Projet PMR : Smart Shopping

Bastien Poberay, Jean Geyer, Léo Rongieres, Marien Rossi

SOMMAIRE

Description du projet

Introduction

Dans un monde où la réalité augmentée connaît un essor fulgurant, de nouvelles opportunités émergent pour l'amélioration de nos expériences quotidiennes. Ce rapport présente un projet visant à exploiter les avantages de la réalité augmentée pour faciliter les courses des utilisateurs à travers le développement d'une application mobile innovante, accompagnée de lunettes de réalité augmentée.

Contexte

De nos jours, les courses peuvent souvent être une tâche fastidieuse et chronophage, avec des clients se trouvant dans des grands magasins, à la recherche d'articles spécifiques parmi une multitude de rayons. La réalité augmentée offre une solution prometteuse en combinant des informations visuelles en temps réel avec l'environnement physique, ce qui permet d'améliorer l'efficacité et l'expérience des utilisateurs lors de leurs courses.

Description du projet

Ainsi, notre projet consiste à développer une application mobile spécialement conçue pour les courses, en utilisant des lunettes de réalité augmentée pour guider les utilisateurs dans les magasins et simplifier leur processus d'achat. L'application permettra aux utilisateurs de saisir leur liste de courses via une interface conviviale, puis les lunettes de réalité augmentée les guideront pas à pas pour trouver les articles nécessaires de la manière la plus efficace possible.

Fonctionnalités clés

Les principales fonctionnalités de l'application comprennent :

- Saisie facile de la liste des courses par les utilisateurs via l'application mobile.
- Affichage des articles à acheter directement sur les lunettes de réalité augmentée, offrant une visualisation claire et pratique.
- Guidage visuel en temps réel pour aider les utilisateurs à trouver les articles dans le magasin en suivant le chemin le plus court.
- Utilisation de commandes vocales pour ajouter ou supprimer des articles de la liste de courses, offrant une expérience mains libres et pratique.
- Intégration avec une base de données du magasin pour obtenir des informations sur les emplacements précis des articles, assurant une précision et une actualisation en temps réel.

Avantages et impacts potentiels

L'application mobile combinée aux lunettes de réalité augmentée apporte plusieurs avantages significatifs aux utilisateurs :

- **Une expérience de course améliorée**, avec une recherche d'articles plus rapide et plus efficace.
- **Une réduction du temps passé dans les magasins**, permettant aux utilisateurs de consacrer plus de temps à d'autres activités.
- **Une navigation simplifiée et intuitive**, réduisant le stress et la confusion lors des courses.
- **Une gestion de la liste de courses plus pratique** grâce aux commandes vocales, offrant une expérience mains libres et sans encombre.

Cahier des charges

Fonctionnalités clés	**Exigences fonctionnelles**	**Exigences non fonctionnelles**
Saisie de la liste des courses par l'utilisateur	Saisie de la liste des courses via une interface intuitive	Performance : L'application et les lunettes de réalité augmentée réagiront rapidement aux actions de l'utilisateur pour assurer une expérience fluide et sans latence.
Guidage visuel pour aider l'utilisateur à trouver les articles dans le magasin	Guidage visuel dans le magasin : Les lunettes de réalité augmentée guideront visuellement l'utilisateur vers les emplacements des articles dans le magasin en utilisant le chemin le plus court	Convivialité : L'interface utilisateur de l'application et les interactions avec les lunettes de réalité augmentée seront intuitives et faciles à comprendre pour les utilisateurs.
Commandes vocales pour ajouter ou supprimer des articles de la liste de courses	Commandes vocales pour la gestion de la liste de courses : Les lunettes de réalité augmentée permettent à	Compatibilité : L'application sera compatible avec les appareils Android récents

	l'utilisateur d'ajouter vocalement des articles à la liste de courses ou d'en supprimer.	et les lunettes de réalité augmentée spécifiées.
Interaction avec une base de données du magasin pour obtenir les informations sur les emplacements des articles	Interaction avec la base de données du magasin L'application interagit avec une base de données du magasin pour obtenir les informations sur les emplacements des articles	Sécurité : Les informations de la liste de courses de l'utilisateur seront sécurisées et protégées contre tout accès non autorisé. De meme pour la base de donnée du magasin

Contraintes techniques

- L'application sera développée pour la plateforme Android.
- Les lunettes de réalité augmentée spécifiées seront utilisées.
- La base de données du magasin devra être accessible et mise à jour régulièrement.

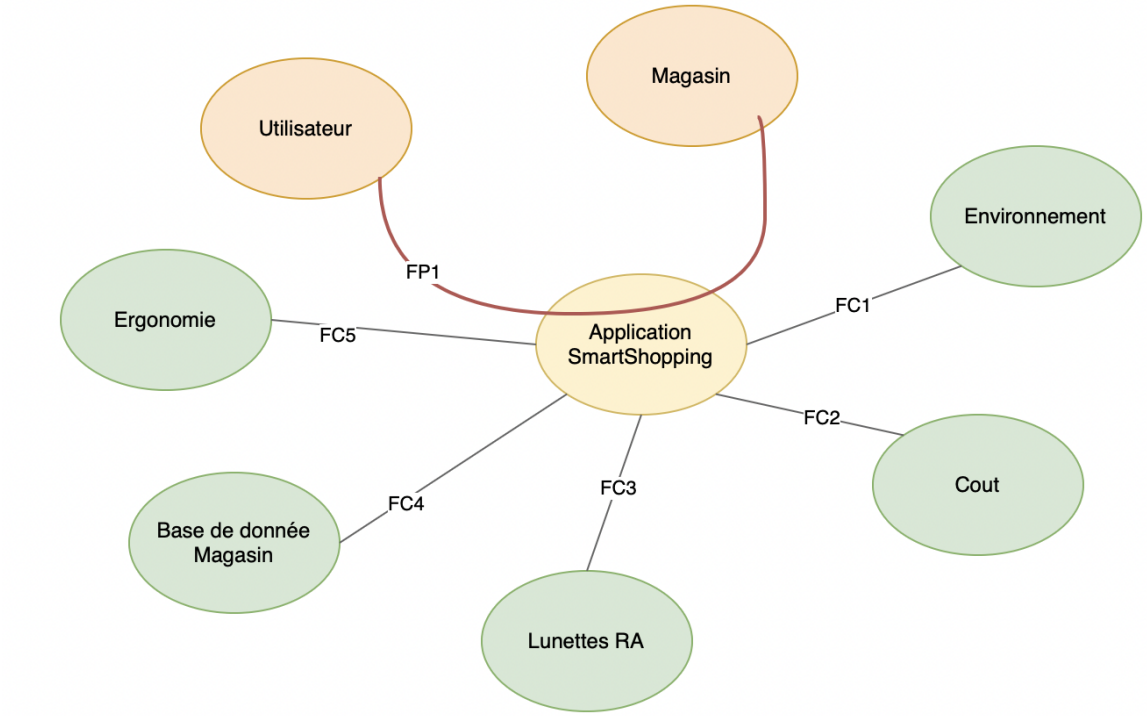
Planning et livrables

- Développement de l'application mobile avec les fonctionnalités spécifiées.
- Intégration des lunettes de réalité augmentée avec l'application.
- Connexion et interaction avec la base de données du magasin.

