe qui est la base de tous les algorithmes. Elle regroupe toutes les références vers les fonctions et les algorithmes permettant de mettre à jour toutes les informations du joueur (Distance, Vitesse, Hauteur, Position).

Un seul appel à la fonction updatePlayer() permet de lancer toutes les fonctions de mise à jour de tous ses attributs en faisant appel aux bons algorithmes mis en place.

Pour calculer la vitesse et la distance parcourue nous avons dû mettre un place un système de mémoire. Nous avions dans un premier temps utilisé une ArrayList comprenant tous les points calculés pour un joueur. Mais nous avons pensé, que due à la grande vitesse de mise à jour des distances, la liste serait extrêmement énorme et les performances de notre application en serait amoindrie. Nous avons donc uniquement gardé les points précédant que nous mettons à jour à chaque fois. A la fin des calculs nous mettons le point courant dans le point précédant et nous attendons une nouvelle valeur de point.

```
public void updatePlayer(ArrayList<Double> newTabDist){
    for(Double mD: newTabDist){
        if(mD == 0.0){
            return;
        }
    }

    //Premier Point Initialisation du joueur
    if(this.lastPoint == null){
        this.coord.updatePoint(newTabDist);
        this.lastPoint = new Point(this.coord.getX(), this.coord.getY(), this.coord.getZ());
    }
    //Deuxieme Point
    else{
        this.lastPoint = new Point(this.coord.getX(), this.coord.getY(), this.coord.getZ());
        this.coord.updatePoint(newTabDist);
    }

    this.coord.updatePoint(newTabDist);
    this.updateHauteur();
    this.updateVitesse();
    this.updateDistance();
}
```

Les différents algorithmes de vitesse et positions ont été déjà expliqués. Mais nous avons aussi mis en place un algorithme de calcul de distance parcourue. Cet algorithme utilise uniquement deux points consécutifs. Cependant, les mesures effectuées par les Tags n'étant pas vraiment précises dans le temps l'algorithme ne marche pas de façon optimale.

```
public void updateDistance(){
    Point VN2 = this.lastPoint;
    Point VN1 = this.coord;
    double tmp = this.distance;
    this.distance =  tmp + this.calculDistance(VN1.getX(), VN1.getY(),VN2.getX(),VN2.getY());
    Log.d(TAG, "Distance : " + this.distance);
}
```

Nous aurions pu avec un peu plus de temps reprendre l'idée de la liste entière comportant tous les points du joueur. Avec un traitement un peu plus précis nous aurions pu obtenir des résultats plus concluants.

Temps réel

Récupération et calcul des données

Pour mettre à jour en temps réel les informations des joueurs nous avions pensé à faire un Thread qui permettrait d'échantillonner les distances reçues par l'application. Mais l'UWBDeviceListener implémente directement une fonction CallBack est appelé à chaque fois que la distance d'un Tag est mise à jour c'est la fonction onDistanceChanged. Elle possède en paramètre de nombreuses informations intéressantes pour nous.

Il possède le device qui appelle cette fonction. La distance calculée, le timestamp qui permet de dater les mesures. Mais aussi la précision des mesures. C'est dans cette méthode que nous allons effectuer tous nos traitements. Nous avons décidé de ne pas mettre à jour les informations à chaque fois qu'une distance est modifié mais lorsque les distances des quatre tags ont été modifiées. Cela permet d'optimiser un peu le traitement et de ne pas faire de calculs inutiles.

```
@Override
public void onDistanceChanged(UwbDevice device, int accuracy, long timestamp, float distance) {
```

```
@Override
public void onDistanceChanged(UwbDevice device, int accuracy, long timestamp, float distance) {

    //String data = "s=uwb,t=" + ((MainActivity)mCtx).getDeviceIdentity(device) + ",tu=" + timestamp + ",ts=" + System.currentTim

    //Log.d(TAG, "timestamp : " + timestamp);
    flag += 1;
    int i = this.myActivity.mUwbAttachedDevices.indexOf(device);


    this.myActivity.myTagHandler.updateTagDistance(device, distance*100);
    //Log.d(TAG, "tabDist : " + this.myActivity.aTabDist.toString());

    this.myActivity.aTabDist.set(i, (double)(distance*100));
    if(this.myActivity.nbAttachedDevice>3){
        if(flag%4==0){
            this.myActivity.myPlayer.updatePlayer(this.myActivity.aTabDist);

            this.myActivity.updatePlayerInfos();
            String vStrToSend = this.myActivity.myPlayer.createStringToSend();
            this.myActivity.sendStringToDDS(vStrToSend);

            /*if(flag%100==0){
                Toast.makeText(this, vStrToSend, Toast.LENGTH_LONG).show();
            }*/
        }
    }
    else{
        //Log.d(TAG,"Vous devez attacher au minimum 4 TAGs");
        if(flag%100==0){
            Toast.makeText(this.myActivity, "Vous devez attacher au minimum 4 TAGs", Toast.LENGTH_LONG).show();
        }
    }

    this.myActivity.updateTagInfos();
}
```

