Crash it - Developpement mobile 2 (iOS & Android)

Olivier PICARD, L3 informatique

27 avril 2018

Résumé

Réalisation d'un jeu de vaisseau spatial sur deux platformes en programmation native.
La version iOS[2] est programmé en Swift et celle android[1] programmé en Java.

1 Introduction

Crash it est un jeu développé dans le cadre d'un projet en développement mobile 2, conçu en programmation native sur iOS[2] et android[1].

— Pour iOS[2] le jeu est programmé en Swift et utilise le framework SpriteKit[4], ce qui facilite grandement la conception. Car plusieurs éléments de base sont fournis par Apple. Le but étant de faire des versions du jeu identique peu importe la platforme. Pour pouvoir recycler le maximum de code entre ces platformes, la stratégie choisie a été d'utiliser le minimun d'outil proposer par Apple et de concevoir soit même les outils afin de pouvoir les recrée à l'identique sur android[1] et obtenir les même comportement sur les deux platforme

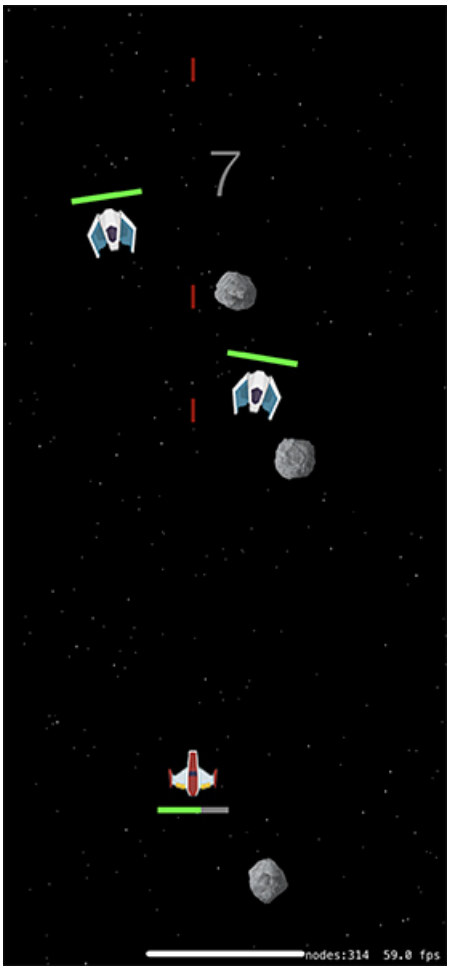
— Pour android[1], la conception à été beaucoup plus laborieuse, car c'est tout l'inverse de iOS[2] avec SpriteKit[4], rien n'est fournis pour concevoir un jeu. Que ça soit la boucle infinie, ou les notion de scène avec les liaison parent/enfant.
Pour android[1] il a fallut allez beaucoup plus loin pour pouvoir garder un code similaire à swift, cà savoir concevoir un moteur de jeu(sans la partie physique) pour immiter SpriteKit[4] en version alléger. J'insisterais beaucoup dans ce rapport dans la partie android[1] sur ce moteur de jeu car c'est un élément complexe, beaucoup plus complexe que le jeu lui même. Ce moteur s'appelle "Graphics" il se présente sous forme de package, et il a pour but de masquer aux développeurs de jeu les éléments complexe comme la gestions des threads, la boucle de jeu infini, la gestion des dessin dans le bon ordre à l'écran, le positionnement des élémentss relatif(par rapport à ses parents), la gestion des noeuds...

2 Contrainte

première contrainte faire en sorte que les ennemies soit visible au loin pour que le joueur ne soit pas trop surpris de se faire attaqer. Pour cela il a fallut utiliser une pratique assez courant, vérouiller le périphérique en mode portrait.

deuxième contrainte Pour reprendre une partie. On peut sauvegarder la partie et la reprendre exactement la ou l'on était. Le score sera le même, les vaisseau ennemie seront de la même couleur avec les même caractéristiques et suivront la même direction qu'au moment de la sauvegarde. **Reprendre une sauvegarde sans issue.** Il se peut que lors d'une sauvegarde on puissent être en cerclé par les ennemies, il devient donc impossible d'augmenter sont score lors de la reprise. Pour pailer à ce problème, il y a un mode d'invincibilité qui s'active pendant quelque secondes pour quelque soit la situation vous pourrez continuer votre partie serènement.

