Team-42 :
Cédric Bekkar
Jérémy Rodrigues
Julien Tay

Projet PokéClicker

Sommaire du rapport :

Team-42 :
Cédric Bekkar
Jérémy Rodrigues
Julien Tay

I. Recherche du projet

Nous étions tous les trois très intéressés par le fait de réaliser un jeu et non pas seulement une application lambda.

Afin de trouver le thème de celui-ci, nous avons décidé de mettre en commun nos jeux préférés et de choisir pour thème celui qui apparaîtrait pour chacun d'entre nous.

Nous avons longuement hésité entre deux thèmes ; celui de Final Fantasy et celui de Pokémon. Après mure réflexion, il a été décidé de s'orienter vers Pokémon qui est un thème ayant de nombreux avantages :

- Beaucoup de Pokémon accessibles
- Des contenus très variés disponible sur Internet
- Des concepts existants et inspirants

Nous avons jugé important de choisir un thème que nous apprécions tous afin que cela nous donne une motivation supplémentaire lors du développement.

Avoir un thème est une bonne chose mais une application sans fonctionnalité n'a aucune utilité.

Nous avons donc cherché des fonctionnalités que nous pourrions implémenter sans trop de difficultés. C'est là que nous est venu l'idée de faire un jeu de type Clicker.

Un jeu de type Clicker est un jeu où l'action principale est de taper sur l'écran le plus possible afin de réaliser des actions spécifiques.

Les jeux Pokémon sont toujours dirigés vers de multiples combats entre dresseurs de Pokémon ou entre Pokémon eux-mêmes comme dans la saga « Donjon Mystère ».

Nous avons donc réalisé un jeu Pokémon de type Clicker où le joueur a pour but de monter de niveau ces Pokémon en mettant KO le plus d'ennemis.

Team-42 :
Cédric Bekkar
Jérémy Rodrigues
Julien Tay

II. Organisation du projet

1. Répartition du travail

Le travail a été séparé en plusieurs partie :

La partie développement a été principalement réalisée par Jérémy RODRIGUES. Cédric BEKKAR et Julien TAY ont participé au code notamment afin de réaliser certains fragments.

Les parties Game Design et infographie ont été principalement réalisées par Cédric BEKKAR. Jérémy RODRIGUES et Julien TAY ont aidé sur le concept et la sélection des assets.

Les parties technique et testing ont été principalement réalisées par Julien TAY. Cédric BEKKAR et Jérémy RODRIGUES ont participé dans cette partie en notifiant les erreurs sur les différents documents.

2. Planning des tâches

Ce projet doit être réalisé en 23 jours (du 14/05 au 05/06). Ce temps nous a permis de décomposer les tâches plus facilement.

Jour 1 à 2 : Préparation des schémas, de l'architecture de l'appli …
Jour 3 à 8 : Création des fragments, premier contenus et tests navigation
Jour 9 à 15 : Relier la BDD avec l'appli pour des interaction plus simples
Jour 16 à 20 : Création de l'interface graphique et intégration finale des sprites
Jour 21 à 23 : Relié Back et Front pour rendre l'application fonctionnel + tests

Team-42 :
Cédric Bekkar
Jérémy Rodrigues
Julien Tay

III. Composition de l'application

1. La page d'authentification

Les utilisateurs ne peuvent se connecter au jeu qu'avec un compte Google.

Lors de leur première connexion, les joueurs s'inscrivent et sont renvoyés vers la page de demande de connexion de Google. Une fois la connexion validée, le compte est ajouté aux utilisateurs de la Firebase et une liste de Pokémon par défaut leur est distribuée.