La fonction s'occupe de mettre à jour la distance du Tag en question. Il récupère l'index du device met à jour la table des distances. Pour effectuer un calcul précis de position et surtout pour avoir la hauteur il nous faut un minimum de 4 points. Nous effectuons alors uniquement les calculs si plus de 3 tags sont attachés. Les algorithmes sont faits de tel sorte qu'il peuvent fonctionner avec autant de Tag que possible.

La fonctions se charge ensuite d'appeler la fonction updatePlayer() qui permet de mettre à jour tous les informations.

Affichage des informations

Pour réaliser une interface assez pratique pour le joueur si il porte son téléphone sur le bras. Nous avons affiché les informations de tous les Tags. C'est informations ne sont pas très intéressantes pour le joueur elles servent surtout pour débuguer l'application. Par contre il peut suivre en temps réel sa vitesse et sa distance parcourue. Nous aurions pu rajouter un affichage de la hauteur max et de la vitesse max qui peut être intéressant après une performance ou n'importe quel match.

Pour afficher ces informations nous avons utiliser deux listesView. Une pour les informations des Tags une autres pour les informations du joueur. Ces deux listes sont mises à jour en temps réel. Les méthodes permettant de mettre à jour ces deux listes sont appelées dans la méthode onDistanceChanged(). Pour afficher des informations dans une listeView il faut utiliser un Adapter. Nous avons utilisé une SimpleAdapter qui permet très facilement de faire le lien entre les identifiants des éléments d'un layout que nous avons définis au préalable et d'une HashMap. La clé de la HahsMap et l'identifiant du TextView et la valeur, la donnée que nous voulons mettre à l'intérieur.

Nous avons alors mis en place deux fonctions permettant de créer ces deux HahsMaps très facilement dans les classes Joueur et TagHandler. Ces deux classes se chargent de construire automatiquement les deux HashMap qui seront très facilement affichées dans les listView par le biais des adapters. Nous allons prendre en exemple la fonction utilisée pour afficher les informations des Tags

```
public ArrayList<HashMap<String, String>> getMapForDisplay(){
    ArrayList<HashMap<String, String>> listItem = new ArrayList<HashMap<String, String>>();
    for(TAGDevice vP:this.TagDevicesList){
        int index = this.TagDevicesList.indexOf(vP);
        //Log.d("Index" , ""+ index);
        String position = "";
        switch(index){
            case 0:
                position = "Coin Haut Gauche";
                break;
            case 1:
                position = "Coin Bas Gauche";
                break;

            case 2:
                position = "Coin Bas Droit";
                break;

            case 3:
                position = "Coin Haut Droit";
                break;

        }

        HashMap<String,String> map = new HashMap<String,String>();
        String Name = vP.getaTagName();
        map.put("name", Name);
        map.put("position", position);
        map.put("distance", "" + vP.getaTagDistance());
        listItem.add(map);
    }
    return listItem;
}
```

On créer alors une HashMap pour chaque Item que nous voulons afficher. Nous avons au préalable créer un layout qui viendra recevoir les informations que nous lui envoyons par l'intermédiaire de l'adapter.

