HAI811I – Programmation Mobile

RAPPORT TP2

SALHI Nina
22115492

Encadrant : Seriai Abdelhak Djamel

Master 1 Informatique - IASD

Université de Montpellier
Département d'Informatique

Table des figures

1 Introduction

Dans le cadre de ce TP, nous avons développé 2 applications Android, une qui regroupe tout les exercicies du TP2 et l'autre qui regroupre ceux du TP2-suite. L'objectif principal était de comprendre comment interagir avec les capteurs matériels d'un appareil Android et d'afficher leurs informations de manière structurée et comprendre leur utilisation. Ce rapport explique la logique de conception, la structure du code, et les choix techniques qui ont été faits pour réaliser cette application.

2 Structure de l'application

L'application s'ouvre sur un menu permettant de naviguer entres toutes les activités disponibles, qui correspon-dent chacune à un exercice du TP. Par exemple, le premier exercice "Liste des Capteurs" est dans l'activité nommée SensorListActivity.kt. Voici la page d'acceuil de l'application :

FIGURE 1 – acceuil de l'application

3 Liste de capteurs

L'objectif de cet exercice était de développer une application Android affichant la liste des capteurs disponibles sur un smartphone. L'application se compose de deux activités principales :

— **MainActivity** : Point d'entrée de l'application, permettant de naviguer vers différentes fonctionnalités, dont l'affichage de la liste des capteurs.

— **SensorListActivity** : Activité dédiée à l'affichage de la liste des capteurs disponibles sur l'appareil.

Pour récupérer la liste des capteurs, nous utilisons le `SensorManager`, une classe Android permettant d'accéder aux capteurs matériels. Voici les étapes clés :

Initialisation du SensorManager :

```
val sensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
```

Récupération de la liste des capteurs :

```
val sensorList: List<Sensor> = sensorManager.getSensorList(Sensor.TYPE_ALL)
```

Transformation des données :

Les capteurs sont transformés en une liste d'objets `SensorInfo` pour faciliter leur affichage. Chaque `SensorInfo` contient le nom, le type, le fabricant et la version du capteur :

```
    val sensorInfoList = sensorList.map { sensor ->
    SensorInfo(
        name = sensor.name,
        type = sensor.stringType ?: "Type inconnu",
        vendor = sensor.vendor ?: "Inconnu",
        version = sensor.version
    )
}
```

Pour afficher la liste, nous utilisons une `ListView` avec un `CustomAdapter` (`SensorAdapter`). Cela permet de personnaliser l'affichage de chaque élément, et ce qui concerne la navigation entre les activités est gérée via des `Intent`. Par exemple, pour afficher la liste des capteurs :

```
    btnShowSensors.setOnClickListener {
    val intent = Intent(this, SensorListActivity::class.java)
    startActivity(intent)
}
```

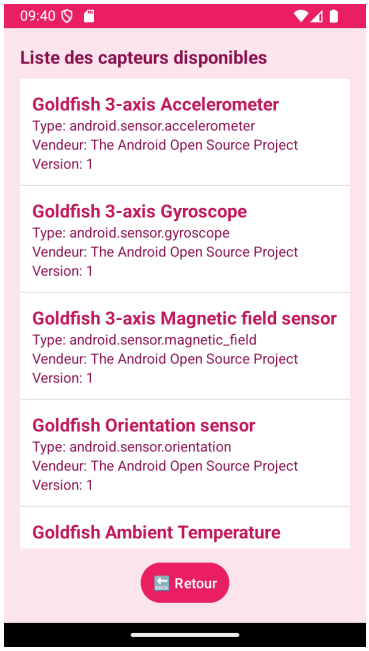

FIGURE 2 – listes des capteurs

4 Détection de présence/absence de capteurs

Dans cet exercice on devait développer une fonctionnalité permettant d'informer l'utilisateur de l'indisponibilité de certains capteurs sur son appareil. L'application vérifie la disponibilité de plusieurs capteurs essentiels (proximité, accéléromètre, gyroscope et lumière) et informe l'utilisateur si l'un d'entre eux est indisponible. Cette vérification est effectuée dans l'activité SensorCheckActivity.
Pour vérifier la disponibilité des capteurs, nous utilisons le SensorManager pour obtenir les capteurs par défaut. Voici un exemple de comment recuperer un capteurs par default, dans le cas de l'accelerometre :

```
val accelerometerSensor = sensorManager.getDefaultSensor
(Sensor.TYPE_ACCELEROMETER)
```