Validation et acceptation

- Les tests seront effectués pour vérifier que l'application répond aux exigences fonctionnelles et non fonctionnelles spécifiées.
- Une démonstration de l'application en conditions réelles sera réalisée pour valider son bon fonctionnement.

Analyse fonctionnelle



FP1	L'application doit permettre à l'utilisateur de réaliser ses courses rapidement dans un magasin
FC1	L'application doit prendre en compte l'environnement du magasin pour guider l'utilisateur
FC2	L'application doit avoir un coût raisonnable
FC3	L'application doit intégrer l'utilisation de lunettes RA
FC4	L'application doit pouvoir interagir avec la base de donnée du magasin
FC5	L'application doit être intuitive à utiliser

Modélisation SysML

La modélisation SysML du projet permettra de représenter de manière structurée et systématique les différentes composantes et interactions du système. En utilisant les diagrammes SysML tels que le diagramme des cas d'utilisation, le diagramme de blocs internes, le diagramme d'activité et le diagramme de séquence, nous serons en mesure de capturer les exigences, les fonctionnalités et les flux de travail du système. Cette approche de modélisation SysML nous permettra de visualiser et de communiquer efficacement la complexité du projet, d'identifier les liens entre les différentes parties et de faciliter la compréhension globale du système, tout en fournissant une base solide pour la conception, le développement et la validation du projet.

