ULB ECOLE POLYTECHNIQUE DE BRUXELLES

INFO-H2001-202021

Projet Informatique BA2

Coder une application avec Kotlin

Rapport

Liam FALLIK

Arié NACAR

Clémence SERMESANT

Professeur : Hugues BERSINI

Année académique 2020-2021

Table des matières

1 Introduction

Dans le cadre du cours d'informatique, il nous a été demandé de créer une application ou un jeu en utilisant le langage de programmation *Kotlin*. Nous avions carte blanche, nous pouvions donc faire parler notre imagination pour concevoir une application révolutionnaire. Nous sommes évalués sur la bonne utilisation de l'orienté-objet et de la gestion d'Android Studio mais pas sur le contenu de notre application, pour peu qu'elle soit originale. Cependant, après avoir réalisé plusieurs petits programmes afin de bien comprendre le fonctionnement de ce langage, dont la création d'un jeu canon, nous avons revu nos objectifs et nous avons commencé à réfléchir à quelque chose de plus réalisable.

Ce projet nous a permis de nous familiariser avec Kotlin et de comprendre les arcanes de la programmation tout en démystifiant la création d'applications qui peut apparaître comme étant un peu magique pour qui n'est pas familier à l'informatique.

Au cours du projet, nous n'avons cessé de découvrir de nouvelles méthodes pour palier aux besoins du jeu que nous voulions créer et une fois celui-ci terminé, nous nous sommes amusés à rajouter des détails d'esthétique ou de fonctionnalité ou encore des options supplémentaires pour l'utilisateur.

Le projet est disponible sur *Github*, accessible au lien suivant `https://github.com/KefakMami/PianoMan`

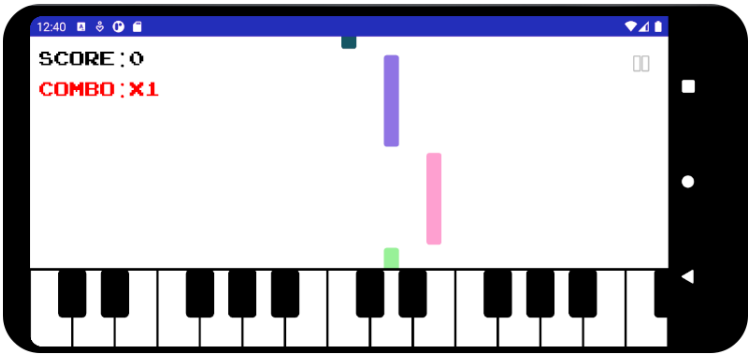
2 Conception

Assez rapidement, la musique a été évoquée pour servir de thème de base à l'application que nous voulions créer car c'est un des intérêts communs des 3 membres du groupe. Nous avions d'abord pensé à s'inspirer grandement du jeu canon en faisant d'un clavier positionné en bas de l'écran un canon à notes qui tirerait sur des ennemis arrivant du haut de l'écran puis à une application plus instructive de type mémory avec les différentes classes d'instrument,...

Finalement, nous avons imaginé un jeu qui permettrait aux utilisateurs de jouer des morceaux de piano mais sous forme ludique, un peu comme le jeu *"Pianotiles"* mais avec un clavier de piano assez grand pour que les notes jouées correspondent aux touches du piano (ce qui n'est pas le cas dans le jeu cité ci-avant). Certaines vidéos qui ont pour but d'apprendre à jouer des morceaux de piano montrent le clavier avec les mains qui jouent et les notes qui descendent au fur et à mesure du haut de l'écran. C'est à partir de ce concept que nous avons commencé à travailler.

Dans un premier temps, nous avions besoin d'un clavier qui fonctionne, c'est-à-dire d'une série de touches (nous avons choisi de représenter 2 octaves) sur lesquelles on pouvait appuyer pour faire le son correspondant.

Ensuite, nous nous sommes occupés de faire descendre les notes correspondantes pour un morceau simple au début : *"Au clair de la Lune"* puis pour des morceaux plus compliqués que vous pourrez découvrir plus loin ou dans l'application.



Aperçu du jeu

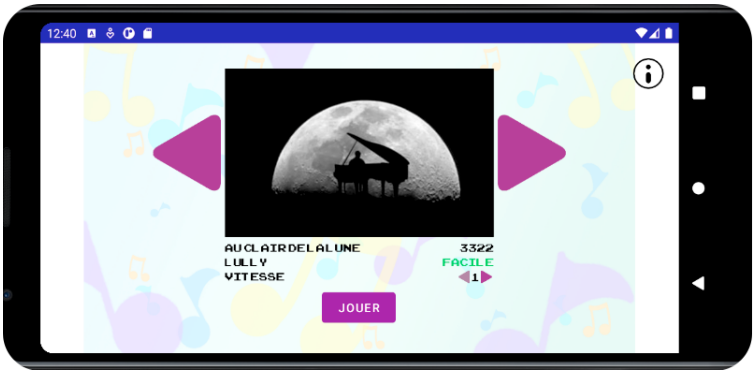
Ces notes devaient non seulement descendre à la bonne place mais également disparaître lorsque l'utilisateur appuyait sur la bonne note au bon moment.

Après cela, il a fallu établir le système de score. Tout d'abord, nous avons établi le fait que lorsque la touche correspondant à la note qui descend est pressée au bon moment, le score augmente. Puis, nous avons amélioré ce système de points, d'abord en raffinant le score donné en fonction de l'écart entre la note et le piano, puis en faisant apparaître la notion de *combo* qui permet de multiplier le score lorsque plusieurs notes correctes sont pressées consécutivement.

Le principal du jeu étant fait, nous avons travaillé sur l'esthétique et l'entrée du jeu : le menu, le retour au menu, le choix des différents morceaux,....