Si un capteur est indisponible (null), un message est ajouté à une liste de capteurs indisponibles, et si tous les capteurs sont disponibles : un message est afficher pour dire que tout est dispo.

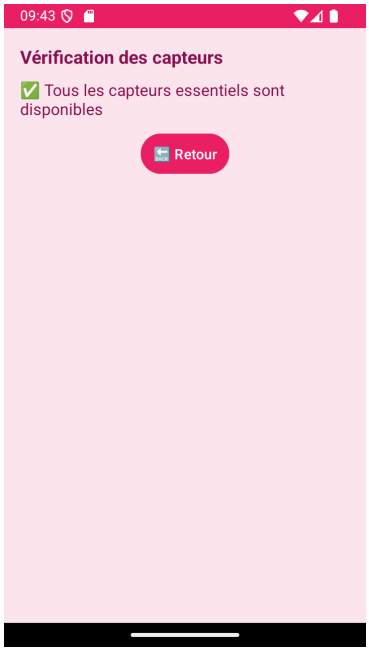

FIGURE 3 – detections des capteurs non disponibles

5 Accéléromètre

Dans cet exercice on a développer une application qui change la couleur de fond en fonction des valeurs de l'accéléromètre. Les valeurs sont réparties en trois catégories : Valeurs faibles : Fond vert, valeurs moyennes : Fond noir et valeurs élevées : Fond rouge.

L'application utilise l'accéléromètre pour mesurer l'accélération de l'appareil et change dynamiquement la couleur de fond en fonction de l'intensité du mouvement. L'activité AccelerometerActivity gère cette logique.

Le SensorManager est initialisé pour accéder aux capteurs matériels et L'accéléromètre est récupéré via getDefaultSensor(Sensor.TYPE_ACCELEROMETER). Si l'accéléromètre n'est pas disponible, le fond est défini en gris et un message est affiché.

Dans onResume(), le SensorEventListener est enregistré pour recevoir les mises à jour de l'accéléromètre et dans onPause(), le SensorEventListener est désenregistré pour économiser les ressources. Les valeurs des axes X, Y et Z sont récupérées dans onSensorChanged et l'accélération totale est calculée à l'aide de la formule :

```
val acceleration = Math.sqrt((x * x + y * y + z * z).toDouble()).toFloat()
```

Trois seuils sont définis pour catégoriser l'accélération :

— Faible (acceleration < 5) : Fond vert.

— Moyenne (acceleration in 5.0..15.0) : Fond noir.

— Élevée (acceleration > 15) : Fond rouge.

De plus, une animation fluide est appliquée pour passer de l'ancienne à la nouvelle couleur.