Diagramme des cas d'utilisation

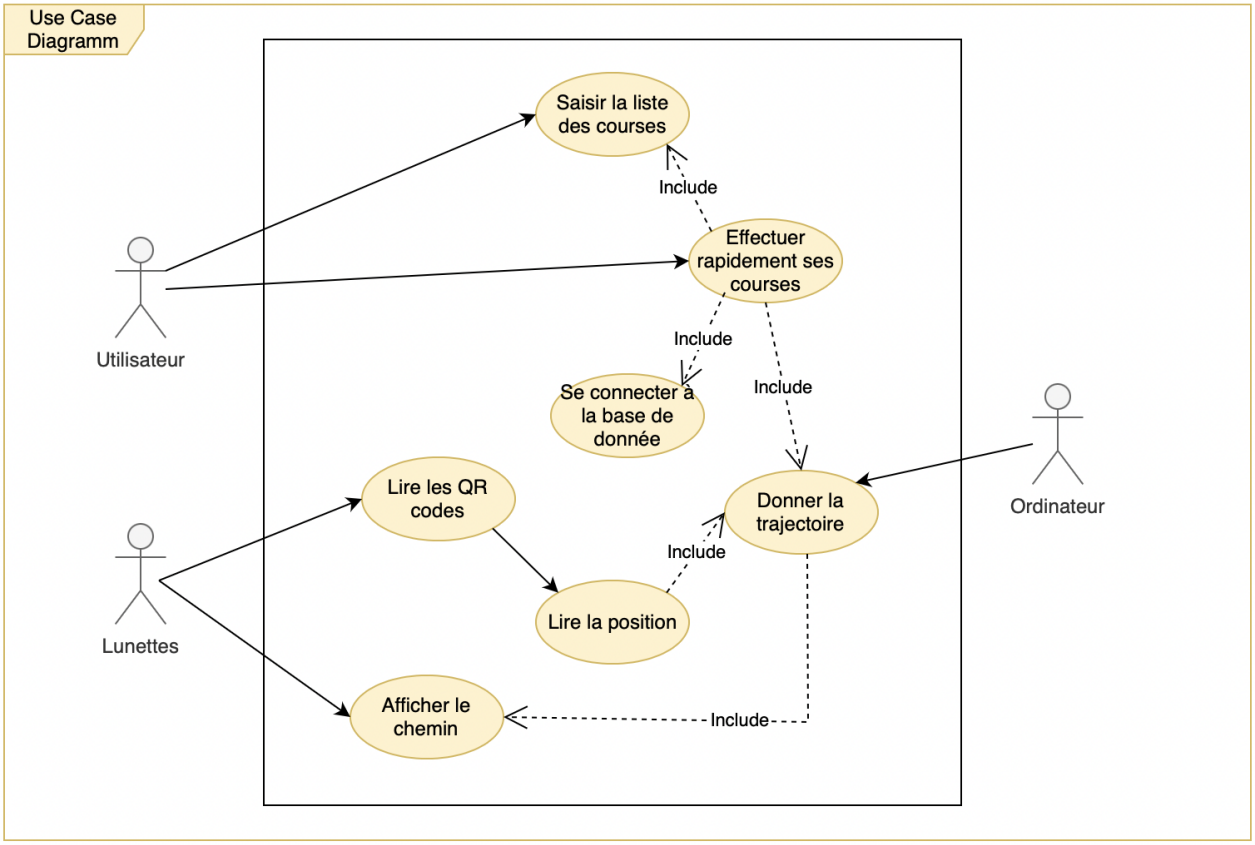


Diagramme de séquence

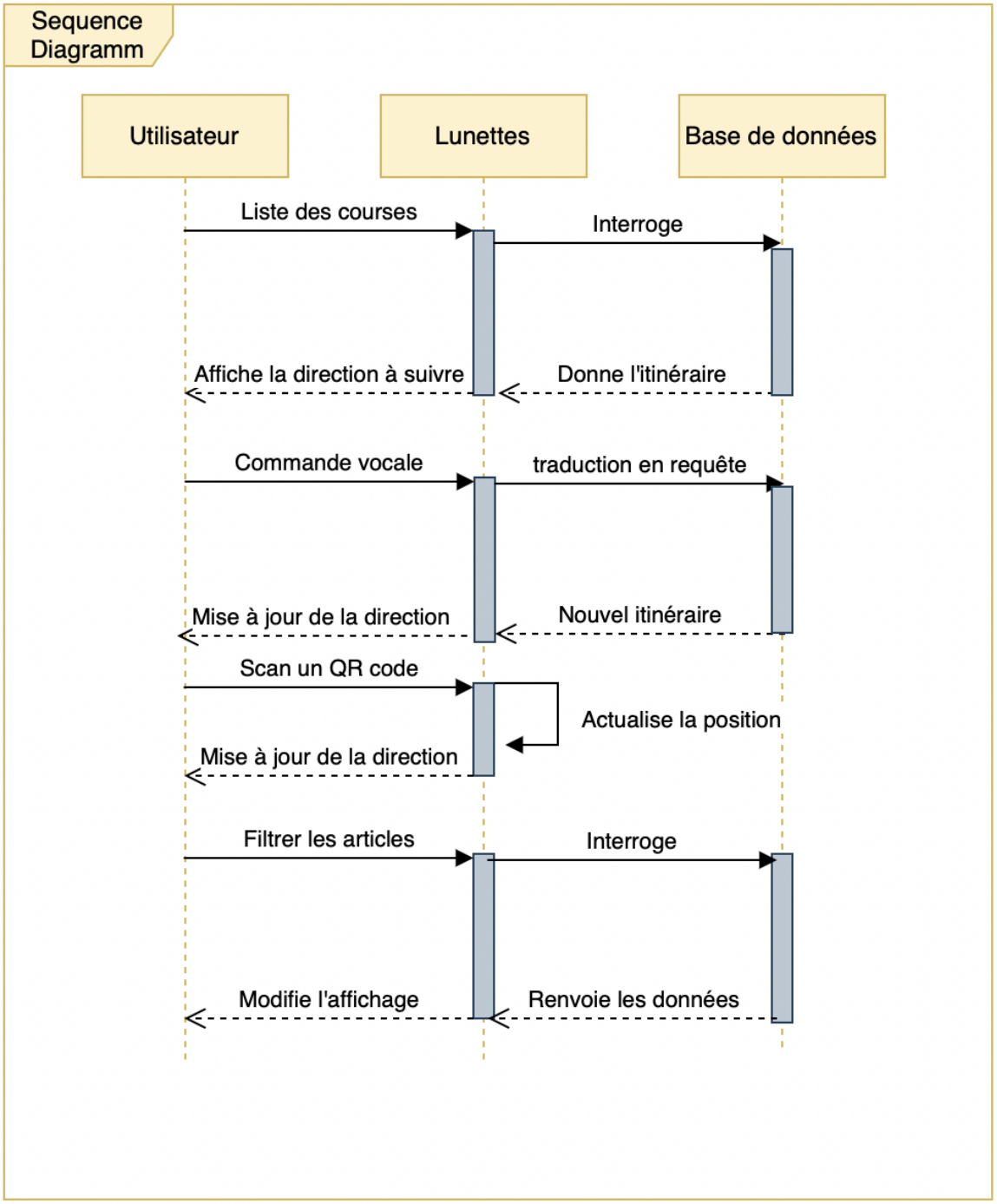


Diagramme des exigences

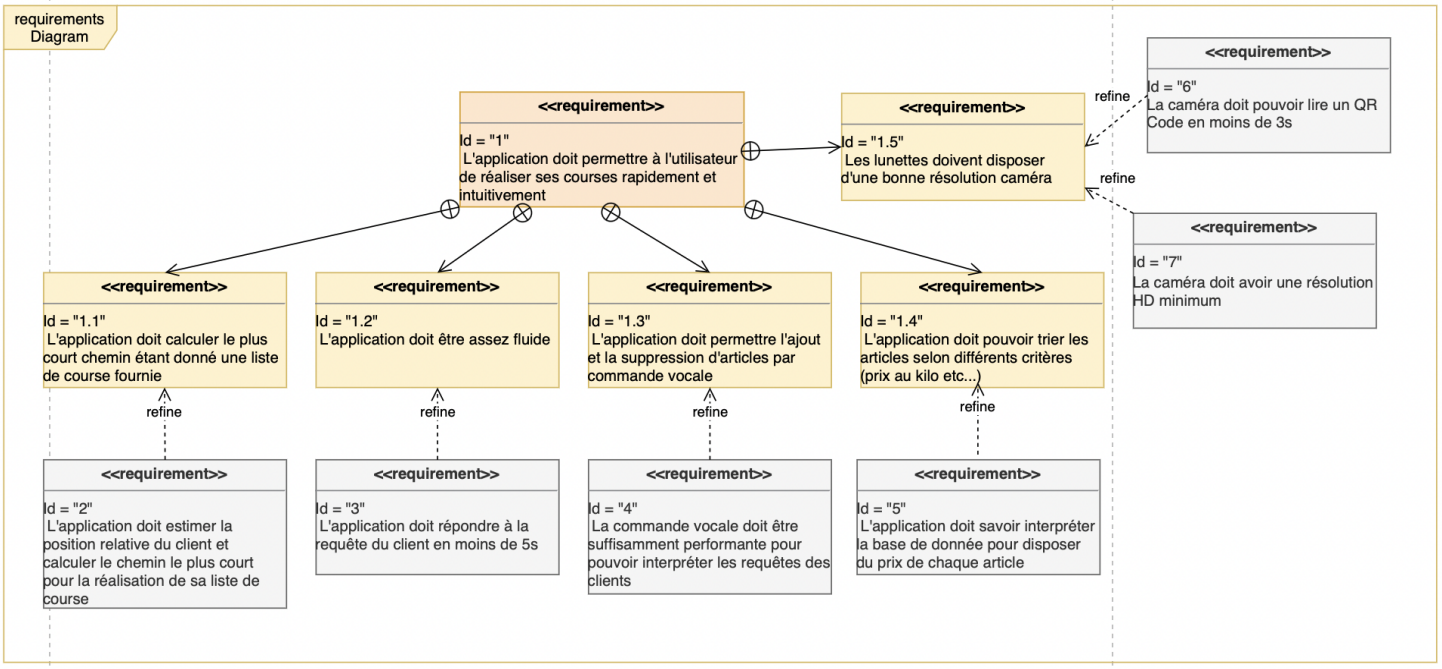
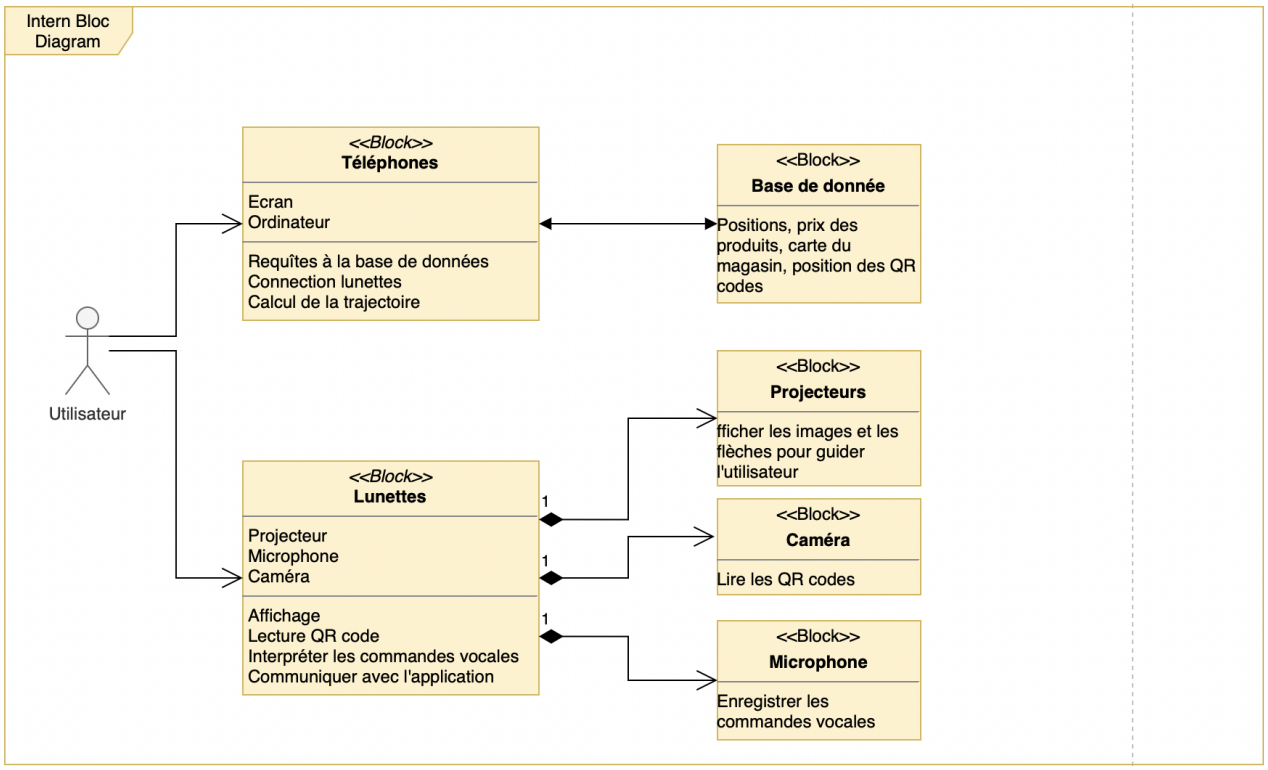