Nous avons opté pour un menu principal avec au milieu une description du morceau, en dessous un bouton pour choisir la vitesse à laquelle le morceau doit être lancé et un bouton pour lancer le jeu. De part et d'autre de ces informations, deux flèches permettant de faire défiler les différents morceaux implémentés dans l'application.

Une fois un morceau lancé, il est possible de le mettre en pause en appuyant sur la flèche retour du téléphone ou sur le bouton pause en haut à droite de l'écran, puis une fenêtre s'affiche pour demander à l'utilisateur s'il veut continuer le morceau en cours ou s'il préfère retourner au menu principal.



Menu de notre application

3 Réalisation

Notre code comporte deux *activity* différentes : MainActivity et PianoActivity. L'activité principale comporte le menu et c'est là que l'utilisateur arrive quand il ouvre l'application. Il peut y choisir le morceau qu'il va jouer ainsi que la vitesse grâce à toutes les informations qui sont présentes et a la possibilité de naviguer entre les différentes pages du menu. PianoActivity renferme tout ce qui est présent lorsqu'une partie est lancée et que le piano apparaît, entre autres le clavier et les notes qui apparaissent au fur et à mesure mais aussi le score et multiplicateur. Douze classes ont été créées pour organiser au mieux le code. Ces classes respectent bien sur le principe d'encapsulation et tous les attributs utilisés par les méthodes de la classe se trouvent eux-mêmes dans la classe. Nous allons donc les présenter une par une et mentionner les liens qui les unissent.

3.1 Classe MainActivity

Commençons par expliciter un peu plus ces deux Activités. MainActivity gère donc tout ce qui se trouve sur la page du menu. Elle permet d'afficher entre autres les informations relatives aux différents morceaux, les flèches qui peuvent se griser quand il n'y a plus d'autres morceaux, les images, la vitesse (qu'on peut choisir),... Et cela grâce à des listes. Elle décrit aussi les niveaux de difficulté en les associant chacun à un nombre allant du plus facile (1) au plus difficile (3).

```
when(difficulty[selector]) {
    1 -> {
        levelDifficulty.text = "Facile"
        levelDifficulty.setTextColor(getColor(R.color.green))
    }
    2 -> {
        levelDifficulty.text = "Moyen"
        levelDifficulty.setTextColor(getColor(R.color.orange))
    }
    3 -> {
        levelDifficulty.text =  "Difficile"
        levelDifficulty.setTextColor(getColor(R.color.red))
    }
}
```

Niveaux de difficulté

C'est également là où on sauvegarde le plus haut score grâce à une fonction que nous découvrirons plus tard et où on le met à jour s'il est supérieur à l'ancien.

Sur cette page on retrouve un bouton *Jouer* qui lance la deuxième activité qu'est *PianoActivity* grâce à la variable *intent* qui sert à présenter cette nouvelle activité.

3.2 Classe PianoActivity

Cette classe s'occupe du jeu en lui-même dès qu'il a été lancé depuis *MainActivity*. Elle fait bien sûr appel à la classe *PianoView* qui sera explicitée plus tard. De plus, elle prend en compte deux fragments : le *ReturnFragment* et le *GameEndFragment* qui sont utilisés pendant le jeu pour mettre pause ou pour afficher les résultats à la fin de la partie. A la fin de la partie, le joueur a la possibilité de retourner au menu,en fermant donc cette activité. Le *garbage collector* effacera alors tous les objets présents lors de la partie tels que les notes ou encore le clavier.

3.3 Classe PianoView

Comme expliqué précédemment, nous avons commencé notre projet en codant seulement un clavier fonctionnel.

Pour afficher tous les composants du jeu, une nouvelle classe *PianoView* a été créée. Elle initialise toutes les couleurs de cette nouvelle *view*, les éléments qui apparaîtront dessus tels que le score, la fonction *soundpool* qui va permettre de jouer et gérer les sons, les paramètres audio, la variable booléenne *drawing* qui générera ou pas le dessin des touches (pour qu'il s'arrête quand on met le jeu en pause par exemple) et quelques autres variables dont la fonction sera décrite par après.

Les variables correspondant à la taille de l'écran sont également initialisées ici mais leur valeur changera plus tard grâce à la fonction *onSizeChanged*.

```
@RequiresApi(Build.VERSION_CODES.LOLLIPOP)
class PianoView @JvmOverloads constructor
        (context: Context, attributes: AttributeSet? = null, defStyleAttr: Int = 0):
        SurfaceView(context, attributes,defStyleAttr), SurfaceHolder.Callback, Runnable {

    var sharedPreference: SharedPreference = SharedPreference(context)
    private lateinit var canvas: Canvas
    private lateinit var thread: Thread
    var screenWidth: Float = 0f
    var screenHeight: Float = 0f
    private val backgroundPaint = Paint()
    private val piano: Piano = Piano( view: this)
    private var drawing = false
    var score: Score = Score()
    var levelId: Int = 0
    var started = false
    var acc = false

    private val textColor = Paint()
    private val redTextColor = Paint()
```