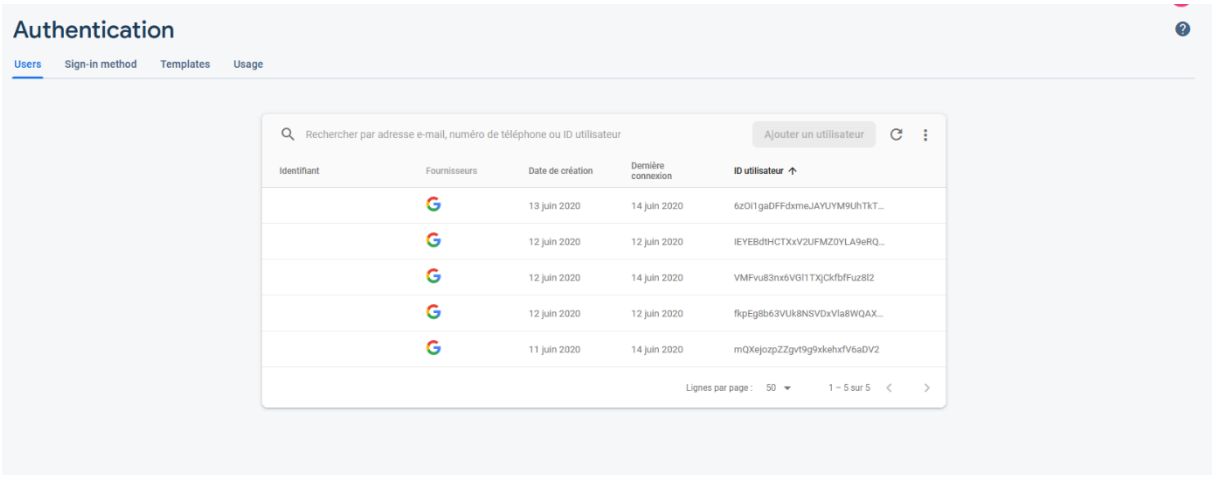
Pour les joueurs ayant déjà un compte, ils ont juste à cliquer sur « Se connecter » afin de reprendre leur partie en cours.

Si un joueur clique sur « s'inscrire avec Google » et qu'il a déjà un compte, il y sera connecté.

2. *La Base de données : Firebase*

Firebase est le nom d'une plateforme mobile de Google qui facilite la création d'un back-end à la fois scalable et performant. En d'autres termes, il s'agit d'une plateforme qui permet de développer rapidement des applications pour mobile et pour le web.

Dans notre projet, Firebase nous permet de créer et stocker de nouveaux utilisateurs grâce à son module d'authentification et donc de gérer leur liste de ressources.



Dans notre cas, nous utilisons l'authentification par Google mais il y a une multitude d'options autre que celle-ci proposées par Firebase.

Firebase implémente aussi un module de base de données « Database » où l'on peut stocker toutes les données que l'on souhaite. (String, Int, Map, List, Any).

Le choix d'utiliser Firebase a été fait pour des question de difficultés du transfert de certaines données et pour la sécurité apportée par Firebase (l'utilisateur ne peut pas trafiquer ses ressources).

```
val currentuser = FirebaseAuth.getInstance().currentUser

//récupère l'uid de gmail du joueur
if (currentuser != null) {
    currentuser.let { it: FirebaseUser
        for (profile in it.providerData) {
            val uid = profile.uid
            Log.wtf( tag: "Login",  msg: "CURRENT USER NOT NULL")
            println(uid)

            val ref = FirebaseDatabase.getInstance().getReference( path: "users").child( pathString: "user$uid").child( pathString: "activePKM")
            ref.setValue(listPokemonUserDefault.getPokemonbyName( nom: "Pikachu"))
            val ref2 = FirebaseDatabase.getInstance().getReference( path: "users").child( pathString: "user$uid").child( pathString: "listPKM")
            ref2.setValue(listPokemonUserDefault)

            break //cette boucle for retourne deux users mais avec un break il n'y en a qu'un seul.
        }
        val intent = Intent( packageContext: this@Login, MainActivity::class.java)
        startActivityForResult(intent, REQUEST_QCM)
    }

}
```

Voici ci-dessus un exemple d'utilisation de Firebase Database. Dans ce cas il s'agit de l'inscription sur la page de login.

On crée une instance de l'utilisateur actuel puis on vérifie s'il est 'null' ou non.

S'il y a bien un utilisateur, le programme va récupérer son UID propre et va créer dans la base de données un nouvel user qui va comporter un Pokémon actif par défaut, ici Pikachu, ainsi qu'une collection de Pokémon par défaut composée de 4 Pokémon (Pikachu, Salamèche, Bulbizarre et Carapuce).

Une fois tous cela réalisé, le programme lance le jeu.

3. *Les divers Constructeurs implémentés*

Nous avons plusieurs constructeurs qui nous permettent de créer les divers objets dont nous avons besoin afin de bien faire fonctionner le jeu.

- Le constructeur Pokémon nous permet comme son nom l'indique de créer un nouveau Pokémon comprenant ; son nom, une image front et une image back, son xp, son niveau, ses dégâts, son évolution et son niveau d'évolution.