Nous devons aussi pour initialiser les Tags au bon endroit avoir l'information d'où ils sont placés pour l'algorithmes. Comme ils sont détectés au fur et à mesure ils s'initialisent dans l'ordre il faut que l'utilisateur sache ou il doit positionner les différents Tags. Donc la liste permet de voir quel Tag on doit poser à quel endroit. On créer une liste de HashMap pour tous les items que nous voulons afficher. Ensuite il faut définir l'adapter permettant de faire le lien entre les deux.

```
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <TextView
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:textColor="#000000">
    </TextView>

    <TextView
        android:id="@+id/distance"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:textColor="#000000">
    </TextView>
</LinearLayout>
```

```
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:text="Position : "
        android:textColor="#000000">
    </TextView>
    <TextView
        android:id="@+id/position"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:textColor="#000000">
    </TextView>
</LinearLayout>
```

On doit alors récupérer la référence de la listView dont on veut changer les informations. On créer ensuite une ArrayList<HashMap<String, String>> grâce à la fonction expliquée plus haut.

```
public void updateTagInfos(){
    ListView maListViewPerso = (ListView) findViewById(R.id.malistTagInfo);
    ArrayList<HashMap<String, String>> listItem = this.myTagHandler.getMapForDisplay();
    SimpleAdapter mSchedule = new SimpleAdapter(this.getBaseContext(), listItem, R.layout.simple_tag_info,
            new String[] {"name", "distance", "position"}, new int[] {R.id.name, R.id.distance, R.id.position});
    maListViewPerso.setAdapter(mSchedule);
}
```

Il suffit alors de faire correspondre la HashMap avec les id du layout. Pour cela il faut créer deux tableaux de String et d'Integer que l'on passe directement à l'adapter et qui se charge du reste. Il faut aussi précisé sur quel layout nous voulons afficher ces informations et le tour est joué.

Nous devons répéter ces étapes deux fois. Pour les infos des tags mais aussi pour celles du joueur. La démarche est exactement la même il faut juste recréer un layout si l'on veut une disposition différente, créer la liste de hashMap pour définir chaque item et ensuite les faire correspondre à l'aide de l'adapter.

Envois des informations

Maintenant que toutes les informations sont calculées et affichées sur l'application du joueur il faut maintenant que le coach y ait accès. On a expliqué plus tôt les fonctionnements du DDS. Il faut maitnenant expliquer comment l'application envois les données sur le serveur et sous quelle forme.

La classe joueur s'occupe de calculer toutes les informations mais possède aussi une méthode permettant de créer la String à envoyer. Elle est appelé après que tous les algorithmes soient terminés dans la méthode onDistanceChanged().

```
public String createStringToSend(){
    String vCoordString = this.convertPourcentage(this.coord);
    return this.aName + "-" + this.aNum + "-" + this.aTeam + "-" + vCoordString + "-" + this.coord.getZ() + "-" +
            this.vitesse.getVitesse() + "-" + this.distance.toString();
}
```

La String comporte tout d'abord les paramètres permettant de différencier les joueurs comme nous avons expliquer au tout début. Soit le nom du joueur, son numéro, mais aussi son équipes (Pour rappel : 1 si il fait partie de l'équipe à domicile et 2 si il fait partie de celle à l'extérieur). Ensuite elle comporte vCoordString qui est seulement composé de X et Y.

Pour faciliter l'affichage et puisque c'est l'application du joueur qui contient les informations du terrain. Il faut que l'on convertisse les coordonnées X et Y en pourcentage de terrain. Pour cela on fait appel à la méthode convertPourcentage() de la classe Joueur.

```
private String convertPourcentage(Point p){
    double x = p.getX();
    double y = p.getY();

    int newX = (int) (x/this.myActivity.aLargeur);
    int newY = (int) (y/this.myActivity.aLongueur);

    return "" + newX + "-" + newY;
}
```

Ensuite la String comporte aussi les valeurs de hauteur, de vitesse et enfin de distance.