Variables de la classe PianoView

Ensuite, les notes sont initialisées une par une et associées aux sons que nous avons importés auparavant dans le dossier *raw* et qui correspondent aux notes des 2 octaves du piano.

```
val audioAttributes : AudioAttributes!  = AudioAttributes.Builder()
        .setUsage(AudioAttributes.USAGE_ASSISTANCE_SONIFICATION)
        .setContentType(AudioAttributes.CONTENT_TYPE_SONIFICATION)
        .setUsage(AudioAttributes.USAGE_MEDIA)
        .build()

soundPool = SoundPool.Builder()
        .setMaxStreams(26)
        .setAudioAttributes(audioAttributes)
        .build()

soundMap = SparseIntArray( initialCapacity: 25)
soundMap.put(0, soundPool.load(context, R.raw.c4,  priority: 1))
soundMap.put(1, soundPool.load(context, R.raw.cd4,  priority: 1))
soundMap.put(2, soundPool.load(context, R.raw.d4,  priority: 1))
soundMap.put(3, soundPool.load(context, R.raw.dd4,  priority: 1))
soundMap.put(4, soundPool.load(context, R.raw.e4,  priority: 1))
soundMap.put(5, soundPool.load(context, R.raw.f4,  priority: 1))
soundMap.put(6, soundPool.load(context, R.raw.fd4,  priority: 1))
soundMap.put(7, soundPool.load(context, R.raw.g4,  priority: 1))
soundMap.put(8, soundPool.load(context, R.raw.gd4,  priority: 1))
soundMap.put(9, soundPool.load(context, R.raw.a4,  priority: 1))
soundMap.put(10, soundPool.load(context, R.raw.ad4,  priority: 1))
```

Initialisation des sons

Fonctions *Pause* et *Resume*

Cette *View* comporte aussi la fonction *Pause* qui fait intervenir la variable booléenne *Drawing* et qui ajoute la possibilité de mettre pause pendant le jeu en faisant prendre la valeur *False* à *Drawing* et donc les notes arrêtent d'être dessinées. Cette fonction a évidemment sa complémentaire *Resume* pour pouvoir recommencer le jeu après l'avoir mis en pause. Elle fait appel à la fonction *Started* pour savoir si le jeu a commencé ou pas. En effet, au début cette variable booléenne est *False* et dès que les notes commencent à apparaître, elle prend la valeur *True*.

```
fun pause() {
    pauseMusic()
    drawing = false
    thread.join()
}

fun resume() {
    if (started) playMusic()
    drawing = true
    thread = Thread( target: this)
    thread.start()
}
```

Fonctions Pause et Resume

Fonction *run*

Nous avons ensuite créé une fonction qui met en mouvement les notes qui descendent. Cette fonction *run* permet que, tant que les notes sont dessinées (tant que *Drawing* est *True*), un certain intervalle est calculé et la distance à laquelle doivent se trouver les notes à la prochaine frame peut être obtenue grâce à cela et à la vitesse qui sera définie plus tard en les multipliant : $d = t \times v$.

Fonction *onSizeChanged*

Ensuite vient la fonction *onSizeChanged* qui gère en général la taille des éléments. Elle permet d'une part que notre *View* occupe toute la place qu'il y a sur l'écran, d'autre part, elle définit la taille du clavier de piano en faisant en sorte que sa hauteur soit équivalente à un quart de la hauteur de l'écran. Et enfin, elle définit la largeur des touches à partir de la largeur du piano et du nombre de touches et va chercher les dimensions du piano et des notes dans la classe *Piano* et dans la classe *Notes*.

```
override fun onSizeChanged(w: Int, h: Int, oldw: Int, oldh: Int) {
    super.onSizeChanged(w, h, oldw, oldh)
    screenHeight = h.toFloat()
    screenWidth = w.toFloat()

    piano.width = w.toFloat()
    piano.height = h.toFloat()
    piano.pianoTop = 3*h/4f

    piano.whiteKeyWidth = piano.width/piano.nTouchesBlanches

    piano.setPiano()
    for(note : Note in notes) {
        note.setNote()
    }
}
```

Fonction onSizeChanged

Fonction *Draw*

Cette fonction est appelée à chaque fois que l'on veut dessiner une nouvelle frame. Grâce à cette fonction, le programme affiche tous les composants nécessaires à l'écran : le fond blanc puis toutes les touches en appelant à chaque note la classe piano pour bénéficier des bonnes dimensions. Puis, les textes du score et du combo sont dessinés avec leurs valeurs correspondantes provenant de la classe *Score*.

```
private fun draw() {
    if(holder.surface.isValid) {
        canvas = holder.lockCanvas()
        canvas.drawRect( left: 0f,  top: 0f, canvas.width.toFloat(), canvas.height.toFloat(), backgroundPaint)
        for(note : Note in notes) note.draw(canvas)
        piano.draw(canvas)
        canvas.drawText( text: "Score : " + score.score,  x: 30f,  y: 100f, textColor)
        canvas.drawText( text: "Combo : x" + score.multiplier,  x: 30f,  y: 200f, redTextColor)
        holder.unlockCanvasAndPost(canvas)
    }
}
```

Fonction Draw

La fonction *updatePositions* permet de mettre à jour la position des notes qui descendent comme l'indique son nom. Elle fait appel à la classe note et lorsque qu'elle détecte que la dernière note est jouée *Drawing* devient *False* et le fragment de fin de jeu 4.2 s'affiche avec toutes les informations qui lui sont propres. Ce fragment s'affiche grâce à la fonction *showGameEndFragment*.

Fonction *onTouchEvent*

Dessiner les touches avec la fonction *Draw* était une première étape déjà assez satisfaisante mais qui ne répondait pas encore aux besoins de notre projet. En effet, il a fallu par la suite rendre ces touches sensibles à la pression qu'exercerait l'utilisateur sur l'écran tactile pour jouer des notes. C'est pour cela que nous avons utilisé la fonction*onTouchEvent*. Elle permet de localiser le point où est exercée la pression et d'en extraire ses coordonnées. Ensuite, elle va en déduire la touche entière sur laquelle l'utilisateur a voulu appuyer et demander à la classe *Piano* qu'elle est la note correspondante. Par la suite, connaissant la touche qui a été pressée, elle va aller rechercher dans la classe *Note* si une note descendait justement au dessus de cette touche, si oui elle va la supprimer de l'écran et augmenter le score. Enfin, elle indique que la touche est relâchée quand il n'y a plus de pression appliquée dessus.