Diagramme de bloc internes



Architecture informatique

Afin de mettre en œuvre la solution, nous avons codé cette application mobile sur l'environnement Android Studio utilisant le langage de programmation Kotlin. L'architecture est composée de différentes activités qui interagissent entre elles afin de parvenir à l'application finale. Nous détaillons par la suite le fonctionnement de chacune de ces activités.

MainActivity

Cette activité permet de gérer l'interface utilisateur pour la saisie des identifiants. Il s'agit de la première page qui s'ouvre au moment du lancement de l'application.

Après avoir saisi ses identifiants, on envoie une requête http de connexion à l'API TODO grâce à la bibliothèque Volley. On récupère la clé retournée par la requête et on la stocke dans les sharedpreferences. On en aura besoin pour manipuler les listes de l'utilisateur (cf documentation API).

Envoi de requête HTTP

```
private fun makeApiRequest(urlAPI: String, user: String, password: String) {
    val requestQueue : RequestQueue! = Volley.newRequestQueue( context: this)
    val url : String = urlAPI +"authenticate?user=$user&password=$password"
    val request = JsonObjectRequest(Request.Method.POST, url, jsonRequest: null,
        { response ->
            // Handle API response here
            if (response.getBoolean( name: "success")) {
                val token : String = response.getString( name: "hash")
                saveToken(token)
                startActivity(Intent(applicationContext, Menu::class.java))
                finish()
            }
        },
        { error ->
            // Handle API request error
            Toast.makeText( context: this@MainActivity, text: "API connection error: ${error.message}", Toast.LENGTH_LONG).show()
        })
    requestQueue.add(request)
}
```

Le code utilise la bibliothèque Volley pour effectuer une requête POST à une API distante. Il envoie le nom d'utilisateur et le mot de passe saisis dans la requête. En cas de succès de la requête, il récupère le jeton d'authentification envoyé par l'API,

le sauvegarde dans les SharedPreferences. Si la connexion est réussi, on ouvre le menu.

Sauvegarde du jeton d'authentification
Le code contient une fonction saveToken() qui enregistre le jeton d'authentification dans les SharedPreferences.

Configuration des écouteurs d'événements
Le code configure les écouteurs d'événements pour les boutons de connexion et de création de compte. Lorsque le bouton de connexion est cliqué, il récupère le nom d'utilisateur et le mot de passe saisis, puis appelle la fonction makeApiRequest() pour effectuer une requête à une API distante. Lorsque le bouton de création de compte est cliqué, il lance une nouvelle activité pour créer un compte utilisateur.

.

CreateAccountActivity

Cette activité permet la création d'un nouveau compte ainsi que la connexion utilisant une API. En saisissant un nom d'utilisateur et un mot de passe, l'utilisateur est en mesure de créer un nouveau compte et de pouvoir s'y connecter par la suite.

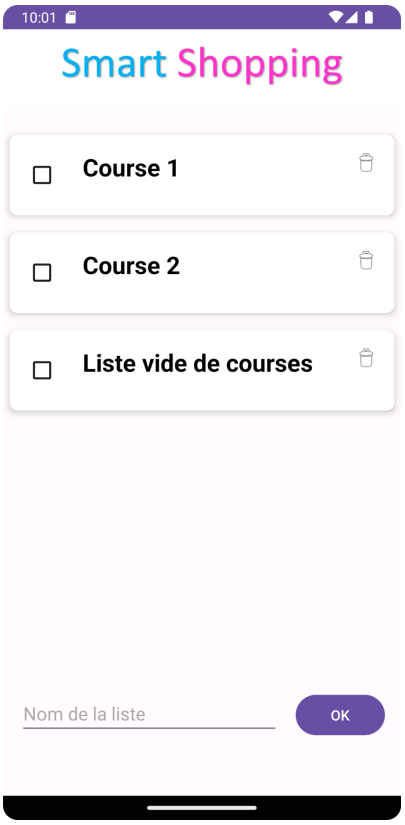
Requête de création de compte

```
private fun createAccount(urlAPI: String, username: String, password: String) {
    val requestQueue : RequestQueue! = Volley.newRequestQueue( context: this)

    val url : String = "$urlAPI" + "users?pseudo=$username&pass=$password"

    val request = object : JsonObjectRequest(
        Request.Method.POST, url, jsonRequest: null,
        Response.Listener {...},
        Response.ErrorListener {...}) {
        override fun getHeaders(): MutableMap<String, String> {
            val headers = HashMap<String, String>()
            val token : String = getToken()
            headers["hash"] = token
            return headers
        }
    }
    requestQueue.add(request)
}
```

Pour créer le compte, on a besoin d'ajouter un header contenant le token sauvegardé dans les sharedpreferences. On doit ajouter la méthode getHeaders() de la bibliothèque volley pour associer un header à notre requête HTTP. Si la création réussit, on fait comme précédemment on ouvre le menu et on enregistre le nouveau token.