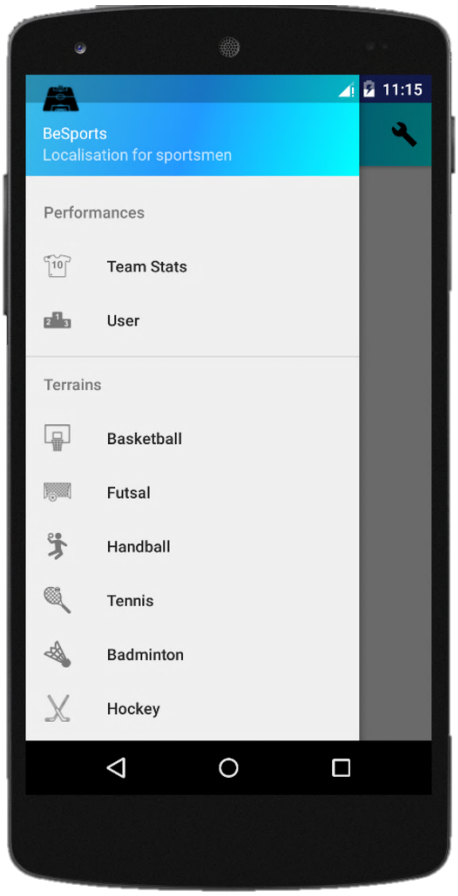
Pour envoyer les informations ils suffit alors de les envoyer dans le writer de DDS avec la méthode sendStringtoDDS().

```
public void sendStringToDDS(String pS){
    try {
        this.MsgWriter.write(new Msg(idTel, pS));
        Log.d(TAG,"Message envoyé...");

    } catch (TimeoutException te) {
        Log.d(TAG,"Message non envoyé ...");
    }
}
```

Application coach

Cette application aura pour but principale d'interagir avec l'utilisateur final. En effet, c'est donc cette application qui nécessite une interface plus travaillée et agréable d'utilisation. Nous pouvons finalement décomposer cette application en 3 principales parties :

- Main Activity (gestion globale de l'application)
- Affichage (gestion de l'affichage des joueurs en temps réel sur un terrain)
- Statistiques Activity (gestion des performances des joueurs en temps réel)



Une interface simple et efficace permet d'accéder à toutes les fonctionnalités de l'application directement depuis un menu à swiper. Les six sports les plus pratiqués en indoor sont inclus dans l'application et contiennent un terrain adapté, un icone de joueur adapté, et les mesures nécessaires à l'étalonnage du terrain.

Réception de données

Afin de pouvoir mettre à jour continuellement les positions et statistiques des joueurs, nous exploitons une String contenant toutes les informations. C'est le système DDS vu plus tôt dans le rapport qui nous permet d'assurer la bonne réception de cette String.

Nous faisons donc appel au Reader fourni par la class Application dans le onCreate() de la class MainActivity. Initialisation du Reader :

```
private void initDDS(){
    this.MsgWriter = ((ChatApplication) getApplication()).getWriter();

    System.setProperty(ServiceEnvironment.IMPLEMENTATION_CLASS_NAME_PROPERTY, Config.DDS_BOOTSTRAP_CLASS);
    this.MsgReader = ((ChatApplication) getApplication()).getReader();
    this.MsgReader.setListener(new DataReaderListener<Msg>() {...});
}
```

Le Listener créé contient des méthodes @Override. Nous exploitons la méthode onDataAvailable() qui est appelée à chaque réception de nouvelle donnée.

```
@Override
public void onDataAvailable(DataAvailableEvent<Msg> dataAvailableEvent) {...}
```

Voici le code qui permet de récupérer la String envoyée et de la stocker dans un attribut :