La dernière étape était d'associer l'action de presser une touche avec celle de jouer un des sons enregistrés dans *raw*. Pour cela, les fonctions *playSound* a été implémentée.

Fonctions de chargement

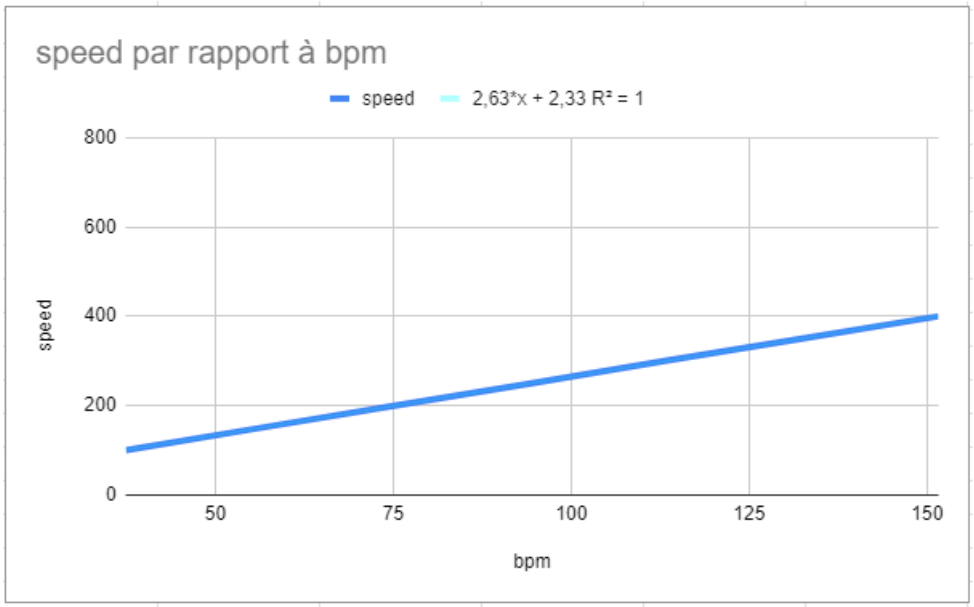
Avant de pouvoir descendre sur l'écran, les notes ont besoin d'être chargées, d'où la fonction *loadNotes* qui va nous le permettre. Elle charge donc tous les paramètres nécessaires qui sont extraits du fichier *xml* comme la position, la durée de la note, la vitesse,... Cette dernière peut être adaptée dans le menu grâce à l'ajout d'un coefficient qui change selon le numéro de la vitesse choisi.

```
fun loadNotes(id: Int, spCoeff: Int) {
    val loadArray: Array<String> = resources.getStringArray(id)
    val speed: Float = loadArray[0].toFloat()
    val coeff: Float = when(spCoeff) {
        1 -> 0.75f
        2 -> 1f
        3 -> 1.25f
        else -> 1f
    }
    var pitch: Int
    var duration: Float
    var position = 1f
    for(i : Int  in 1 until loadArray.size) {
        pitch = loadArray[i].split( ...delimiters: ";")[0].toInt()
        duration = loadArray[i].split( ...delimiters: ";")[1].toFloat()
        if(i == 1) {
            notes.add(FirstNote( speed: speed * coeff, piano,  view: this, pitch, position, duration))
        }
        else notes.add(Note( speed: speed * coeff, piano,  view: this, pitch, position, duration))
        position += duration
    }
    levelId = id
    loadMusic(id)
}
```

Fonction de chargement des éléments

Dans le fichier *Morceaux.xml*, toutes les notes nécessaires à chacun des morceaux sont écrites

dans l'ordre avec leur *pitch* et leur durée. (3.10 Pour la durée, nous avons décidé de prendre comme référence la noire, ce qui est le plus courant en musique. Une unité de durée correspondant donc à la noire, une demi à une croche et deux à une blanche.



Graphique de correspondance entre les bpm et le tempo à la noire du morceau

La fonction *loadMusic* va quant à elle charger l'accompagnement s'il y en a un pour le morceau choisi. Si c'est le cas, l'arrêt du morceau et l'affichage de la fenêtre de fin se basent sur la fin de l'accompagnement. La gestion de cet accompagnement se fait également grâce aux fonction *playMusic* qui démarre l'accompagnement, *pauseMusic* qui le met en pause et *stopMusic*.

3.4 Classe Note

Comme souvent évoqué précédemment, *PianoView* fait appel à d'autres classes telle que la classe *Note* dans différentes fonctions. C'est dans cette classe que sont définies toutes les variables concernant chaque rectangle qui descend du haut de l'écran et qui sert dans notre jeu à indiquer les touches sur lesquelles doit appuyer l'utilisateur. Notamment leur couleur, que nous avons décidé de mettre en aléatoire et leur taille. Il y a donc une chance pour que la note soit de la même couleur que le fond mais c'est le cas d'une note sur 16 millions, ce qui est négligeable. Leur épaisseur est choisie de manière arbitraire avec comme seule condition d'être strictement inférieure à celle des touches de manière à ce que chaque rectangle corresponde à une seule note et donc indique une seule touche. Leur longueur de base est arbitraire également mais elle est ensuite multipliée par la durée de la note à jouer. La valeur de la coordonnée y du rectangle est définie grâce à la position et la longueur du rectangle à laquelle ont rajoute quelques millimètres pour

que les notes soient un minimum espacées afin que l'on différencie une même note longue de trois temps de trois fois la même note d'un seul temps (cf *Au clair de la lune*)

Variable *noteOnScreen*

Ces notes peuvent apparaître seulement lorsque la variable booléenne *noteOnScreen* prend la valeur *True*. Cela nous permet d'indiquer que lorsque la note est arrivée au niveau du clavier ou bien si la note est bien jouée, elle ne doit plus apparaître sur l'écran et la fonction prend alors la valeur *False*. Cela active alors *deleteNote* qui est utilisée plus tard pour justement supprimer des notes. Par ailleurs, pour éviter un travail non nécessaire, les notes ne sont dessinées que lorsqu'elles arrivent assez proches de l'écran grâce à une condition sur la fonction *Draw* de cette classe qui va dessiner toutes les notes.