Menu

Une fois le menu lancé, l'application va interroger la base de données pour récupérer le nom et l'id de toutes les listes de l'utilisateur connecté et l'afficher à l'écran. Une fois cela fait, on eut ajouter des nouvelles listes en utilisant l'EditText en bas de l'écran et en appuyant sur OK. On peut également en supprimer en ayant sur la petite corbeille.
En cliquant sur le nom d'une liste, cela ouvrira l'activité SowListActivity qui affichera les articles dans la liste cliquée.



Les requêtes HTTP

Pour réaliser les différentes modification des listes, on a utilisé les mêmes fonctions que celle de la page de connexion en changeant l'URL et le type de requête (GET/POST/DELETE):

Pour afficher les listes (GET) :

```
val url = urlAPI+"lists"
```

Pour ajouter une liste (POST) :

```
val url = urlAPI+"lists?label=$label"
```

Pour supprimer une liste (DELETE) :

```
val url = "$urlAPI/lists/${itemList.id}"
```

Récupérer les listes des requêtes

La fonction utilisée JsonObjectRequest() renvoie void. Or, on veut récupérer les listes retournées par l'API. On utilise donc une data class:

```
data class ListData(var itemLists: MutableList<ItemList>) {
    Gueck
    data class ItemList(val id: String, val name: String)
}
```

Elle permet de récupérer les attributs de chaque liste.

Affichage des listes

Pour afficher les listes, on a utilisé un Recycler view, qui permet d'afficher un nombre variable d'éléments:

```
inner class ItemListAdapter(private val itemList: MutableList<ListItem>) :
    RecyclerView.Adapter<ItemListAdapter.ItemViewHolder>() {
    Gueck
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {...}
    Gueck
    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {...}
    Gueck
    override fun getItemCount(): Int {...}
    Gueck
    inner class ItemViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView), View.OnClickListener {...}

}
```

En plus du layout du layout de l'activité, on a créé un autre layout qui va être appelé par le recycler view. Ce layout correspond à la case contenant une case cochable, le nom de la liste et le petit logo poubelle. On fait appel à ce layout dans ItemViewHolder :

```
inner class ItemListViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView),
    View.OnClickListener {

    private val textView: TextView = itemView.findViewById(R.id.textView)
    private val imageViewDelete: ImageView = itemView.findViewById(R.id.imageViewDeleteItem)

    Gueck
    init {
        itemView.setOnClickListener(this)
        imageViewDelete.setOnClickListener(this)
    }

    Gueck
    fun bind(itemList: ListData.ItemList) {
        textView.text = itemList.name
    }

    Gueck
    override fun onClick(view: View) {
        when (view) {
            itemView -> {
                val itemList : ListData.ItemList = itemLists[adapterPosition]
                val intent = Intent( packageContext: this@Menu, ShowListActivity::class.java)
                intent.putExtra( name: "id", itemList.id)
                startActivity(intent)
            }
            imageViewDelete -> {
                val itemList : ListData.ItemList = itemLists[adapterPosition]
                delList(itemList)
            }
        }
    }
}
```

On y définit nos boutons dans chaque case, avec les actions qu'ils exécutent s' ils sont activés.

Si on ajoute ou supprime un élément de la liste, on doit utiliser la méthode notifyDataSetChanged() qui recharge l'affichage avec la liste modifiée.
Si on clique sur le nom d'un liste, l'application lance l'activité ShowList avec l'id de la liste en Intent (nécessaire pour la requête)

Show ListActivity

Cette activité permet à l'utilisateur de saisir sa liste de course en ayant la possibilité de rajouter et de supprimer des articles.
La partie affichage, suppression et ajout d'item est la même que pour la classe Menu, on ne va donc pas la détailler.

Lorsque l'utilisateur appui sur commencer la navigation, elle lis les coordonnées des articles de la liste et lance l'activité Itinéraire

Les requêtes HTTP utilisées
Pour les mêmes raisons que précédemment, on a créé une data class pour les articles reprennant ses atributs dans la base de données :

```
data class ListItem(val id: String, val label: String, val url: String?, val checked: String)
```

Pour afficher les articles (GET) :

```
val url = "$urlAPI/lists/$id/items"
```

Pour ajouter un article (POST) :

```
val url = "$urlAPI/lists/$id/items?label=$label"
```

Pour supprimer un article (DELETE) :

```
val url = "$urlAPI/lists/$id/items/${item.id}"
```

Retrouver les coordonnées des articles
Ne sachant pas faire des API, on préféré stocker dans l'application les coordonnées des produits du magasin dans un json de la forme :

```
{
  "ligne": 2,
  "colonne": 2,
  "nom_produit": "Oranges"
},
```

Maintenant qu'on a le nom des articles de la liste, on va lire le json contenant les coordonnées de tous les articles dans le magasin :

```
val cartographyJson : String = applicationContext.resources.openRawResource(R.raw.product).bufferedReader().use { it: BufferedReader
    it.readText() }
val cartographyArray = JSONArray(cartographyJson)
```

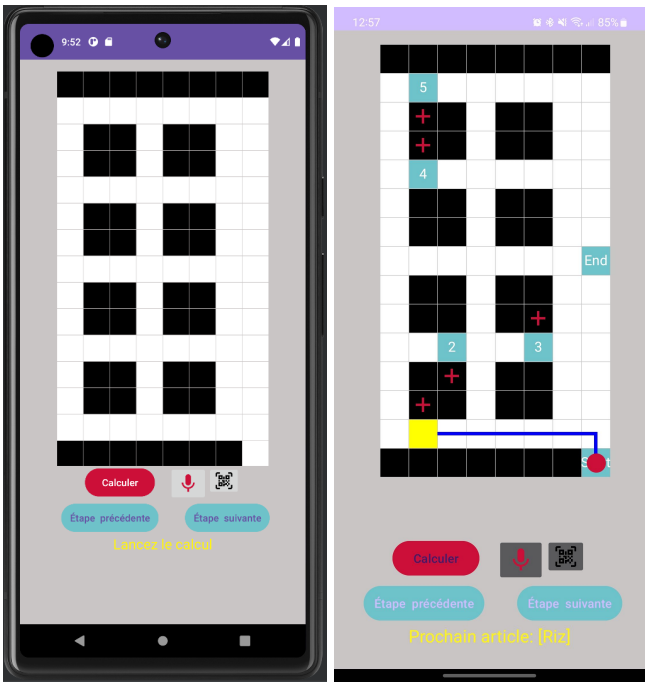
On parcourt ensuite l'enemble des éléments et on récupère les coordonnées lorsqu'on a une correspondance avec les noms.
On stocke les points de passage dans une liste :

```
coordinatesList: MutableList<Point>
```

Qu'on envoie à la classe itinéraire.