troisième contrainte concernant le déplacement du joueur, une idée original est de d'utilisé l'accéléromètre du périphérique. Mais un problème survient car quand on penche le périphérique on ne voit plus(ou moins bien) l'écran ; Une autre idée plus basique est que quand on bouge sont doight sur l'écran, le vaisseau suit le déplacement du doight, mais bizarrement en esayant cette technique il est preque qu'impossible d'obtenir un score décent et en plus ça fatigue le doight à mesure que l'on fasse glisser sur l'écran. Donc l'option choisit est de déplacé le vaisseau en cliquant sur la droite ou la gauche de l'écran. Tout simplement.



3 Description générale de l'application

3.1 Fond étoilé

Le jeu se base sur des éléments de coneption simple. On a un fond étoilé, qui est en réalité si l'on regarde bien deux couches d'étoiles générées aléatoirement. Une éloigné moins brillante, et plus nombreuses, et l'autre qui donne l'apparence d'être plus proche par le fait quel se déplace plus vite et quels sont plus brillant(moins transparente). Tout ceci pour but de créer un effet de profondeur plus réaliste, enfin... c'est l'effet souhaité. Mais après au visuel deux couches d'étoiles sont largement mieux qu'une.
Par contre il me semble important de notifier que pour faire ce fond il faut 200 étoiles environs(quantité adaptable en fonction de la taille de l'écran) ce qui peut être gourmand en terme de ressource même si ce ne sont que des carré(pas d'image). Il serait intérresant pour une question d'optimisation de créer des images d'étoiles et de déplacer ces images au lieu de déplacer les 200 étoiles.

3.2 Ennemies

Les ennemies sont générés aléatoirement avec des couleurs différentes et des caractéristiques différentes. Parce que tous les ennemies ne se valent pas. Pendant que d'autre sont lent à tirer d'autre sont beaucoup plus rapide et mais on beacoup moin de vie. Les caratéristiques en fonction des couleurs :

- Les vaisseaux bleus sont en règle général plus résistant que les autres, ils ne tirent pas très vite et ils n'infligent pas beaucoup de dégât.
- Les vaisseaux vert sont pas très résistant, ils infligent beaucoup de dégâts, mais ils ne tirent pas rapidement
- Les vaiseeau orange ont une résitance faible, une attaque moyenne, mais ils tirent très vite

Les ennemies pointent(se tournent) toujours vers leurs cibles peut importe ou il se trouve. Pour cela les ennemies utilisent un algorithme qui se base sur la trigonométrie[5]. On aurait put appelé cette fonction la fonction lookAt(Point p).

3.3 Items annimés

Si on regarde attentivement le jeu, on peut remarquer que certain item sont fixe(déplacement seulement de haut/bas et gauche/droite sur l'écran) et y en a d'autre qui sont annimé à la manière des GIFs. Si les élements ont été intégrer dans le décors avec succès cela ne se remarque peut-être, semble logique on ne se pose pas la question. Mais quand on s'interresse aux astéroides par exemple on peut remarqué qu'en plus de parcourir l'écran de haut en bas les astéroides tournent sur eux même. Le but recherché est de donner le sentiment d'appesenteur, de flottement. Pour les explosions aussi on remarquera qu'en plus de se déplacer de haut en bas les explosions sont annimés elles donnnes vraiment le sentiment d'explosion. Ce sont le même principe que pour les GIF, c'est une succession d'image qui va être lu à un interval donnée, durant un nombre de fois donné.

4 Architecture du code

Malgrés l'énorme différence entre les deux platformes l'architecture et le code reste très similaire. Comme le code reste le même(syntaxe du language oblige une différence) dans la partie iOS[2] nous parleront du jeu lui même et des mécanismes utilisés pour construire, placé et faire intéragir ses éléments. Dans la partie android[1] on accordera beaucoup plus d'importance à l'aspect très technique, à savoir la création du moteur de jeu "Graphics" créer pour l'occasion et qui occupe une place central dans la version Android[1]. Mais je le re-dis de nouveau, les codes sont casiment identique pour ne pas dire totalement identique. L'intégralité des sujets abordé dans la partie iOS[2] (même les élément faisant référence à SpriteKit[4]) est aussi valable pour android[1], parfois au travers de "Graphics".