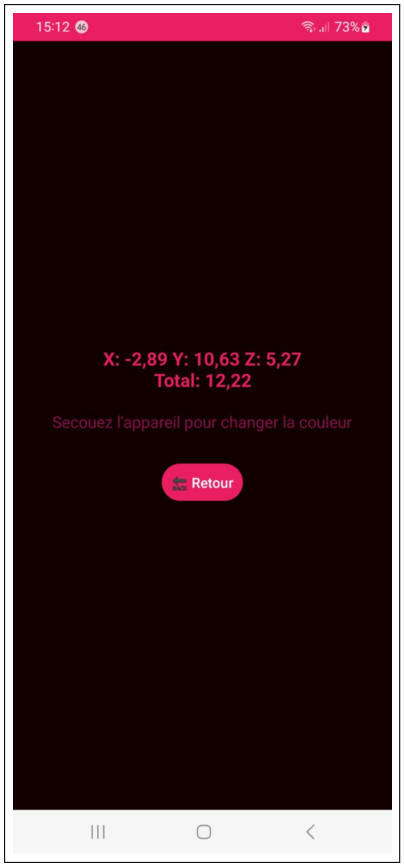

FIGURE 4 – Fond d'écran noir

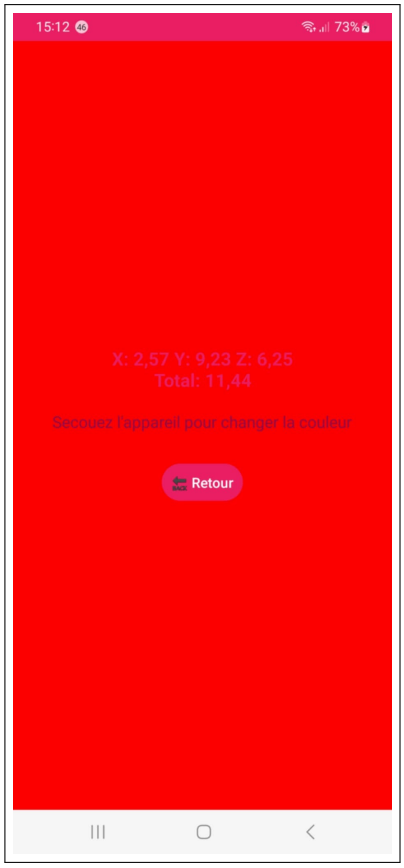

FIGURE 5 – Fond d'écran rouge

FIGURE 6 – Fond d'écran vert

6 Direction

Ici on va développer une application qui affiche la direction du mouvement de l'appareil (gauche, droite, haut, bas) en utilisant les données du capteur de rotation. L'application utilise le capteur de rotation vectorielle (TYPE_ROTATION_VECTOR) pour déterminer l'orientation de l'appareil. En fonction de l'angle d'azimut (orientation horizontale), l'application affiche une flèche directionnelle correspondante (gauche, droite, haut, bas) et met à jour un texte indiquant la direction détectée.
Le SensorManager est initialisé pour accéder aux capteurs matériels et le capteur de rotation vectorielle est récupéré via getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR), si le capteur de rotation n'est pas disponible, un message est affiché. Dans onResume(), le SensorEventListener est enregistré pour recevoir les mises à jour du capteur de rotation.

```
    override fun onResume() {
    super.onResume()
    rotationVectorSensor?.let {
        sensorManager.registerListener(this, it, SensorManager.SENSOR_DELAY_UI)
    }
}
```

Dans onPause(), le SensorEventListener est désenregistré pour économiser les ressources.

```
    override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(this)
}
```

Les valeurs du capteur de rotation sont utilisées pour calculer la matrice de rotation et la méthode SensorManager.getOrientation est utilisée pour obtenir les angles d'orientation (azimut, pitch, roll). Sachant que l'angle d'azimut (orientation horizontale) est converti en degrés et utilisé pour déterminer la direction :

- Haut : Azimut entre -45° et 45°.
- Droite : Azimut entre 45° et 135°.
- Gauche : Azimut entre -135° et -45°.
- Bas : Azimut en dehors de ces plages.

FIGURE 7 – Direction vers la gauche

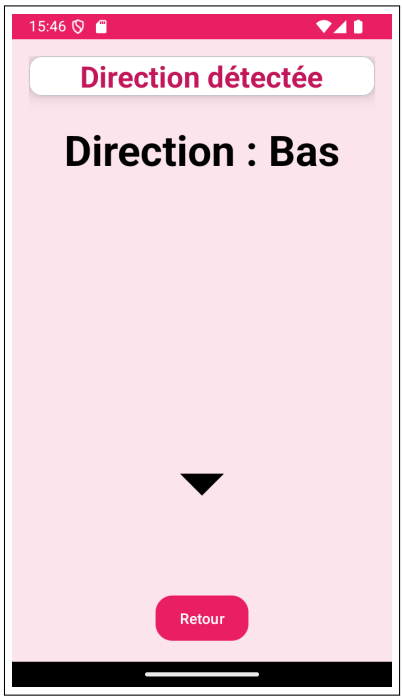

FIGURE 8 – Direction vers le bas

7 Secouer un appareil

L'objectif de cet exercice était de développer une application qui allume et éteint le flash de l'appareil en détectant un mouvement de secousse. L'application utilise l'accéléromètre pour détecter un mouvement de secousse. Lorsqu'une secousse est détectée, le flash de l'appareil est allumé ou éteint en fonction de son état actuel. L'activité FlashActivity gère cette logique.