Activité Itinéraire

Le code est commenté mais voici une introduction à la compréhension du programme :



Itineraire.kt possède 2 objectifs:

- Calculer l'itinéraire (classe Itineraire)
- Le dessiner à l'écran (classe MapPanel)

Ici lorsqu'on parle d'un point, on considère l'objet de la classe Graphics : Point(x,y), ce qui équivaut à une position sur la carte du magasin.
Suivant la manière dont notre carte de magasin a été définie, les mouvements se font 1 case à la fois, uniquement verticalement ou horizontalement, sur des cases blanches (1 dans la matrice).

Les boutons sont les suivants :

-Calculer : lance l'itinéraire
-Etape suivante : passe à l'étape suivante
-Etape précédente : passe à l'étape précédente
-Micro : lance la reconnaissance vocale
-QRcode : lance le scanner de QRcode

<u>Variables</u>

```
val map = arrayOf(
    intArrayOf(0, 0, 0, 0, 0, 0, 0, 0),
    intArrayOf(1, 1, 1, 1, 1, 1, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 1, 1, 1, 1, 1, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 1, 1, 1, 1, 1, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 1, 1, 1, 1, 1, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 0, 0, 1, 0, 0, 1, 1),
    intArrayOf(1, 1, 1, 1, 1, 1, 1, 1),
    intArrayOf(0, 0, 0, 0, 0, 0, 0, 1),
)
```

-**coordinatesList** : La liste des points des articles de l'activité précédente, les coordonnées x et y étant inversées, on utilise la fonction *inverseCoordinates()* ce qui nous donne la nouvelle variable **invertedCoordinatesList**.
-**articleNamesList** : La liste des noms des articles en question.
-**relationTable** : Table de relation entre une point et le nom de l'article qui s'y situe.
-checkPoints: Liste des points chemin par lesquels il faut passer pour ramasser tous les articles.
-**optimalOrder** : La liste **checkPoints** triée pour minimiser la distance totale.

-**optimalPath** : La liste des listes de points qui relient de manière optimale les points de la liste **optimalOrder** 2 à 2. Pour chaque point de **optimalOrder** (excepté le dernier), **optimalPath** contient le trajet qui permet à partir de ce point d'atteindre le point suivant.
-**currentStep** : Indice de l'étape en cours.
-**currentPath** : Liste des points décrivant l'étape en cours (=optimalPath[currentStep])
-**currentArticle** : nom de l'article auquel mène l'étape en cours.

Fonctions

Opérations

```
fun inverseCoordinates(points: List<Point>): List<Point>
```

Inverse les coordonnées x et y de l'argument, et retourne le nouveau point.

```
private fun generateRandomPoints(count: Int, map:
Array<IntArray>): List<Point>
```

Génère **count** points aléatoire sur des cases noires pour simuler des articles, retourne la liste de ces points.

```
private fun checkpoints(points: List<Point>): List<Point>
```

Associe à chaque point de la liste **points** une case blanche adjacente (au dessus ou en dessous), retourne la liste des coordonnées des cases blanches.

```
private fun sortCheckpoints(checkpoints: List<Point>):
Pair<List<Point>, List<List<Point>>>
```

Trie la liste **checkPoints** afin de minimiser la distance totale, retourne un couple contenant la liste ordonnée, et la liste des chemins de celle-ci. (optimalOrder, optimalPath)

```
private fun generatePermutations(points: MutableList<Point>,
size: Int, permutations: MutableList<List<Point>>)
```

Génère les permutations (30 000 actuellement) pour trouver le plus court chemin passant par tous les points. Basé sur Heap.

```
private fun calculateTotalDistance(startPoint: Point,
endPoint: Point, points: List<Point>): Pair<Float,
List<List<Point>>>
```

Somme les distances entre l'entrée et le premier point de la liste **points**, entre les points de la liste **points**, entre le dernier point de la liste **points** et la sortie. Cette fonction est utilisée pour chaque permutation, et on retourne un couple contenant la longueur totale et la liste des chemins reliant les points entre eux.

```
private fun calculateDistance(matrix: Array<IntArray>, p1: Point, p2: Point): Pair<Int, List<Point>>
```
Algorithme de parcours en largeur BFS, pour estimer la distance (en nombre de cases) entre **p1** et **p2**, on retourne la longueur du chemin le plus court et le chemin le plus court entre ces 2 points. Cette fonction est utilisée pour calculer chaque distance séparant 2 points.

Dessin

```
fun setMap(map: Array<IntArray>)
```
Permet de dessiner la carte à partir d'une matrice de 0 et de 1.

```
fun setPoints(randomPoints: List<Point>, orderedCheckpoints: List<Point>, currentPath: List<Point>)
```
Permet de dessiner une croix rouge pour les cases articles (**randomPoints**), colorier en bleu les cases des points de passage (**orderedCheckPoints**) et écrire dessus l'ordre de passage, dessiner la première étape de l'itinéraire (**currentPath**).

```
fun setPath(currentPath: List<Point>)
```
Dessine une nouvelle étape de l'itinéraire.

```
fun setHiddenCheckpoints(hiddenCheckpoints: List<Point>)
```
Efface les points déjà parcourus.

CartographyDatabaseHelper

//

Activité NavActivity

Cette activité se lance depuis l'activité *Itineraire* avec le bouton "QR code" et reçoit deux variables :

- optimalPath : liste qui contient les listes de chaque étape de l'itinéraire, qui elles-mêmes contiennent tous les points de passage de leur étape
- currentStep : entier qui définit l'étape de l'itinéraire actuelle

Cette activité permet de scanner un QR Code afin de connaître la position de l'utilisateur dans le magasin et d'afficher une flèche directionnelle de guidage en fonction de la direction à suivre.

Le scanner de QR Code utilise les bibliothèques *MLKit* et *Camera* et l'affichage de la scène de réalité augmentée est gérée avec *Sceneview*.