4.1 iOS

4.1.1 Polymorphisme

Mise à jour des éléments Dans l'ensemble du projet le polymorphisme est utilisé massivement. Car cette notion est au centre même de l'architecture. Comme tous les éléments de la scène sont item en mouvement alors on à une classe mère "MovingItem" qui hérite d'un Sprite (est une image si l'on donne un chemin vers un fichier ou sinon correspond à un rectangle de couleur dessiner sur la scène). MovingItem représente donc un sprite dont la position et autres attributs peuvent être mise à jour régulièrement a travers la méthode update(). Que ça soit les vaisseau, astéroids, étoiles, ou explosion elles héritent toute de la classe "MovingItem", et contiennent toutes par conséquent une méthode update().

```
for item in self.children {
    if let movingItem = item as? MovingItem {
        movingItem.update(currentTime)
        ...
    }
    ...
}
```

Lorsque l'on ajoute un élément à la scène l'élément s'ajoutent dans un tableau children contenu dans la scène, ce tableau est un tableau de CGNodes (CGNodes, classe mère la plus haute dans la hierarchie fournie par SpriteKit[4]). Pour que la scène puissent mettre à jour tous les éléments visibles, alors ont parcours le tableau children de la scène, on regarde si l'élément en cours est du type "MovingItem" et si c'est le cas on appelle la méthode update(). Ainsi grâce à la surcharge de la méthode update() un vaisseau ennemie aura un comportement différent d'une étoiles par exemple.

Collisions

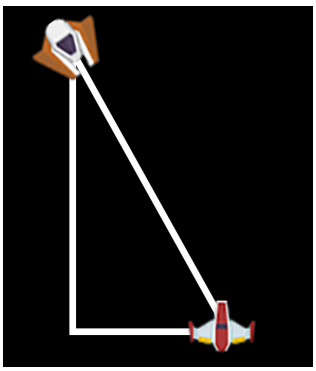
```
for item in self.children {
    ...
    else if let collisionableItem = item as? Collisionable {
        let overlapsedItems = overlapsListItems(collisionableItem)
        for ovItem in overlapsedItems {
```

```
                collisionableItem.inCollisionWith(item: ovItem)
                ovItem.inCollisionWith(item: collisionableItem)
            }
        }
    }
```

Pour la détection des collisions on implémente le protocole(interface) Collisionable(ICollisionable) aux classes dont le sprite peuvent entrer en collsion avec un autre sprite. Dans la scène on parcourt le tableau "children" on vérifie si l'élément actuel est du type "Collisionable" si c'est le cas on appelle la méthode "overlapsListItems(_ item : Collisionable)" pour vérifier si l'élement chevauche un autre. Si c'est le cas on applique la collision aux deux objets.

4.2 Orientation des vaisseau ennemies

Chaque ennemie à une cible(en l'occurence le joueur) et est capable de s'orienter vers lui pour ouvir le feu sur elle. L'orientation des vaisseaux ennemie se fait par le biais d'un algorithme fait maison basé sur la trigonométrie[5].



Il faut imaginer un triangle rectangle qui se forme entre l'ennemie et le joueur. Pour que l'ennemie regarde en direction du joueur il faudra donc pouvoir déduire l'angle au sommet du triangle(ou se situe l'ennemie)

```
func updateDirectionToTarget() {
    // Soit un triangle ABC rectangle en A
    let len_BA = self.position.y - target.position.y
    let len_CA = target.position.x - self.position.x

    let angleRadian = atan(len_CA/len_BA)
    self.zRotation = angleRadian
    self.direction = CGVector(dx: sin(angleRadian), dy: -cos(angleRadian))
}
```

Ce qui peut se traduire mathématiquement par :

$$\boxed{\begin{array}{l} BA = yPosition_{ennemie} - yPosition_{joueur} \\ CA = xPosition_{joueur} - xPosition_{ennemie} \\ \alpha_{radian} = \arctan(\frac{CA}{BA}) \\ \vec{v}_{direction} = (\sin(\alpha_{radian}), -\cos(\alpha_{radian})) \end{array}} \quad (1)$$

4.2.1 Générateur

Le jeu est constitué d'une multitide de générateur aléatoires, par exemple pour les étoiles, les vaisseaux ennemies et les astéroides. Les générateurs constistuent à la fois une interface facilement paramétable par la scène et un allègement du code.

