Swing it - Developpement mobile 2 (iOS & Android)

Olivier PICARD, M1 informatique

7 novembre 2018

Résumé

Swing it est un jeu dont l'expérience varie en fonction de l'utisateur. Le jeu utilse les différents capteurs du périphérique. sur deux platformes en programmation native. La version iOS[**?**] est programmé en Swift et celle android[**?**] programmé en Java.

1 Introduction

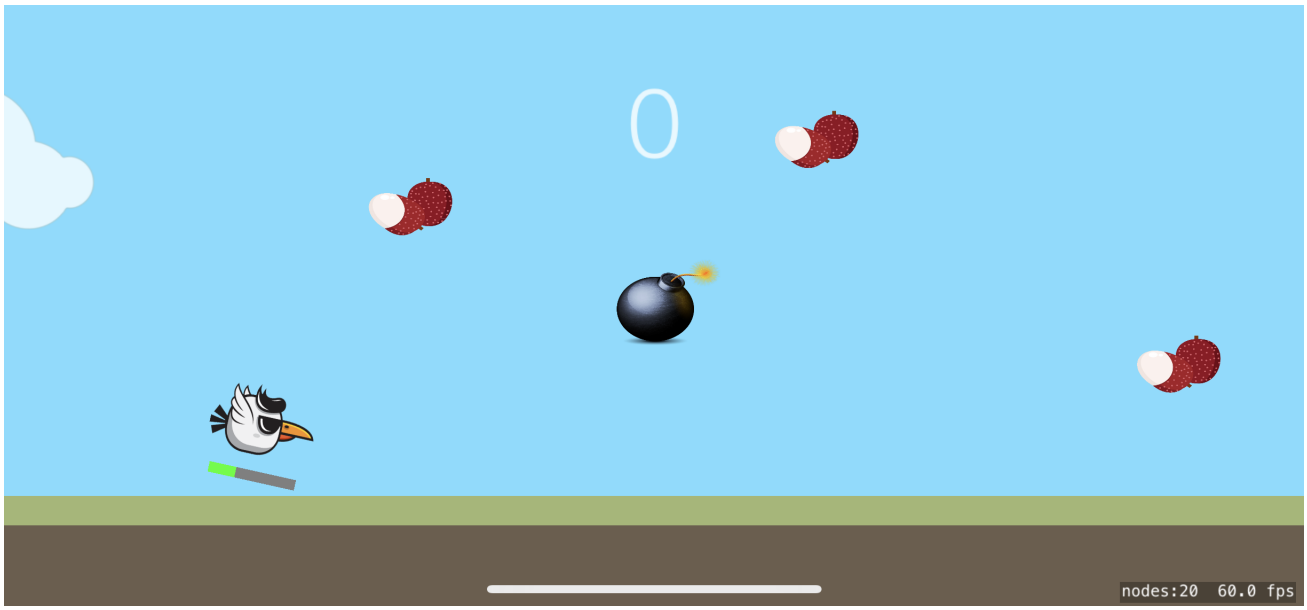
Swing it est un jeu développé dans le cadre du cours de développement mobile. Le but étant de réaliser un jeu en programmation native à la fois pour iOS et pour Android. Pour que le code soit le plus recyclable entre les deux platforme Swing it se base sur une API faite maison. L'API a été réalisée en L3 dans cette même UE et a été grandement mis a jour. Elle prend en charge maintenant quelque capteurs du péréphérique, les vecteurs et elle gère nativement les collisions en utilisant des interfaces(protocoles en swift) ce qui simplifie grandement la conception. Même sur iOS la méthode faite maison en utlisant les protocoles à été préféré plutôt que celle faite par Apple en utilsant les masques de bits.

2 Contraintes

première contrainte faire en sorte que les bombes soit visiblent et qu'il y ai une marge pour que les joueurs puissent avoir le temps de réagir entre le moment ou il voit une bombe et le temps que le personnage va prendre pour se déplacer.

deuxième contrainte utilisation de l'accéromètre. Pour faciliter grandement et éviter de nombreuse sources de bugs, nous avons choisit de verrouiller le périphérique en mode paysage sur la droite. Parce que supposons que le joueur incline le téléphone de plus de 90 dans ce cas le périphérique va effectuer une rotation et les commandes seront inversées.

troisième contraite utilisation de l'appareil photo. Afin d'ajouter une touche d'originalité au projet, nous avons utilisé l'appareil pour pourvoir scanner des QR Code afin de modifier le jeu en fonction de ce que l'on scanne