La fonctionnalité de scan est donc lancée en utilisant le bouton *scanButton*.

```
scanButton = findViewById<Button>(R.id.scanButton)
scanButton.setOnClickListener { it: View!
    if (allPermissionsGranted()) {
        startCameraWithDelay()
    } else {
        ActivityCompat.requestPermissions(
            activity: this, arrayOf(Manifest.permission.CAMERA), REQUEST_CODE_PERMISSIONS
        )
    }
}
```

On vérifie que les permissions nécessaires (la caméra du téléphone) sont bien données avant de lancer la caméra.

La fonction *startCameraWithDelay()* permet simplement d'arrêter la caméra une seconde après son démarrage. En effet, la superposition de l'affichage de la scène de réalité augmentée et du scan de QR Code entraîne des importants ralentissements de l'application et il est donc nécessaire d'arrêter la caméra après utilisation.
L'arrêt de la caméra est géré par la fonction *stopCamera* qui réinitialise les différentes instances utilisées pour l'analyse des images scannées.

La fonction principale de cette activité est *startCamera()* qui, comme son nom l'indique, démarre la caméra et gère les actions à faire (traitement des QR Code principalement).
La partie de récupération et du traitement de l'image est directement faite par les bibliothèques décrites plus haut et notamment avec la classe *BarcodeAnalyzer*. On récupère alors le contenu de QRCode scanné.

```
val scannedCoords = barcode.split( ...delimiters: "_")  //Sépare les deux coordonnées (QR code de la forme x_y)
val x = scannedCoords[0]
Log.d( tag: "xcoord", x)  //Debug
Log.d( tag: "xreel", x.toInt().toString())
val y = scannedCoords[1]
val currentPoint = Point(x.toInt(), y.toInt())
Log.d( tag: "currentScan", currentPoint.toString())
Log.d( tag: "currentPath", navigation[currentStep].toString())
val isInPath: Boolean = currentPoint in navigation[currentStep]
Log.d( tag: "isInPath", isInPath.toString())

if (navigation[currentStep] != null) {
    if (currentPoint in navigation[currentStep]) {
        placeModel(currentPoint)
    } else {
        Toast.makeText( context: this,  text: "Vous êtes perdus", Toast.LENGTH_SHORT).show()
    }
}
```

Pour rappel, les QR Code permettent de connaître la position de l'utilisateur. Nous avons donc choisi d'intégrer directement les coordonnées du point où se trouve l'utilisateur dans le texte du QR Code, qui renvoie un string de la forme *"x_y"*.

Le contenu récupéré est donc séparé en deux varibales x et y avec la méthode *split("_")*. On définit ensuite le point actuel, currentPoint, avec x et y.
Esuite, on va vérifirer si le point scanné est bien présent dans l'itinéraire en cours (navigation[currentStep]). Si c'est le cas, on appelle la fonction *placeModel(currentPoint)*. Sinon, on prévient l'utilisateur qu'il s'est égaré et est sorti de l'itinéraire.

La deuxième fonction très imoportante est *placeModel(currentPoint)* qui charge un modèle 3D de flèche et l'affiche en fonction de la direction à suivre. Cette fonction récupère comme argument le point actuel scanné.
Dans cette fonction, on récupère l'itinéraire en cours avec une méthode *getSerializableExtra()* et l'indice de cet itinéraire avec *getIntExtra()*. On peut aussi connaître l'indice du point scanné dans l'itinéraire actuel avec la méthode *indexOf(currentPoint)*.

```
val navigation = intent.getSerializableExtra( name: "optimalPath") as List<List<Point>>
val currentStep = intent.getIntExtra( name: "currentStep", defaultValue: 0)
val posIndex: Int = navigation[currentStep].indexOf(currentPoint)
val nextPos: Point = navigation[currentStep][posIndex!! + 1] //Position à venir

if (currentPoint.x < nextPos.x) { //Si la prochaine position est derrière
    selectedRotation = 270f
} else if (currentPoint.x > nextPos.x) { //Si la prochaine position est devant
    selectedRotation = 90f
} else if (currentPoint.y < nextPos.y) { //Si la prochaine position est à gauche
    selectedRotation = 0f
} else { //Si la prochaine position est à droite
    selectedRotation = 180f
}
```

On peut alors comparer le point actuel avec le point suivant dans l'itinéraire et distinguer les cas :

- si x du point actuel < x du point suivant : on définit la rotation de la flèche à 270° (le modèle à 0° pointe vers la droite).

On fait de même avec x actuel > x suivant, y actuel < y suivant et y actuel > y suivant.

On peut alors charger le modèle 3D pour l'afficher dans la scène avec la bibliothèque sceneView :

```
modelNode = ArModelNode(PlacementMode.INSTANT).apply { this:
    loadModelGlbAsync(
        glbFileLocation = "models/direction_arrow.glb",
        scaleToUnits = 1f,
        centerOrigin = Position( v: -0.5f),
    ) { it: ModelInstance
        sceneView.planeRenderer.isVisible = true
    }
}
//Rotation de la flèche
val rotation = Rotation(y = selectedRotation)
modelNode.worldRotation = rotation

sceneView.addChild(modelNode)
scene = true
```

Améliorations possibles :

Il y a des améliorations simples à faire dans cette activité qui permettraient de l'améliorer :

- améliorer la gestion des paramètres récupérés en intent : actuellement ces paramètres sont récupérés dans deux fonctions différentes. Il aurait été judicieux de faire cela dans la fonction *onCreate()* et de passer les paramètres en argument des fonctions *startCamera* et *placeModel*.
- ajouter un QR Code sur les articles que l'on cherche : en les scannant on pourrait alors directement passer à l'étape suivante sans avoir à revenir sur l'activité *Itineraire*.
- gestion de la position : si l'utilisateur n'est pas sur un point de l'itinéraire actuel (si il est "perdu") il faudrait alors démarrer l'activité *Itinéraire* avec en argument le point actuel pour recalculer un itinéraire.
- déclenchement du scan par commande vocale

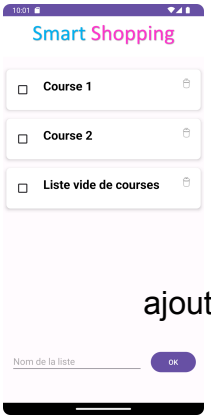
Interaction avec les lunettes de réalité augmentée

L'application a été développée sur smartphone mais peut facilement être intégrée sur des lunettes. En effet, les lunettes de réalité augmentée permettent une navigation en toute autonomie, sans avoir à utiliser ses mains. Il faudrait pour cela développer encore plus la commande vocale.