Ce constructeur est primordial pour ce jeu car il est utilisé dans quasiment toutes les activités et permet donc de récupérer les images qui seront à afficher par exemple.

- Le constructeur PokemonEnnemy permet de créer les Pokémon que devra combattre le joueur. Il est composé de l'image Front et Back, leur nom, leur Points de vie et l'xp qu'ils donneront lorsqu'ils seront tués.

- Le constructeur Pokédex n'est pas utilisé pour le moment mais le sera très prochainement. Celui-ci va permettre de stocker tous les pokémons ainsi que leurs évolutions.

- Le constructeur PokemonCollection est essentiel au fonctionnement du jeu. Il permet de créer une Arraylist de Pokémon. Cette Arraylist est stockée dans la base de données et répertorie tous les pokémons dont dispose un joueur.

4. Le Fragment « Home »

Lorsque le joueur arrive sur le fragment « Home » qui correspond au fragment où le joueur va combattre, Le programme défini toujours le premier Pokémon ennemi comme étant Ratata. De plus il récupère à l'aide de la base de données, le Pokémon actif du joueur.

Les Pokémon adverses sont désignés par une fonction qui définit aléatoirement le Pokémon adverse. La fonction fait apparaître périodiquement un Pokémon légendaire.

Cette dernière fonction appelle un fonction « setter » qui va afficher le Pokémon adverse ainsi que ses points vie totaux et restants.

```
fun combatGenerator(degat : Int){
    val hpString : String = enemy_hp.text.toString()
    var hpInt = hpString.toInt()
    hpInt = hpInt-degat
    if(hpInt <= 0){
        var indexWaveInt = enemy_index.text.toString().toInt()
        val xpToGive: Int = enemy_xp.text.toString().toInt()
        giveXP(xpToGive)
        indexWaveInt += 1
        if(indexWaveInt == 15){
            val rnds = (1..3).random()
            getRandomLeg(rnds)
            indexWaveInt = 0
        }
        else{
            val rnds = (1..9).random()
            getRandomEnemy(rnds)
        }
        println(indexWaveInt)
        enemy_index.text = indexWaveInt.toString()
    }
    else{
        enemy_pkm_hp.text = hpInt.toString()
        enemy_hp.text = hpInt.toString()
    }


}
```

La fonction ci-dessus permet de gérer les vagues et ainsi savoir quand le joueur doit combattre un Pokémon lambda ou un Pokémon légendaire plus puissant. Cette fonction est gérée par un compteur qui est réinitialisé à chaque fois que le joueur a rencontré un légendaire (toute les 15 vagues).

De plus cette fonction prend en compte les dégâts qu'inflige le Pokémon à chaque clique et le soustrait aux points de vie des Pokémon adverses.

Une fois qu'un ennemi est battu elle se charge aussi de récupérer le montant d'xp à donner et appelle la fonction permettant de distribuer l'xp au Pokémon actif.

5. Le Fragment « Collection »

Fonction présente sur le Fragment collection : Ce fragment permet de consulter les Pokémon que l'on possède, de changer de Pokémon et de regarder les informations détaillées d'un Pokémon. Les slots dans lesquelles les Pokémon apparaissent sont codé de cette manière :

```
val slot01 = root.findViewById<TextView>(R.id.pokemon_slot_1)
```

Pour les remplir nous avons réalisé une boucle for qui parcours les slots et les remplis avec les Pokémon présent dans la BDD. La fonction saveActiveData, permet de garder en mémoire les informations du Pokémon actif, afin d'actualiser le Pokémon lorsque ce dernier monte de niveau.

```
//Définit les stats du pokémons qui ont changées avec le niveau
FirebaseDatabase.getInstance().reference.child( pathString: "users").child( pathString: "user$uid")
    .child( pathString: "listPKM").child( pathString: "listCollec").child( pathString: "$index").child( pathString: "level")
    .setValue(levelOfPKM)
FirebaseDatabase.getInstance().reference.child( pathString: "users").child( pathString: "user$uid")
    .child( pathString: "listPKM").child( pathString: "listCollec").child( pathString: "$index").child( pathString: "degat")
    .setValue(degatOfPKM)
FirebaseDatabase.getInstance().reference.child( pathString: "users").child( pathString: "user$uid")
    .child( pathString: "listPKM").child( pathString: "listCollec").child( pathString: "$index").child( pathString: "xp")
    .setValue(xpOfPKM)
```