3 Description général du jeu

Un oiseau vole et doit manger en plein vole pour pouvoir survirvre. Il doit éviter les bombes qui sont dans les airs et il doit manger régulièrement pour pouvoir avoir suffisament d'énergie pour pouvoir continuer à voler. La particularité de ce jeu est qu'il est personalisable en fonction de l'utilisateur. Grace à l'appareil photo on peut scanner des QR Code qui vont modifier l'expérience du jeu. On peut ainsi changer les décors enlever les bombes et bien plus... Dans le jeu il peut y avoir un temps nuageur, éclaircie, pluvieux, et le plus beau orageux. Les items à récupérer peuvent être des lectchie, jambon, pain... Ce sont des modification mineurs sur le jeu mais ce qui est original c'est que le jeu peux être totalement personnalisé, on peut même avoir un mode sans bombe si elles nous dérange.

4 Android

Pour Android nous avons repris, la bibliothèque(API) "Graphics" crée en L3 dans le cadre de ce même cours. C'est déjà le deuxième jeu qui est batie sur cette même API ce qui montre la robustesse de l'architecture. Le concepte de cette API étant à la base de pouvoir reprendre l'intégralité du code iOS (swift - SpriteKit) et de le remettre sous android avec un nombre mineur de modification (outre la syntaxe de language). Cette fois ci les chose on été un peu différente et la conception a démarer par Android. puis porter sur iOS. Le coeur de "Graphics" est resté la même, les syntaxes et la méthodes de rendu est la même, mais quelque mise à jour ont été effectué pour cette nouvelle version.

4.1 Collision

Maintenant la bibliothèque prend en charge nativement la gestion des collisions. Auparavant on vérifiait les collisions entre chaque items du jeu. Sur la capture d'écran du jeu ci-dessus on aurait par exemple : 27 vérifications de collisions à chaque frame(60 fois par secondes) et cici

juste pour 6 items sur l'écran. Car chaque item était considéré comme un éléments qui accepte la collision. Après la mise a jour, on utilise les interfaces pour pouvoir vérifier les collisions. La grandes différence est qu'on segmente les collisions en deux groupes. Les collisionneurs actifs et passif. Ceux qui sont passifs vont servir uniquement à un vérification de leur coordonés et taille. En aucun cas il seront analysé par le moteur graphique. Il indique seulement qu'on peut entré en collision avec eux. Ceux qui sont actifs seront analysé par le moteur de jeu, il va voir si l'élément actif courant et vérifier si chaque éléments passif visible est en collision en ce dernier. Ce qui fait eau total pour la capture d'écran ci contre 5 vérifications de collision, pour 6 items sur l'écran.

```
public interface IGCollisionable
{
    Rect getBound();
}

public interface IGCollisionListener extends IGCollisionable
{
    void collisionEnter(@NonNull IGCollisionable collisionable);
    void collisionExit(@NonNull IGCollisionable collisionable);
    List<IGCollisionable> getCollisionItems();
    void setCollisionItems(List<IGCollisionable> itemInCollision);
}
```

On peut faire la différence entre IGCollisionable qui représente les collisions passives et qui n'est jamais mis au courant dans le cas d'une collision. Et on trouve les collisions activent représentées par IGCollisionListener qui vont être informé par le moteur s'il y a collision, l'objet implémentant cette interface sera notifié lorsque l'objet entre en contact et lorqu'il ne l'est plus. Ca reprend les même nom de fonction que dans le célèbre moteur de jeu "Unity". Le moteur de jeu sera fournir aussi à l'objet tous les items avec lesquels il est en collision.

4.2 Out of screen

Le moteur de jeu gère maintenant la sortie hors de l'écran. Dans l'ancienne version à chaque frame on vérifiait pour chaque item quelque soit ça nature si son bord gauche ainsi que son bord droitn, bas et haut n'étaient pas sorti de l'écran. Voici l'algorithme avant

```
public boolean isOutOfScreen() {

    // Anchor point 0.5 par défaut donc on divise la taille par 2
    if (getPosition().y + getSize().height / 2 < -MARGIN_OUT_OF_SCREEN_TO_DELETE
            || getPosition().y - getSize().height / 2 > this.getScene().getSize()
                                    .height + MARGIN_OUT_OF_SCREEN_TO_DELETE
            || getPosition().x - getSize().width / 2 > this.getScene().getSize()
                                    .width + MARGIN_OUT_OF_SCREEN_TO_DELETE
            || getPosition().x + getSize().width / 2 < -MARGIN_OUT_OF_SCREEN_TO_DELETE) {
        return true;
```

```
    }
    return false;
}
```