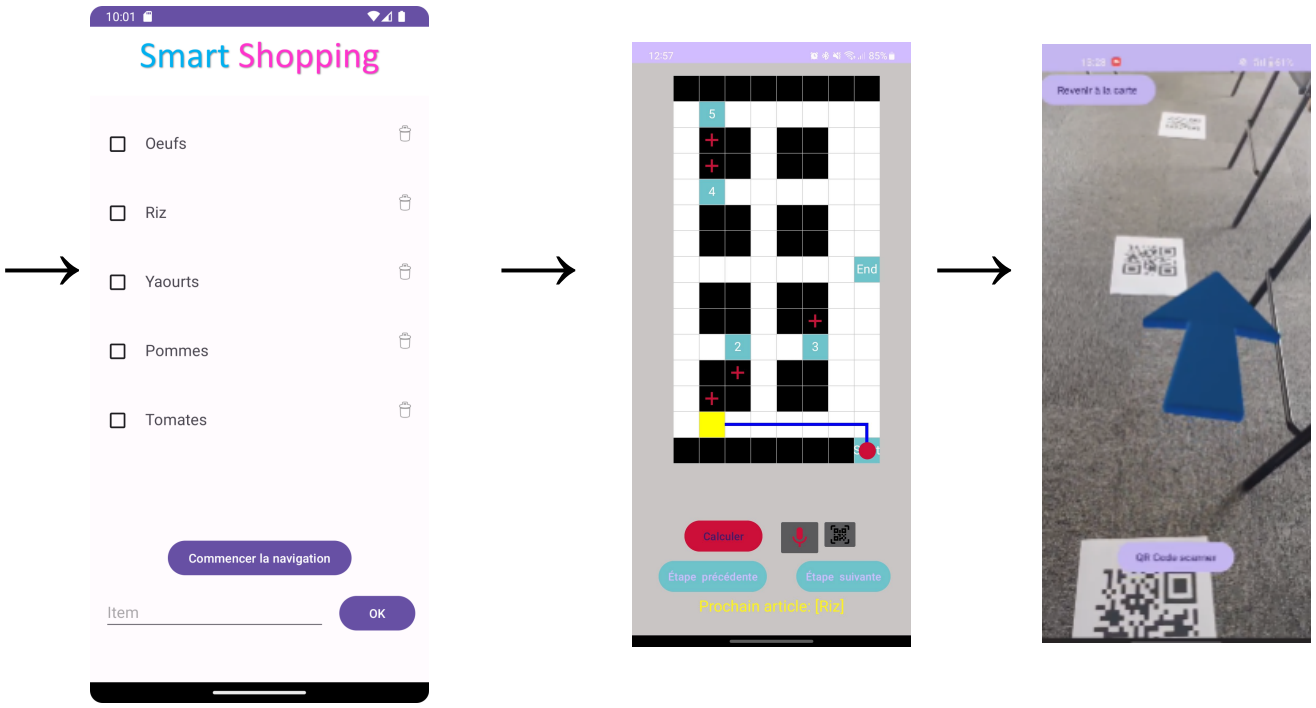
Exemple de situation

Connexion

Créer un compte



Choisir / ajouter / supprimer liste



Ajouter / supprimer article | Calculer itinéraire | Navigation

Difficultés rencontrées

Itineraire.kt

-Trop de permutations dès qu'on dépasse 10 points, trouver une manière d'optimiser le programme pour résoudre le voyageur de commerce avec la spécificité des déplacements sur cases blanches. (On ne peut pas bêtement calculer la distance vol d'oiseau entre 2 points, on ne peut se déplacer que d'une case à la fois avec des conditions ce qui alourdit considérablement les calculs)

-Rendre la matrice dynamique et juste actualiser son état lors des changements d'étape par exemple, plutôt que de tout redessiner à chaque fois.

NavActivity

-Gestion de la scène de réalité augmentée et du scan de QR code en simultané compliquée. Ces fonctionnalités sont gérés par des bibliothèques externes sur lesquels nous sommes autodidactes et il est donc difficile de s'approprier les nombreuses fonctionnalités (parfois certaines fonctionnalités existantes permettent sûrement d'alléger/améliorer le code) ou de trouver les bugs/problèmes.

-J'ai aussi rencontré beaucoup de problèmes de compilation, de conflits de versions entre les différentes bibliothèques utilisées. C'est pourquoi les dernières versions ne sont pas forcément utilisées.

Conclusion et perspectives

En conclusion, ce projet d'application mobile avec des lunettes de réalité augmentée pour faciliter les courses des utilisateurs représente une solution innovante et prometteuse dans un monde en plein essor de la réalité augmentée. En combinant les avantages de la technologie de réalité augmentée avec une interface conviviale, cette application offre une expérience de course améliorée, plus efficace et agréable pour les utilisateurs. Les fonctionnalités telles que la saisie intuitive de la liste des courses, le guidage visuel dans le magasin, les commandes vocales et l'interaction avec la base de données du magasin apportent une valeur ajoutée significative.

Malgré les fonctionnalités intéressantes et l'apport notable de ce projet, quelques pistes d'amélioration peuvent être explorées pour optimiser davantage l'expérience utilisateur et la performance du système :

- **Intégration de la reconnaissance d'objets**

En utilisant des techniques de vision par ordinateur avancées, il serait possible d'ajouter la fonctionnalité de reconnaissance d'objets pour aider les utilisateurs à localiser rapidement les articles souhaités dans le magasin.

- **Personnalisation de l'interface utilisateur**

Offrir des options de personnalisation de l'interface utilisateur permettrait aux utilisateurs d'adapter l'application en fonction de leurs préférences, de leur style de shopping et de leurs besoins spécifiques.

- **Optimisation des itinéraires**

Améliorer les algorithmes de calcul d'itinéraires pour garantir que les utilisateurs sont guidés de manière optimale. Prendre en compte des dispositions plus complexes du magasin.

- **Intégration avec les systèmes de paiement**

Permettre aux utilisateurs de réaliser des paiements directement via l'application, en intégrant des systèmes de paiement sécurisés, offrirait une expérience plus fluide et pratique.

- **Élargissement de la compatibilité matérielle**

En prenant en compte la diversité des lunettes de réalité augmentée disponibles sur le marché, élargir la compatibilité matérielle de l'application permettrait à un plus grand nombre d'utilisateurs de bénéficier de cette solution.

En poursuivant ces pistes d'amélioration, ce projet pourrait offrir une expérience de course encore plus intuitive, rapide et personnalisée, répondant ainsi aux besoins et aux attentes croissants des utilisateurs dans un monde en constante évolution.