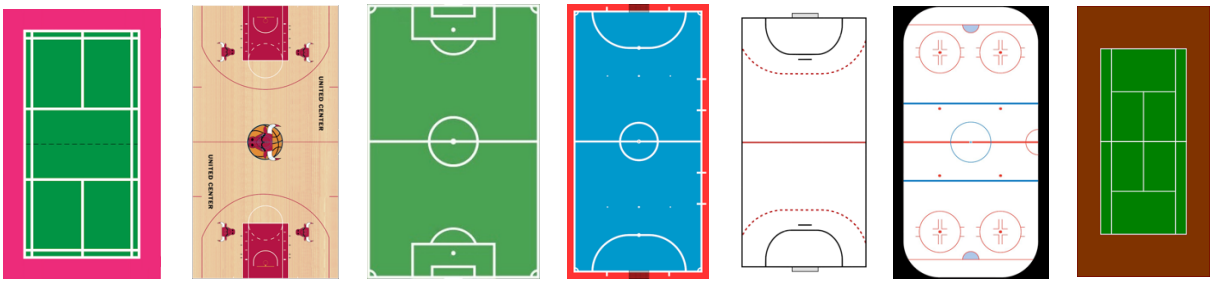
```
StringBuilder vStr = new StringBuilder("");
Sample.Iterator<Msg> it = dataAvailableEvent.getSource().take();
while(it.hasNext()) {
    Msg notif = it.next().getData();
    vStr.append( notif.message);
}
aStringRecup = vStr.toString();
```

Tout le décodage de cette String reçue se fait dès la réception et donc dans cette même méthode onDataAvailable() (voir le fichier source).

Classe Affichage

Cette classe gère l'affichage des points localisés sur un terrain. Elle étend un objet de type View et est contenue dans ce qu'on appelle un Fragment. Elle comprend 4 principales fonctions : Affichage des joueurs, mise en pause de l'affichage, possibilité de dessiner des stratégies lors d'une mise en pause, et possibilité d'enregistrer les stratégies dessinées.
La méthode appelée *onDraw(Canvas pCanvas)* permet de dessiner sur une application Android. C'est cette méthode qui servira à dessiner les joueurs, et faire en sorte qu'ils se déplacent en temps réel. En effet cette méthode est appelée sans cesse, dès que possible afin de « rafraichir » l'écran avec les nouvelles positions grâce à l'emploi de la méthode *invalidate().*
Nous affichons des icones représentant les joueurs grâce aux objets de type *Bitmap* et *BitmapDrawabale* d'Android, via la méthode *pCanvas.drawBitmap().*

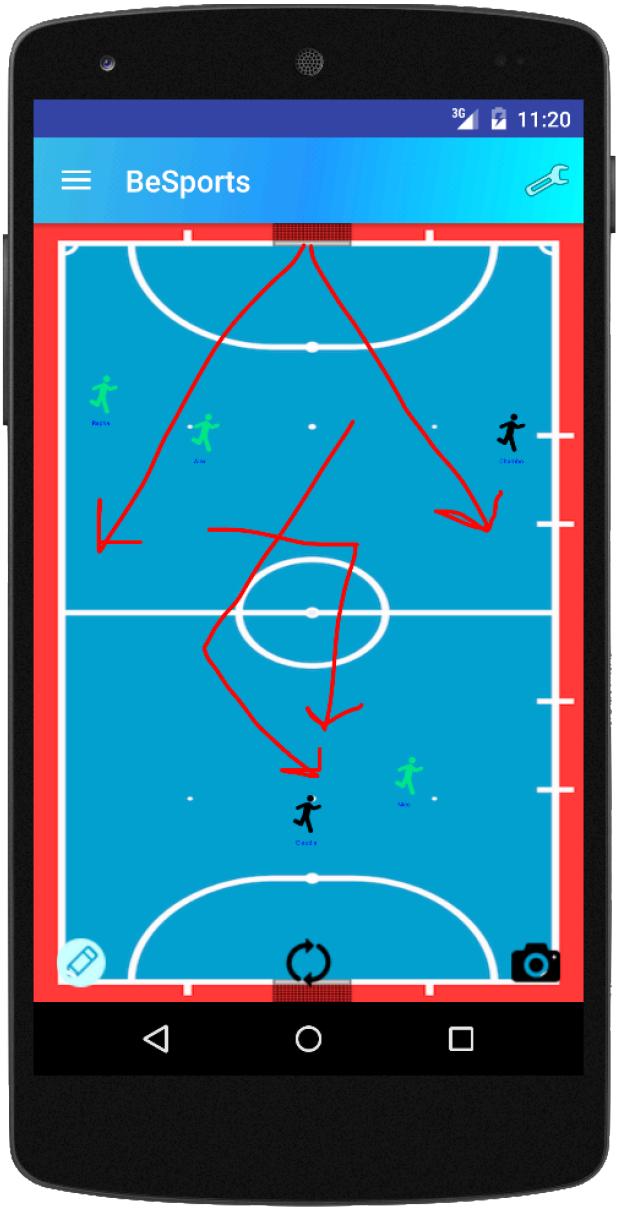
Les terrains qui serviront de « fond » sont eux directement définis dès lors que le choix du sport est réalisé. Ils seront placés en Background du Fragment utilisé.