Le SensorManager est initialisé pour accéder aux capteurs matériels et l'accéléromètre est récupéré via getDefaultSensor(Sensor.TYPE_ACCELEROMETER). Si l'accéléromètre n'est pas disponible, un message est affiché. pour l'enregistrement du listner

Dans onResume(), le SensorEventListener est enregistré pour recevoir les mises à jour de l'accéléromètre et pour le désenregistrement dans onPause(), le SensorEventListener est désenregistré pour économiser les ressources.

Pour le detection des secousses, la force de gravité est calculée à l'aide de la formule :

```
val gForce = Math.sqrt((x * x + y * y + z * z).toDouble()).toFloat()
/ SensorManager.GRAVITY_EARTH
```

Une secousse est détectée si la force de gravité dépasse un seuil (ici, 3) et si un délai minimal (1 seconde) est respecté depuis la dernière secousse.

Pour la gestion du flash, le CameraManager est initialisé pour accéder au flash, l'ID de la caméra arrière est récupéré et le flash est allumé ou éteint en fonction de son état actuel. Une mise a jour de l'interface utilisateur avec l'état du flash est affiché sous forme de texte et d'icône comme ceci :

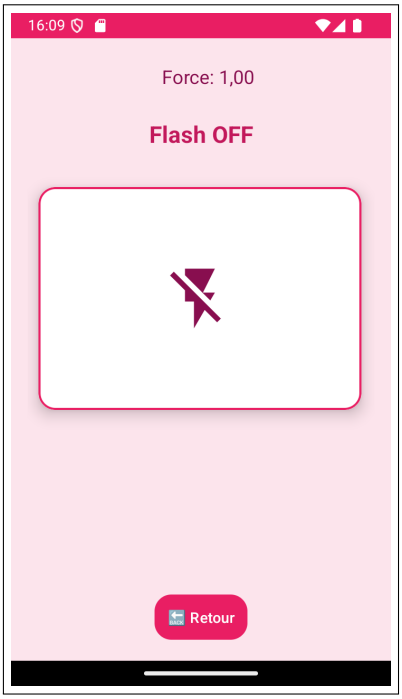

FIGURE 9 – Flash fermer

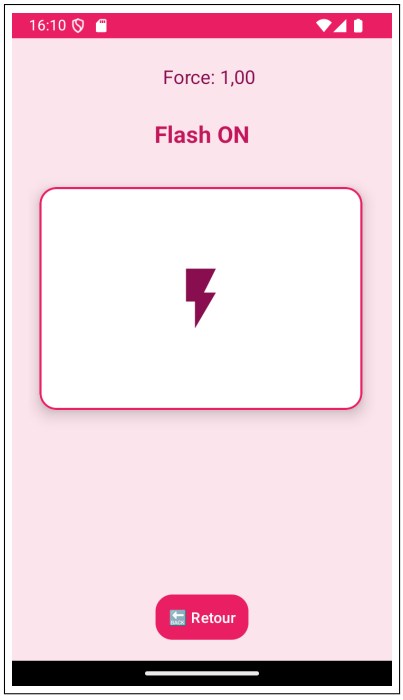

FIGURE 10 – Flash ouvert apres la secousse

8 Proximité

L'objectif de cet exercice était de développer une application qui affiche une image et un texte en fonction de la distance mesurée par le capteur de proximité. Si un objet est proche, l'application affiche une image indiquant la proximité, et si l'objet est loin, une autre image est affichée. Ici on va utiliser le capteur de proximité pour mesurer la distance entre l'appareil et un objet qui dans ce cas est la main de l'utilisateur. Lorsqu'on approche

la main du capteur, l'application affiche une image indiquant "Votre main est proche". Lorsque la main s'éloigne, l'application affiche une autre image indiquant "Loin".