Génération d'étoiles Le génération d'étoiles se fait de manière aléatoire, mais aussi en prenant en considération la vitesse de défillement des étoiles. Car plus les étoiles défile vite, moins elle seront nombreuses sur l'écran, donc plus il faudra en généré d'avantage.

```
if(arc4random_uniform(101) > stars_percent*Int(MovingItem.base_moving_speed))
{ return }
```

Au démarrage le générateur va remplir l'écran d'étoile pour ne pas commencer sur un écran noir, et ensuite les étoiles vont être généré aléatoirement

Génération des vaisseau ennemies Ce générateur (*ShuttleEnemiesGenerator*) ce distingue des autres par le faits qu'il doit généré des vaisseau en prenant en compte leurs couleurs et l'atribution des statistiques correspondantes au type de vaisseau(attaque, défense...)

4.3 android

Dans cette partie nous traiterons de l'architecture sous android[1], notamment du moteur de jeu "Graphics". Cette partie ce distingue de la précédente car les éléments traité ici sont disponible uniquement pour la version anroid.
Il faut savoir qu'android[1] ne met pas d'outils préfabriqués à la disposition des développeurs pour la création de jeu vidéo. Afin de pouvoir représenté un item à l'écran il existe plusieur moyen :

- L'utilisation des ImageView, qui est une classe de haut niveau, facile d'utilisation, qui ne nécessite pas l'utilsation d'une scène, ni de boucle infini, ni de thread séparer. Mais l'inconvénient est que les ImageView comme leur nom l'indique sont basé sur des images, et lorsque leurs nombres devient important, les performances diminuent.

- Il y a l'utilsation des Canvas, qui permettent de dessiner directement sur une vue (View ou SurfaceView). L'utilisation de Canvas est beaucoup plus bas niveau que les ImagesView avec des performances assez correcte sous haute charge et avec un code optimisé. Il revient au développeur de créer ça scène sur laquel dessiner, de gérer des Thread différents, de gérer la manière dont s'affiche les objets. Cette méthode bien que

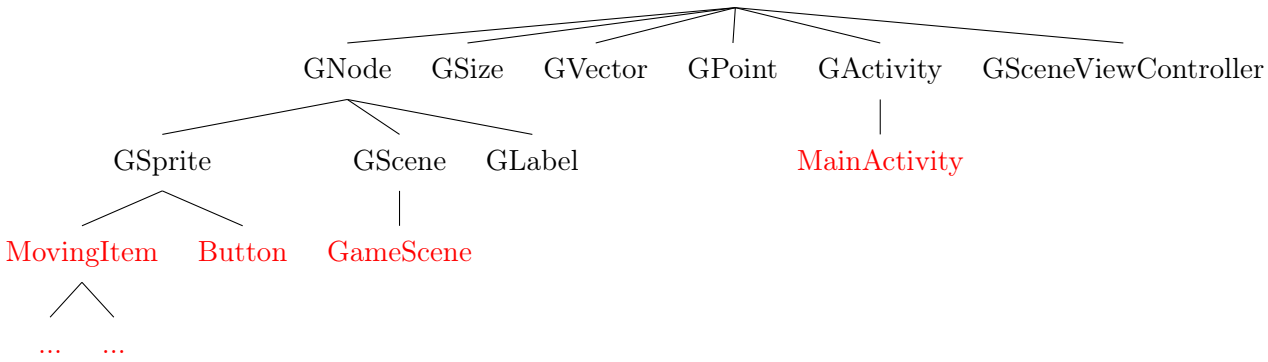
contraignante donne plus de liberter à la création.