Et voici l'algorithme actuellement.

```
public interface IGDeletable
{
    boolean canBeDeleted();
}

override fun canBeDeleted(): Boolean
{
    return position.y - size.height / 2 >= scene.size.height
}
```

On peut voir qu'il a été simplifié à l'extrême. On a juste a dire quel sont les conditions pour lesquels un item peu être suprimmer. Ca ne sert à rien de vérifier les bords supérieur et inférieur d'un objet qui va se déplacer seulement à l'horizontal. On donnant à l'objet ses propres conditions de suppression, ce qui allège le temps de traitement pour chaque frame.

Sur la version android, avec ces différentes techniques on arrive à gagner quelque FPS sur le global du jeu. Avant on était entre 22 et 25 FPS et maintenant on atteint les 30 à 38 FPS.

4.3 Accéléromètre

Dans la mise à jour du moteur nous avons ajouter la prise en charge de l'accéléromètre intégrés au moteur de jeu lui même. C'est à dire que le moteur fourni à l'utilisateur final des fonctions déjà préfabriquées pour lui éviter d'avoir à implémenter lui même les capteurs et à manipuler les données brut de ce capteur. Le moteur "Graphics" va lui même détecter l'orientation du téléphone et fournir des données stable indépendement de tous autres informations. Par exemple quand l'utilsateur penche le périphérique, peut importe l'orientation les nombres négatifs seront pour une inclinaisont vers le sols et les positif l'inverse.

Le moteur de jeu, ajoute en transparence pour l'utilsateur final, un seuil virtuel. Ce seuil va déterminer, à partir de quel point on considère que le périphérique à été incliné. Ce seuil peut être modifié à tous moment par l'utilsateur. Et si jamais les fonctions prédéfinies dans le moteur, ne suffisent pas on peut tout de même accéder au données brut fourni par android.

L'avantage est aussi qu'avec la prise en charge de l'accéléromètre nativement par le moteur, c'est qu'on peut utiliser avoir accès à l'accéléromètre depuis n'importe ou et pas seul en faisant le lien avec un "Activity" On peut voir ci-dessous la seul fonction nécessaire pour utitiliser l'accéléromètre.

```
override fun onAccelerometerEvent(axisValues: FloatArray)
{
    super.onAccelerometerEvent(axisValues)
    if(!isAccelerometerEnable || gameState != GameState.PLAY) return
    val directionVector = GVector(axisValues[1], axisValues[0])
```

```
    if(directionVector.dy != 0f)
        character.directionVector = GVector.normalize(directionVector)
}
```

4.4 GPS

Sur Android la géolocalisations ce fait bien évidemment au moyen de Google Maps. Il faut d'abord configuré un compte et obtenir une clée à copié coller dans notre application.

Ensuite il faut dans le manifeste pas oublié dire qu'on a besoin de la permission pour la carte et la demander ensuite en kotlin.

On ajoute un listener pour que notre position s'affiche de manière asynchrone

```
  val mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
     mFusedLocationClient.lastLocation
             .addOnSuccessListener { location ->
                 ...
             }
```

5 iOS

L'idée de faire un moteur de jeu pour Android, c'est avant tout pour que le code soit recyclable le plus possible entre iOS et Android. (Syntaxe de language mise à part). Cette fois ci contrairement à l'ancient projet, nous avons commencé par Android, ce qui nous a permis une fois sur iOS d'utiliser le plein potentiel de de SpriteKit. (enfin... A quelque exception près).

La majeur différence entre les deux platformes est d'abord, la différence entre les origines de repère. Android se trouve sur le bord supérieur gauche tandisque celui de SpriteKit il se trouve au centre de l'écran. Mais après quelque ajustement il se trouve sur le bord inférieur gauche.

SpriteKit est un outil très complet que fourni Apple. Il est permet de faire des collisions de supprimer des objets à la volé sur l'écran et de modifié la z-Position.