Fonction *update*

Cette fonction va permettre de modifier la position de ces notes au fur et à mesure de leur descente grâce à un intervalle et à la vitesse qu'on a définit plus haut. Elle prend aussi en charge les notes qui on été ratées si elle sont toujours sur l'écran (ce que l'on sait grâce à la variable *noteOnScreen*). Dans ce cas, elle remet le multiplicateur à zéro et n'ajoute rien au score.

Fonction *detectPlay*

Cette fonction est assez importante dans notre projet, elle permet de détecter lorsqu'on appuie sur une touche. Elle est donc indispensable à la fonction *onTouchEvent* qui elle traduit la pression détectée en coordonnées. Si la note est au niveau du clavier et que c'est la bonne note, elle est supprimée et le score est augmenté avec le multiplicateur et tout ce qui s'en suit. Si la touche pressée n'est pas la bonne, le multiplicateur est remis à zéro.

Fonction *setNote*

A la fin de cette classe, on associe chaque touche à un pitch avec une fonction qui prend en compte le fait que il y ait des touches noires aussi. Et ensuite, on réinitialise les valeurs du rectangle avec les bonnes coordonnées.

3.5 Classe Piano

Cette classe prend en compte la taille de l'écran, le nombre de touches blanches que l'on veut,... Elle va servir à construire la partie plus fonctionnelle du piano quand la classe *PianoView* gérait plus l'aspect esthétique. Elle commence donc par créer deux listes, une de touches noires et une autres de touches blanches.

Fonction *pressKey*

Lorsqu'une pression est exercée sur l'écran, la classe *PianoView* appelle cette fonction. Elle va en effet pouvoir l'éclairer et lui dire quelle touche contient les coordonnées de la pression detectée. Elle va d'abord regarder dans les touches noires si c'est le cas puis dans les touches blanches puis retourner le *pitch* de la touche. La fonction *release* permet elle de relever la note après qu'elle ait été pressée. Si la note a été pressée, elle se grise et puis reprend sa couleur initiale quand elle est relevée, cela grâce à la fonction *Draw* qui a aussi comme mission de dessiner les lignes entre les notes.

Fonction setPiano

Enfin, la fonction *setPiano* met en place le piano en définissant la taille des touches grâce aux nouvelles valeurs de l'écran et ensuite en associant chaque touche à un son grâce aux *pitch* après avoir bien positionner ces dernières.

3.6 Classe FirstNote

Cette classe a pour but de créer une note fictive qui aura pour mission de lancer l'accompagnement du morceau s'il y en a un. Dans notre cas, ce sera pour la *Fantaisie-impromptu* de Chopin et *Ce rêve bleu* de Aladdin auxquels nous avons rajouté l'accompagnement. Cette classe hérite de la classe *Note*, elle reprend les mêmes variables et méthodes que cette dernière mais elle en rajoute une (*playSound* qui joue l'accompagnement) et elle en modifie une (*update*). La fonction *update* ne prend dès lors plus le score en compte car c'est une note fictive et c'est elle qui active la fonction ajoutée. Une fois qu'elle atteint le clavier, la variable *started* devient *True* et l'accompagnement se lance.

3.7 Classe PianoKey

Cette classe est assez particulière car elle ne prend en compte que très peu d'éléments. En effet, elle ne contient que la variable booléenne *isPressed* qui a la valeur *False*. Elle sert à savoir si une touche est pressée ou pas, elle est entre autres utilisée dans la classe *Piano* pour la fonction *pressKey*. Il était nécessaire de l'isoler dans une classe pour pouvoir individualiser chaque pression pour chaque note.

3.8 Classe Score

Tout au long de la partie, un score est calculé en fonction du nombre de touches correctes et un niveau de précision est établi. Cette classe va s'occuper de calculer et stocker les performances de la partie. Il calcule donc le score en fonction du multiplicateur qui pour plusieurs notes correctes pressées d'affilée ajoute un coefficient devant les points gagnés par note.

Nous l'avons codé de sorte que dès qu'il y a plus de sept notes correctes consécutives, le coefficient multiplicateur augmente de un et ce, jusqu'à huit. Ensuite, il stagne et revient à zéro quand une note incorrecte est jouée. Nous l'avons limité à une multiplication du score par huit pour éviter que les joueurs professionnels comme par exemple un des membres éminents de note groupe **Liam** ne marquent des scores si grands que l'écriture du nombre obtenu dépasserait la largeur de l'écran.

```
fun increaseMultiplier() {
    consecutiveNotes += 1
    if(consecutiveNotes == 7 && multiplier <= 8) {
        multiplier *= 2
        consecutiveNotes = 0
    }
}

fun resetMultiplier() {
    consecutiveNotes = 0
    multiplier = 1
}

fun countCorrect(isPressed: Boolean) {
    if (isPressed) {
        correctNotes += 1
    }
    totalNotes += 1
}

fun precision(): Float {
    if(totalNotes > 0) return correctNotes.toFloat()/totalNotes*100
    return 0f
}
```

Classe score

Enfin, cette classe calcule la précision dont on a fait preuve durant le morceau. Le nombre de note correctes stockées dans la variable *correctNotes* est divisé par le nombre total de notes du morceau et une multiplication du résultat par cent nous donne le pourcentage de réussite.