L'application récupère le capteur de proximité du smartphone via le SensorManager pour écouter uniquement le Sensor.TYPE_PROXIMITY, garantissant une gestion efficace des ressources. Si le capteur n'est pas disponible, un message indique "Capteur de proximité non disponible". Lorsque l'application est en arrière-plan (onPause()), l'écoute du capteur est désactivée pour économiser la batterie et lorsque l'application revient au premier plan (onResume()), l'écoute du capteur est réactivée.

FIGURE 11 – La main proche de l'écran

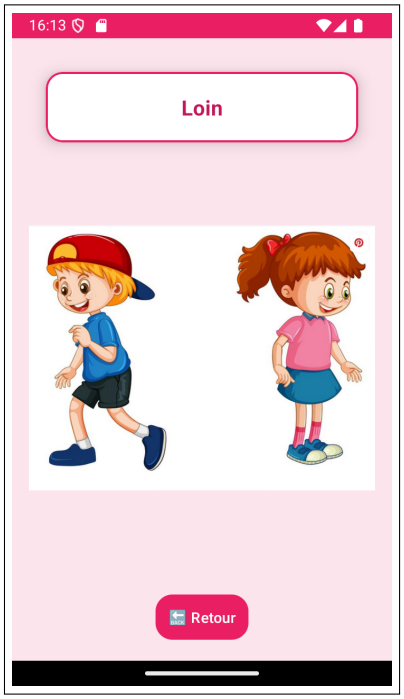

FIGURE 12 – La main loin de l'écran

9 Géolocalisation

L'application a pour objectif de récupérer et afficher en temps réel la position géographique de l'utilisateur en utilisant les services de localisation du smartphone. Elle intègre une carte interactive OpenStreetMap pour afficher la position sur une carte. L'application utilise le GPS de l'appareil pour obtenir les coordonnées de latitude et de longitude, puis affiche ces informations ainsi qu'une carte interactive centrée sur la position de l'utilisateur.

L'application utilise le LocationManager pour accéder aux services de localisation du téléphone. Elle vérifie si le GPS est activé et demande à l'utilisateur d'activer la localisation si nécessaire, si l'utilisateur n'a pas encore accordé la permission de localisation, une demande d'autorisation est affichée. Une fois la permission accordée et le GPS activé, l'application récupère la dernière position connue de l'utilisateur, elle sélectionne la meilleure source GPS disponible en fonction de la précision et de la fiabilité.

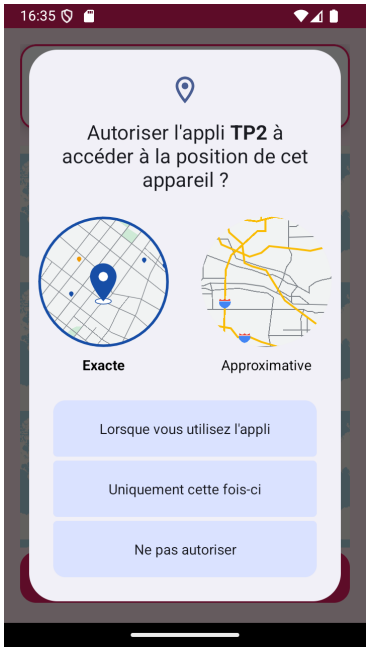

FIGURE 13 – Authorisation d'utilisation de la localisation

Les coordonnées GPS (Latitude et Longitude) sont affichées dynamiquement à l'écran et la carte OpenStreetMap est mise à jour automatiquement pour centrer l'affichage sur la position de l'utilisateur. Aussi, la carte est configurée pour utiliser les tuiles de OpenStreetMap.