Il n'est pas rare que grace à la conception simpliste du moteur "Graphics" certaine partie du code ont été plus simple à écrire sur Android que sur iOS. On peut prendre l'exemple des collisions. Avec SpriteKit il faut utiliser des masques de collisions avec un ID composer d'un certain nombre de bit et un masque de catégorie pour indiquer quel objet est autorisé à entrer en collision avec qui. Cette méthode avec SpriteKit est assez compliqué à mettre en place et au final, c'est la méthode des interfaces (protocol en swift) utilisé pour Android qui a été choisi. (voir 4.1)

Pour le GPS là aussi Apple fourni ses propres services, avec Apple Maps. Très simple d'utilisation, on a la aussi une fonction qui nous notifie quand été localisé.

```
manager.delegate = self
manager.desiredAccuracy = kCLLocationAccuracyBest
manager.requestWhenInUseAuthorization()
manager.startUpdatingLocation()
```

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations
locations: [CLLocation]) { ... }
```

6 Camera

6.1 android

L'originalité du jeu est que l'on peut changer le jeu en fonction de ces envies. Pour l'instant on change seulement les décors, mais on peut s'imaginé qu'avec le même concepte, on puisse changer le cours du jeu entièrement ou avoir des avantages facent aux ennemies.

Tout ceci est rendu possible par la prise en charge de la caméra qui va scanner des QrCodes et les interprétés.

Sur android on utilise une API tiers (ZXingScannerView) pour l'analyse.

On vérifie d'abord les permissions,

```
scannerView = ZXingScannerView(this)
setContentView(scannerView)
ActivityCompat.requestPermissions(this, arrayOf(CAMERA), REQUEST_CAMERA)

override fun handleResult(result: Result) {
    ...
}
```

Dans un premier temps on initialise le scanner de l'API, puis on lui donne le vue sur laquel elle doit afficher le preview de la caméra. On n'oublie pas de demander la permission pour utiliser la caméra. Si la permission est accordé tout fonctionne normalement sinon on n'affiche rien. Ensuite si l'API détecte un QrCode il va nous faire parvenir le résultat grace à la fonction "handleResult(result : Result)"

6.2 iOS

Chez Apple tout est déjà intégrer, ils mettent à notre disposition avec l'API qui gère la caméra, une fonction pour décrypter les QRCode.

```
import UIKit
import AVFoundation

class CameraController: UIViewController, AVCaptureMetadataOutputObjectsDelegate {
...
var video = AVCaptureVideoPreviewLayer()
var session: AVCaptureSession!

session = AVCaptureSession()

//Define capture devcie
let captureDevice = AVCaptureDevice.default(for: AVMediaType.video)
do
{
```

```
    let input = try AVCaptureDeviceInput(device: captureDevice!)
    session.addInput(input)
}
catch
{
    print ("ERROR")
}

let output = AVCaptureMetadataOutput()
session.addOutput(output)

output.setMetadataObjectsDelegate(self, queue: DispatchQueue.main)

output.metadataObjectTypes = [AVMetadataObject.ObjectType.qr]
}
```

Malgrès que Apple fournie tout les composants pour décrypter les QrCode, cela n'en est pas plus simple. Le concepte étant d'ouvrir une session "AVCaptureSession" et d'essayer de capture des donnée (en fournissant le type de donnée) sur un périphérique (AVCaptureDevice).

Dès qu'un QRcode est détecté on est notifié grâce à la fonction

```
func metadataOutput(_ output: AVCaptureMetadataOutput,
                    didOutput metadataObjects: [AVMetadataObject],
                    from connection: AVCaptureConnection)
{ ... }
```

7 Conclusion

Ce qu'il faut retenir de tous ceci c'est que le moteur de jeu "Graphics" a été mis à jour est gère les collisions et les sortis d'écran de manière personnalisé pour chaque item du jeu, ce qui fait gagné en performance. "Graphics" prend en charge toutjours de manière très simple l'accéléromètre et ceux à partir de n'importe ou dans le code pas besoin de faire le lien avec une activitée, tous est automatiquement configuré. Le moteur de jeu devient de plus en plus complet et performant, et commence à démontré la robustesse de son architecture en ayant été la base de déjà deux jeu.

L'originalité du jeu est le fait qu'on puissent scannée des QrCodes et changer le jeu lieu même en fonction de ce que l'on scanne. On peut aussi combiné les ajouts les changements. Il faut s'imaginer ce même principe pour une jeu beaucoup plus grand et complet et que le joueur puisse changer le cours de l'histoire jeu et avoir des avantages sur ces adversaires en scannant des QrCodes.

Du coté d'Apple, tout le code à été recycler depuis celui d'android, même les collsions ont été faite maison. La simplicité chez Apple a été surtout remarquable au niveau de la géolocalisation et des demande de permission.