3.9 Classe SharedPreference

Nous savions que l'utilisateur serait amené à fermer notre application assez souvent pour la ré-ouvrir par après et nous avions envie qu'il puisse conserver la valeur de son plus haut score à la réouverture. Cela étant, il peut, d'une partie à l'autre, observer sa progression et challenger ses amis en comparant leurs performances aux siennes. Et tout cela grâce à la fonction *save* qui sauvegarde le highscore pour chaque morceau. La fonction *getValueInt* quant à elle permet d'afficher le score enregistré.

3.10 Fichiers XML

Pour la mise en page de notre projet, nous avons utilisé beaucoup de fichiers*XML*. Un pour chaque activité et trois pour les trois fragments que nous faisons intervenir dans ces activités. Ces fragments ont été rajouté après pour augmenter la qualité de notre jeu et seront donc expliqués au chapitre suivant. Pour chaque fichier, nous avons utilisé l'espace design pour rajouter nos boutons, gérer les couleurs et la place des éléments. Nous en avons aussi utilisé pour les couleurs bien sur et également pour stocker les notes de chaque morceau. Nous encodions chaque note une par une dans l'ordre du morceau grâce à son pitch et sa durée. Puis nous testions pour voir si cela rendait bien et si il y avait besoin d'un accompagnement.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- 0: Speed (float), 1..n: Pitch;Durée -->
    <string-array name="Scale">
        <item>600</item>
        <item>1;1</item>