Le rendu est le suivant :

Drawing Mode

Nous avons souhaité que le coach puisse mettre en pause l'affichage de ses joueurs pour pouvoir valoriser certains positionnements. Afin qu'il puisse accéder à cette option le plus facilement possible, nous avons mis en place une réaction à ce qu'Android appelle le **DoubleTap**. Lorsque l'utilisateur touche 2 fois l'écran assez rapidement, les positions des joueurs sont stockées et l'écran figé.
Une fois l'affichage figé, il est possible pour le coach de dessiner une stratégie, les mouvements qu'il souhaite imposer à ses joueurs, les positionnements intéressants etc.
Il lui sera par la suite possible d'enregistrer ses « notes » dans sa galerie de photos pour pouvoir les consulter ultérieurement.



L'objet Java utilisé est un **GestureListener** qui nous permet d'utiliser la méthode **onDoubleTap**(MotionEvent pMotionEvent). Dès lors que l'utilisateur effectue un DoubleTap cette méthode sera appelée et son contenu effectué. Les dernières positions connues sont alors utilisées pour dessiner une dernière fois les joueurs. Puis la méthode onDraw() ne nécessite plus d'être appelée constamment avec invalidate().

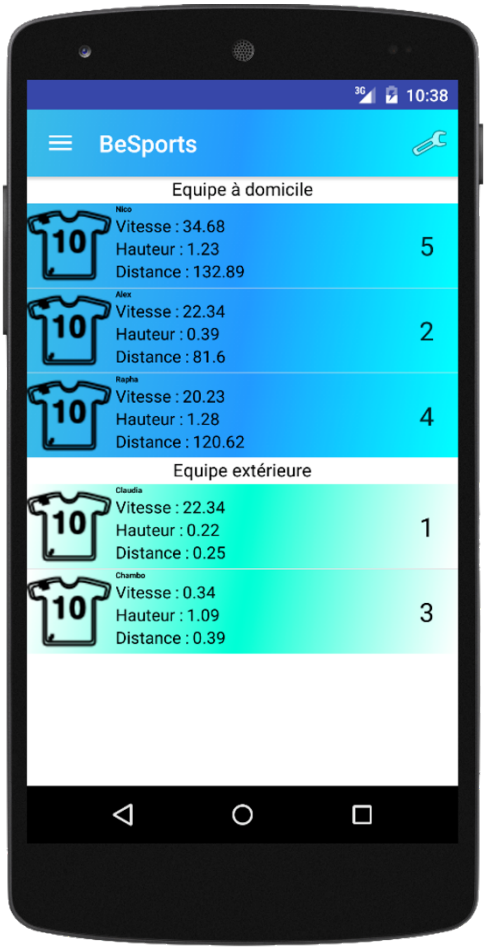
La méthode utilisée pour dessiner sur le canevas est onTouchEvent(). Elle nous permet d'accéder aux trois évènements suivants :
- ACTION_DOWN
- ACTION_MOVE
- ACTION_UP

Ils nous permettent de créer un système de dessin sur le canevas.

Statistiques Activity

Afin de pouvoir suivre pleinement ses joueurs en temps réel, un accès à leurs performances est possible pour le coach grâce à ce mode. Les données récupérées, à savoir la vitesse, la hauteur, et la distance parcourue par le sportif sont affichées à l'aide d'une ListView.