Voici la partie permettant de mettre à jour le niveau du Pokémon actif. La fonction « leBoutonQuandOnClique » permet de récupérer le texte sur les slots des Pokémon afin de l'utiliser comme élément de recherche dans la liste des Pokémon.

```
val fragment = PokemonFragment.newInstance(nameOfPKM,levelOfPKM.toInt(),artworkOfPKM,degatOfPKM.toInt(),xpOfPKM.toInt(),index.toString())
activity!!.supportFragmentManager.beginTransaction()
    .replace((view!!.parent as ViewGroup).id, fragment,  tag: "findThisFragment")
    .addToBackStack( name: null)
    .commit()
```

Cette partie de code permet d'envoyer les informations recueillies vers le Pokémon Fragment.

Fonction présente sur PokémonFragment : A l'aide d'un companion object nous allons récupérer les valeurs envoyées depuis la classe

CollectionFragment. Ce fragment va afficher les détails d'un Pokémon, avec les valeurs récupérées depuis le fragment précédent. Une fois les valeurs associées à des variables, nous affichons ces informations :

```
//Affiche le niveau et le nom du pokémon
name.text = pokemonImportedName
level.text = pokemonImportedLevel.toString()

//Affiche l'artwork officiel du pokémon
val artworkImage = pokemonImportedArtwork + "_art"
val imageResource = resources.getIdentifier(artworkImage,  defType: null,  defPackage: "com.example.pkclicker")
val res = resources.getDrawable(imageResource)
artwork.setImageDrawable(res);


//Affiche les dégats et l'xp totale du pokémon
damage.text = pokemonImportedDamage.toString()
xp.text = pokemonImportedXP.toString()
```

Ici on affiche le niveau, le nom, l'artwork, les dégâts et l'expérience du Pokémon. Depuis ce fragment, il est aussi possible de définir le Pokémon actif :

Ici on définit le Pokémon à envoyer, et on modifie les sprites présent sur le fragment Home.

6. Le Fragment « Échange Miracle »

Fonction d'échange miracle dans la partie multijoueur : Cette partie permet au joueur d'échanger un Pokémon dont le niveau est supérieur à 20 avec un autre compris dans la BDD.

Ici nous allons donc devoir modifier un Pokémon ainsi que tous les autres emplacements dans lequel ce dernier se trouve. Pour se faire nous allons devoir récupérer chacune de ses valeurs et ensuite les modifier de cette manière :

```
FirebaseDatabase.getInstance().reference.child( pathString: "echangemiracle").child( pathString: "pkm0$rnds")
    .child( pathString: "nom")
    .setValue(futurName)
```

Ce code permet de récupérer une information et la modifier, nous avons donc fais cela pour les autres informations à modifier.

Une fois le Pokémon et ses informations modifié, nous devons actualiser la collection. En parcourant la liste entière des Pokémon, nous remplaçons donc les informations de l'ancien Pokémon avec le nouveau :

```
FirebaseDatabase.getInstance().reference.child( pathString: "users").child( pathString: "user$uid")
    .child( pathString: "listPKM").child( pathString: "listCollec").child( pathString: "$index").child( pathString: "nom")
    .setValue(newName)
```

Ce code ressemble à celui du dessus, car la plupart des interactions se font via la BDD

Team-42 :
Cédric Bekkar
Jérémy Rodrigues
Julien Tay

IV. Problèmes rencontrés

Voici la liste des problèmes rencontrés :

1. Transfert de données à travers les fragments (Firebase)

   La manière dont nous devions récupérer les valeurs n'était pas clair pour nous. Pour régler ce problème nous avons suivi un tutoriel sur Youtube nous permettant de régler ce problème

2. Circulation inter fragment

Nous n'avons pas réussi à circuler entre les fragments, nous avons donc décider d'utiliser une base de données (Firebase), après cela nous avons dû créer un companion object dans la destination du clic, pour envoyer, depuis la source les infos voulues.

3. Problème d'accès à la Firebase (DataBase)

Il semblait que la Firebase ait des problèmes de connexion sur certains émulateurs, le problème s'est réglé tout seul.

4. Définir une image dans le xml depuis son nom

Pour récupérer une image nous avons dû rajouter quelques lignes de codes :

```
val imageResource = resources.getIdentifier( pkm.getFrontPic(), null, "com.example.pkclicker") val res = resources.getDrawable(imageResource) enemyPKM.setImageDrawable(res);
```

Ces lignes permettant de récupérer le nom du drawable, le définir et ensuite le set