        <item>1;1</item>
        <item>2;1</item>
        <item>3;1</item>
        <item>4;1</item>
        <item>5;1</item>
        <item>6;1</item>
        <item>7;1</item>
        <item>8;1</item>
        <item>9;1</item>
        <item>10;1</item>
        <item>11;1</item>
        <item>12;1</item>
        <item>13;1</item>
```

Morceaux.xml

3.11 Diagrammes

Diagramme de séquence

Pour bien comprendre le fonctionnement du programme, nous avons réalisé un diagramme de séquence qui décrit le déroulement des opérations lorsque l'utilisateur interagit avec l'écran pendant une partie. On y retrouve bien sûr certaines des classes et méthodes décrites ci-dessus, et les interactions/messages entre objets (qu'ils soient synchrones ou asynchrones). Enfin, ce diagramme permet de représenter les durées d'activités des différents objets pendant la durée d'exécution de cette partie du programme.

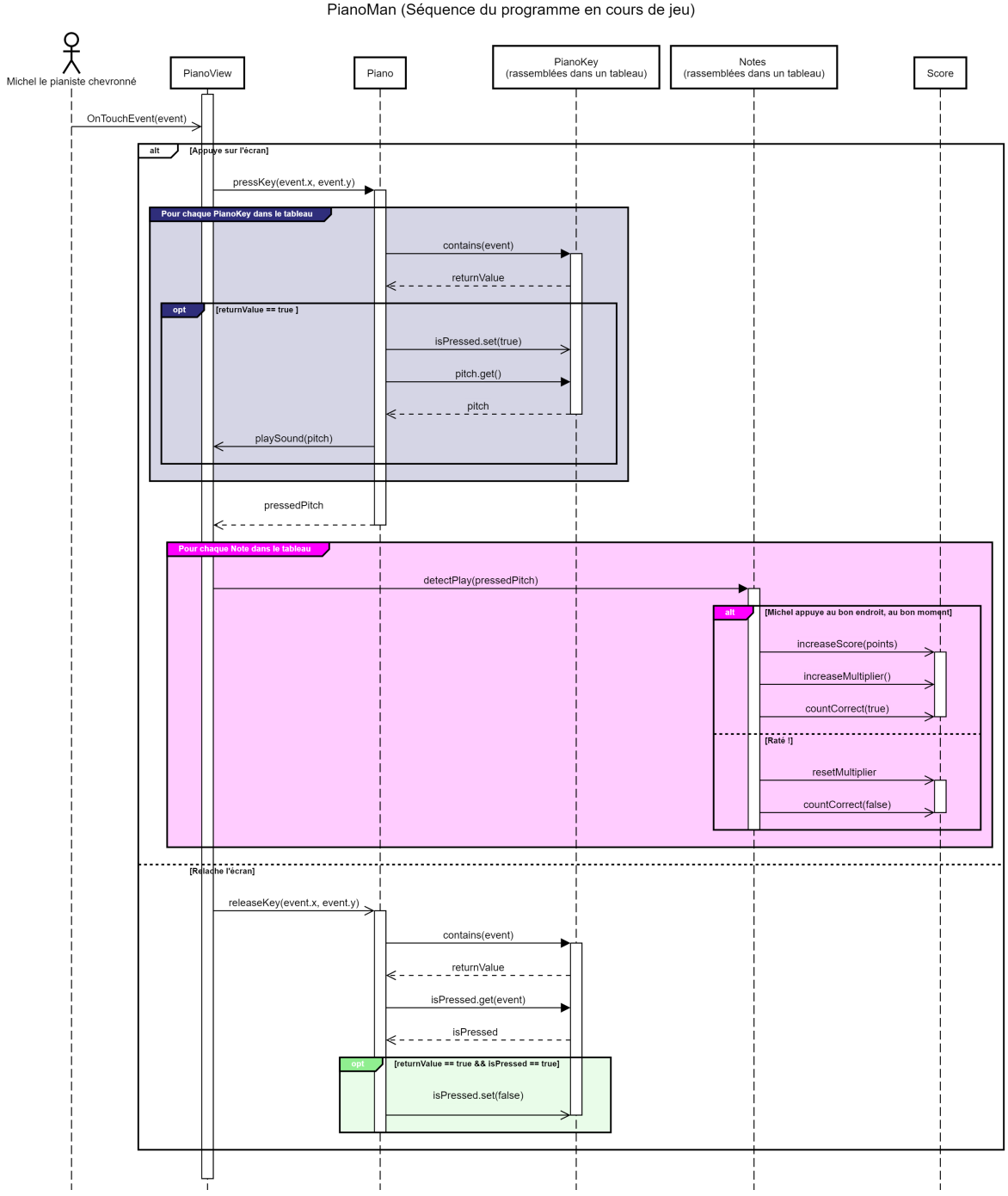


Diagramme de séquence lors d'une interaction de l'utilisateur en jeu

Diagramme de classe

Finalement, toute la description du programme réalisé est résumée sur le schéma de la page suivante, un diagramme de classe. Celui-ci reprend toutes les classes, leurs attributs et méthodes mais également les différents liens entre celles-ci (association, composition, héritage,...).

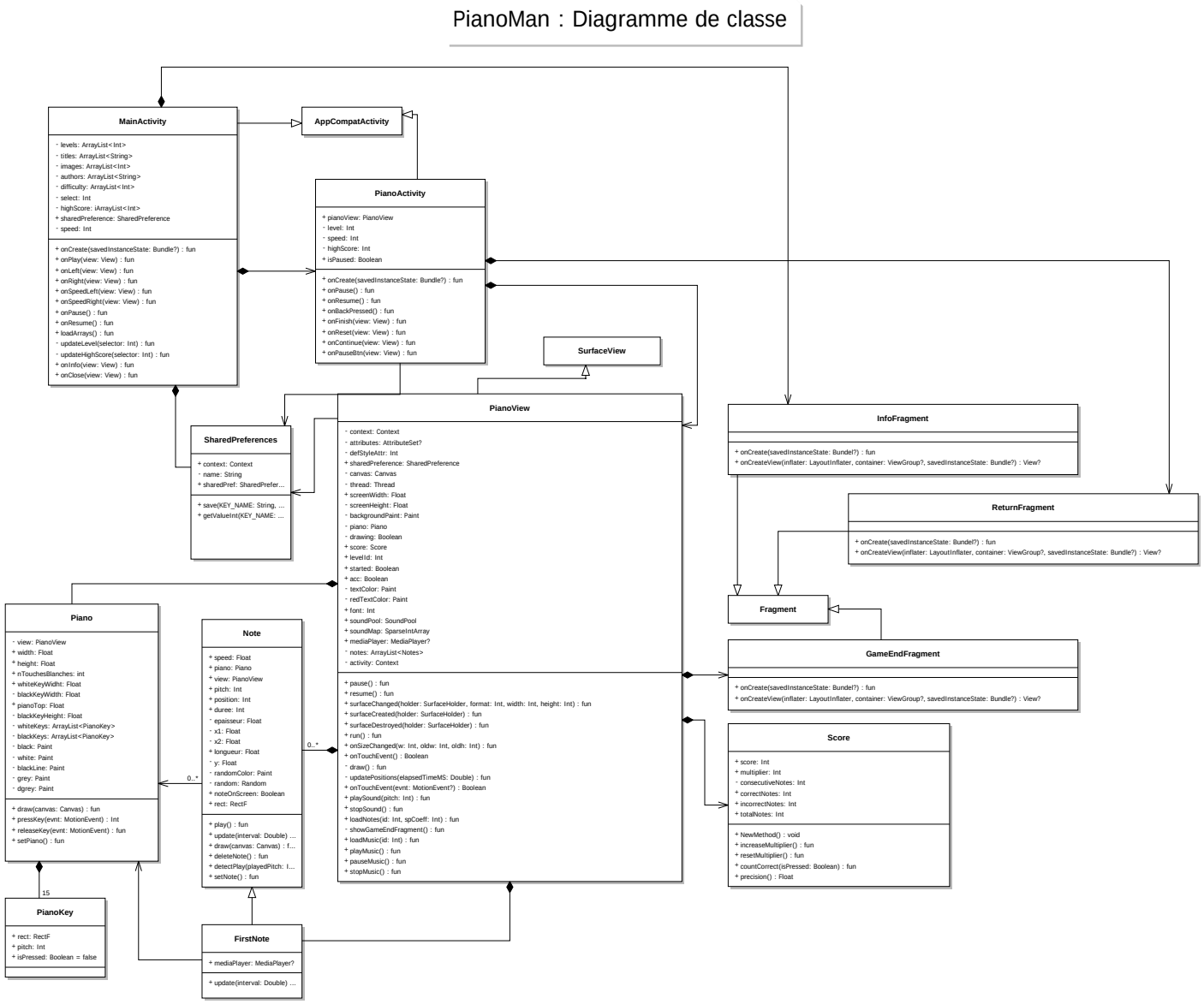


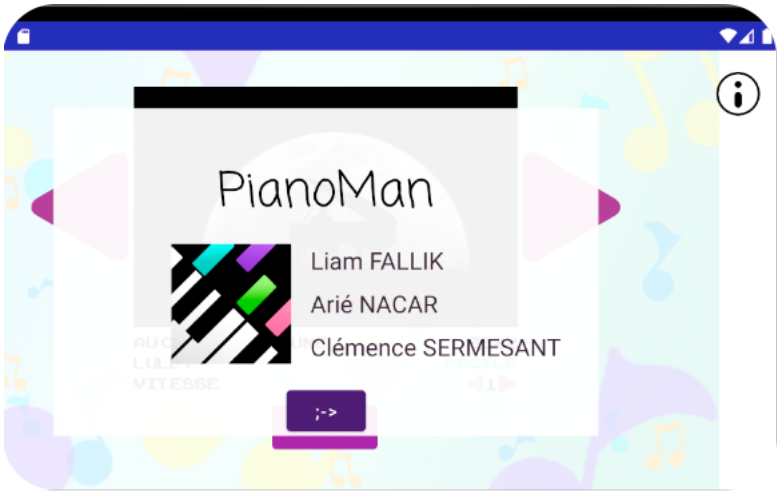
Diagramme de classe du programme

4 Finalisation

Après avoir réalisé notre application et l'avoir rendue parfaitement fonctionnelle, nous avons voulu rajouter quelques petites options supplémentaires pour améliorer notre projet. Ces fonctions ont déjà été pour la plupart mentionnées dans le chapitre précédent lors de l'explication des classes. Par exemple, le fait que la vitesse soit réglable à partir du menu fut rajouté par après, de même que le fait de sauvegarder le highscore,...

4.1 Fragment d'information

Sur la page du menu, on retrouve une petite icône d'information. En la touchant, un fragment *InfoFragment* s'ouvre pour laisser apparaître une fiche d'information qui présente l'application et ses créateurs.



InfoFragment

4.2 Fragment de fin

Lorsque le joueur a fini la partie, nous voulions qu'il ait un accès à un retour au menu et à son score et sa précision finale. De plus nous voulions qu'il puisse recommencer le morceau directement à la fin pour pouvoir s'améliorer. Nous avons donc créé un deuxième fragment *GameEndFragment* qui avait pour mission d'afficher ces informations à la fin. C'est la raison d'être de la fonction *showGameEndFragment*.

GameEndFragment

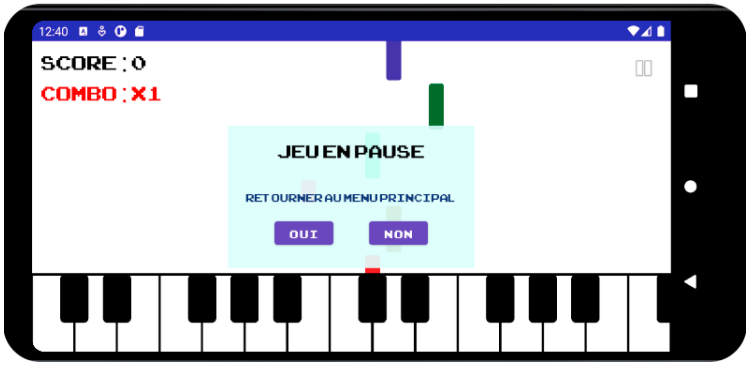
Ce fragment reçoit comme variable le score, la précision, une lettre pour le grade et un texte pour la précision et pour le score pour définir les chiffres qui s'afficheront. Le grade sera défini en fonction du degré de précision par une lettre allant de F à A respectivement du moins bon au meilleur avec en plus la lettre S pour les performances incroyables qui auraient plus de 95% de réussite. Chaque lettre étant associée à une couleur pour une question d'esthétique.