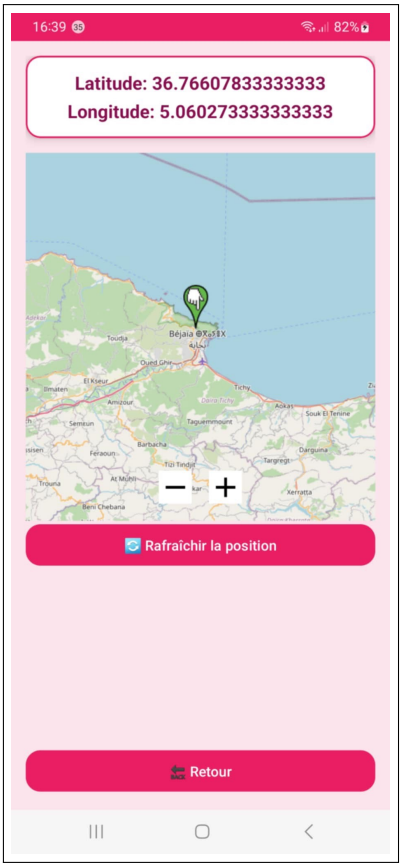

FIGURE 14 – Localisation dezoumer

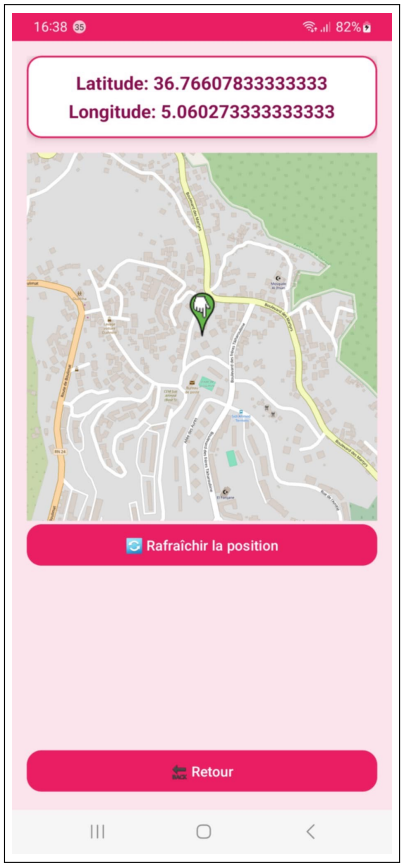

FIGURE 15 – Localisation éxacte

10 Activité et liste

L'objectif de cette application est de présenter une liste de pays avec leurs informations principales (nom, capitale, population, drapeau, etc.) et de permettre à l'utilisateur de consulter les détails de chaque pays en cliquant sur un élément de la liste.

L'activité principale affiche une liste de pays sous forme de carte (RecyclerView), recyclerView est utilisé car il est performant et permet le recyclage des vues pour éviter une surconsommation de mémoire.C haque carte contient le nom du pays et son drapeau. Un Adapter (PaysAdapter) est associé à RecyclerView pour gérer l'affichage des éléments et lorsqu'un utilisateur clique sur un pays, un Intent est lancé pour ouvrir PaysDetailActivity qui affiche des informations détaillées sur le pays sélectionné (capitale, population, région, monnaie, langue, etc.). Un bouton "Retour" permet de revenir à l'écran principal.

En ce qui concerne le modèle de Données (Pays) : il représente un pays avec ses attributs (nom, capitale, population, drapeau, région, etc.). Les données sont encapsulées dans une classe Pays qui peut être transmise entre les écrans. Aussi, la source de Données (PaysData) qui contient une liste statique de pays avec leurs informations. Cette liste est utilisée pour alimenter la RecyclerView de l'écran principal.

Lorsque l'utilisateur clique sur un pays, l'objet Pays correspondant est transmis à

l'écran de détails via un Intent.L'écran de détails récupère cet objet et affiche les informations correspondantes.