Les éléments de cette liste sont directement liés aux valeurs récupérées dans la MainActivity par connexion DDS, puis affichées dès que la valeur change.
Le nom et numéro du joueur sont bien sûr aussi affichés.



Remplissage de la liste :

```
this.maListViewPerso = (ListView)
view.findViewById(R.id.listviewmyteam);

ArrayList<HashMap<String, String>> listItem =
this.aMainActivity.getMyTeam().getListForDisplay();


SimpleAdapter mSchedule = new SimpleAdapter
(this.getActivity().getBaseContext(), listItem,
R.layout.display_player_stat,
        new String[] {"img", "playername", "vitesse",
"hauteur", "distanceparcourue"}, new int[] {R.id.img,
R.id.playername, R.id.vitesse, R.id.hauteur,
R.id.distanceparcourue});

maListViewPerso.setAdapter(mSchedule);
```

Accès aux stratégies enregistrées

Comme mentionné précédemment, il est possible d'enregistrer les stratégies dessinées en appuyant sur le petit icone d'appareil photo lorsqu'on est en DrawingMode. Afin d'éviter à l'utilisateur de devoir quitter l'application, puis de naviguer jusqu'à sa galerie de photos, et d'y retrouver les captures qu'il a effectuées, nous avons mis en place un système plus commode. En effet, les stratégies enregistrées depuis l'application *BeSports* sont toutes stockées dans un même répertoire appelé *besports* sur la mémoire SD disponible. De sorte que si des photos sont prises entre 2 sessions avec le téléphone, celles-ci ne seront pas mélangées avec les stratégies sauvegardées. Puis nous avons finalement créer un accès direct à ce répertoire depuis le menu de notre application. Il est donc possible au coach de consulter ses différentes stratégies dessinées sans avoir à quitter son application *BeSports*. Lors de la création de la *ListView*, nous mettons en place un *SimpleAddapter* afin de remplir correctement la liste.
[Screen Sratégie] [Screen Selection] [Screen Pellicule]

Le code suivant permet d'écrire sur la mémoire SD disponible et d'enregistrer un fichier dans un répertoire créé :

```
File directory = new File(Environment.getExternalStorageDirectory().toString() +
"/besports/");
directory.mkdirs();
String mPath = ""+ now + ".png";

View v1 = getWindow().getDecorView().getRootView();
v1.setDrawingCacheEnabled(true);
Bitmap bitmap = Bitmap.createBitmap(v1.getDrawingCache());
v1.setDrawingCacheEnabled(false);

File imageFile = new File(directory, mPath);

FileOutputStream outputStream = new FileOutputStream(imageFile);
int quality = 100;
bitmap.compress(Bitmap.CompressFormat.PNG, quality, outputStream);
outputStream.flush();
outputStream.close();
```

Puis l'accès rapide s'effectue comme suit :

```
Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
Uri uri = Uri.parse(Environment.getExternalStorageDirectory().getPath() + "/besports/");
intent.setDataAndType(uri, "*/*");
startActivity(Intent.createChooser(intent, "Open folder"));
```

Conclusion

Nous avons créé une application capable d'assister un coach dans ses entrainements et ses matchs sportifs. La technologie UWB de Bespoon nous a permis de localiser les joueurs de l'équipes dans des sports pratiqués en Indoor. La librairie DDS nous a permis de communiquer efficacement entre les téléphones. Le couplage de ces technologies a abouti à un projet réalisable et fonctionnel d'un point de vue expérimental.

En revanche la technologie BeSpoon utilisée n'est en tout est pour tout qu'une technologie de démonstration. Il est possible de connecter plusieurs Tags à un même téléphone, mais tous les Tags connectés communiquent avec le même canal de fréquences. Nous avons donc observé des déconnections récurrentes lorsque les Tags sont trop éloignés par exemple.

Il serait donc intéressant d'utiliser une version plus spécifique de cette technologie BeSpoon pour améliorer ce projet.