```
scoreText.text = args?.getInt( key: "score").toString()
precisionText.text = precision.toString() + " %"
if (precision != null) {
    if (precision < 50f) {
        gradeLetterText.text = "F"
        gradeLetterText.setTextColor(resources.getColor(R.color.red))
    }
    if (50 <= precision && precision < 60) {
        gradeLetterText.text = "E"
        gradeLetterText.setTextColor(resources.getColor(R.color.orange))
    }
    if (60 <= precision && precision < 70) {
        gradeLetterText.text = "D"
        gradeLetterText.setTextColor(resources.getColor(R.color.yellow))
    }
    if (70 <= precision && precision < 80) {
        gradeLetterText.text = "C"
        gradeLetterText.setTextColor(resources.getColor(R.color.lgreen))
    }
    if (80 <= precision && precision < 90) {
        gradeLetterText.text = "B"
        gradeLetterText.setTextColor(resources.getColor(R.color.dgreen))
    }
```

Code des différents grades

4.3 Fragment de pause

Durant la partie, le joueur peut mettre le jeu sur pause si l'envie lui prend en appuyant sur les deux traits verticaux, symbole de la mise en pause, en haut à droite de l'écran ou bien en appuyant sur la flèche retour du téléphone comme implémenté de base dans Kotlin. S'affiche alors un nouveau fragment *ReturnFragment* qui permet de soit continuer la partie soit retourner au menu sans devoir attendre le fragment de fin de partie.4.2



Fragment de mise en pause du jeu

5 Conclusion

Ce projet fut pour nous une réelle occasion de s'approprier la programmation orientée objet et toutes les notions qui lui sont associées comme l'héritage, le polymorphisme, et pleins d'autres encore. Nous avons acquis des compétences de programmation qui nous ont donné accès au monde assez méconnu de la création d'application et jeux que nous utilisons pourtant tous les jours.

Par ailleurs, ce projet nous a permis également de partager de très bons moments en groupe, ce qui n'est pas chose aisée en ce moment...

Nous avons également eu l'occasion de penser à quelques améliorations pour notre jeu s'il était amené à devoir être commercialisé. Plusieurs mises à jour pourront être développées comme par exemple l'ajout de morceaux supplémentaires (qui serait très aisé grâce au fichier *xml*), l'accès restreint à certains morceaux tant qu'un score minimum n'est pas atteint, le changement de design du clavier et des sons qui serait possible grâce à des points obtenus en fonction de la précision à la fin de chaque partie. Des achats intégrés permettront d'accéder à des morceaux, des claviers et des sons exclusifs ! Pour rajouter les morceau plus rapidement à l'application, un script *Python* pourrait être développé afin de convertir des fichiers midi en fichiers *xml* au format adapté.