FIGURE 16 – Liste de pays

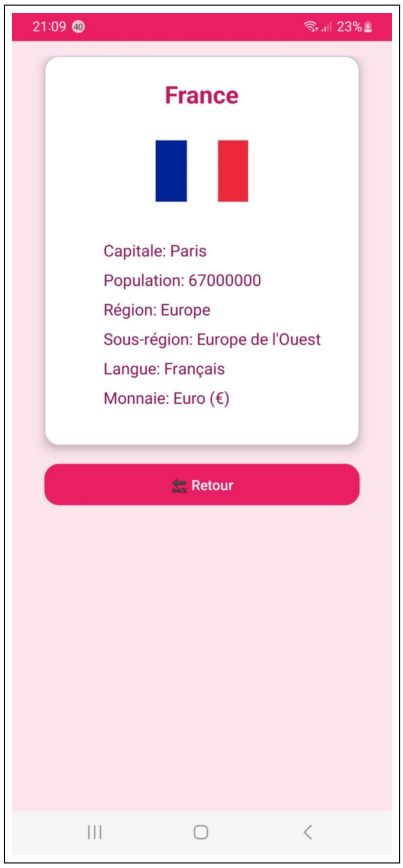

FIGURE 17 – Detaille du pays selectionnée (la France)

11 Fragments

L'objectif ici est de refactoriser l'application de liste de pays en utilisant des fragments pour séparer les fonctionnalités d'affichage de la liste des pays et des détails d'un pays. Cette approche permet une meilleure modularité et une gestion plus flexible de l'interface utilisateur, notamment pour s'adapter à différents formats d'écran (téléphone vs tablette). L'application est maintenant divisée en deux fragments principaux :

— ListePaysFragment :Affiche la liste des pays sous forme de cartes (RecyclerView) et chaque carte contient le nom du pays et son drapeau. Lorsque l'utilisateur clique sur un pays, un événement est déclenché pour afficher les détails du pays.

— DetailPaysFragment : Affiche des informations détaillées sur le pays sélectionné (capitale, population, région, monnaie, langue, etc.) et un bouton "Retour" permet de revenir à la liste des pays.

Pour la communication entre fragments, le ListePaysFragment communique avec l'activité principale (MainActivity) via une interface (OnPaysSelectedListener). Lorsqu'un pays est sélectionné, l'activité principale reçoit l'événement et décide d'afficher les détails

dans le DetailPaysFragment. Le DetailPaysFragment reçoit les informations du pays sélectionné via un Bundle et les détails du pays sont affichés dans une interface utilisateur structurée (nom, capitale, population, drapeau, etc.).

FIGURE 18 – Liste de pays (1er fragment)

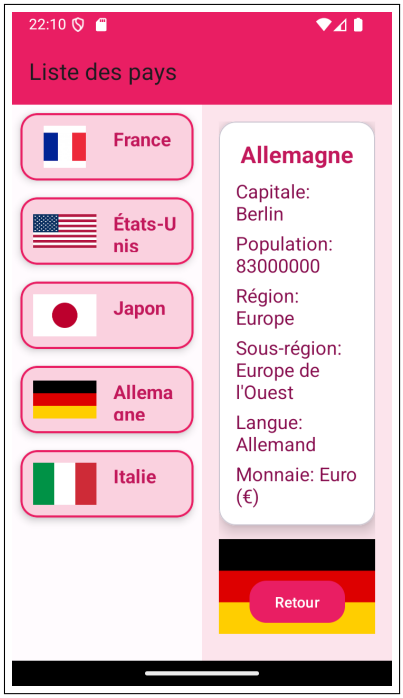

FIGURE 19 – Detaille du pays selectionnée (l'allemagne) 2eme fragment

12 Conclusion

À travers ces exercices, l'application développée couvre plusieurs aspects fondamentaux du développement Android, notamment l'intégration des capteurs du smartphone, la gestion de la localisation, la navigation entre activités et l'affichage dynamique des données. En explorant des concepts clés tels que l'utilisation des capteurs, la géolocalisation, l'affichage interactif avec RecyclerView et la gestion des fragments, j'ai acquis des compétences pour concevoir des interfaces utilisateur intuitives et assurer une communication efficace entre les différents composants de l'application.

13 Lien github

Le TP est disponible sur GitHub : Programmation-Mobile Repository