— Enfin vient l'utilisation d'OpenGL et Vulkan[6](API graphics dernière génération équivalent de Metal chez Apple). Les performances sont très bonne même lorsque la scène est chargé. Donne une liberté total sur les possibilité de création, mais reste plus compliqué à mettre en place.

Les ImageView n'offrant pas une possibilité de création suffisante et des performances insuffisante, elles n'ont pas été retenu. L'utilisation d'OpenGL[3] ou Vulkan[6] est un bon choix mais étant limité par le temps, la mise en place et le temps d'apprentissage me semble trop longue. Donc le choix c'est porté sur l'utilisation des Canvas. Afin de réutilisé le même code que pour Swift et le traduire, casiment à l'identique en Java, l'idée à été de rajouter à Android[1] toutes les fonctionalitées utilisé par le jeu, dans SpriteKit[4] et de masquer à l'utilisateur de *Graphics* toute les rouages comme la gestion des threads, la suppression et l'ajout de noeuds en différé, la gestion des noeuds enfant/parents, même une activity(GActivity) et un viewController(GSceneViewController) devient disponible afin de surcharger automatiquement les méthodes android comme **touchDown** ou **touchUp** et de transmettre en toute transparence à la scène... Et delà est Naquis "Graphics".

4.3.1 Graphics : Le moteur de jeu

Architecture



Les éléments en noirs représente le package *Graphics*(Moteur de jeu) et ceux en rouge sont les éléments propre au jeu (crash it version android[1]) commune à la version iOS[2].

Description Graphics est une architecture logiciel conçut pour supporter n'importe quel jeu ne nécessitant pas de physique. Il a pour but de ressembler à une version allégé de SpriteKit[4] d'Apple. La convention de nommage étant de rajouter un "G" devant le nom des classes, ça a donné des noms quelque peu innatendu, en particulier sur la classe Point qui devient GPoint...(le sens de la traduction en français est intéressant...) Il est composé de plusieurs classes basiques à savoir, *GPoint*(représente un point dans l'espace), *GVector*(Représente un vecteur et effectue des opération de base), *GSize*, *GInterval*(Représente une intervale [min, max] ou l'on peut généré un nombre aléatoire compris dans ce domaine de définition) et bien évidemment *GScene*(représente la scène ou se trouve tous les objets à dessiner)

```
public void run() {
```

```
        didInitialized();
        while(this.enable) {
            refreshSceneNodes();
            processTouch();
            update(System.currentTimeMillis());
            ...
                canvas.drawColor(backgroundColor, PorterDuff.Mode.CLEAR);
                render(canvas);
            ...
             Thread.sleep(16);
        }
    }
```

Exécution du thread loop lors de la création de la scène(GScene) on initialise tout les composants de la scène et on passe dans la boucle infinie, avec une pause de 16 millisecondes => 1/60 (60 images/sec). On récupère aussi le canvas qui va nous servir pour dessiner les éléments. Dans l'ordre les étapes de rendu sont :

- l'ajout ou la suppression des noeud de la scène
- La vérification d'un éventuel click sur l'écran
- La mise à jour des éléments de la scène, on y effectue toutes la logique du jeu
- On rend(dessine) tous les élements de la scène.

Update Cette fonction permet de mettre à jour les éléments de la scène. Par exemple de changer la position, la taille, la couleur ou de créer ou de supprimer un élément de la scène. C'est dans cette partie qu'on va définir le comportement même du jeu.

L'ajout et/ou suppression différé Cette fonctionnalité est totalement transparente pour l'utilisateur final, il n'a jamais accès à cette méthode. Elle permet d'éviter les conflits d'accès à une ressource(en l'occurence ici la liste des enfants de la scène). Car lors du update, on parcours la liste d'enfant de la scène et on décide d'en supprimer un, le programme s'arrête avec une *ConcurrentException*. Pour pallier à ce problème quand l'utilisateur fait addChild() ou removeChild() rien n'est ajouté directement, on va attendre la fin de toutes les opérations avec de modifier cette liste.

Transmission inter-thread de la détection du clic Il faut savoir que la détection des clics se fait dans *GActivity* sur le thread principal(main). Ensuite *GActivity* transmet cette information au controller *GSceneViewController* toujour sur le thread principal, et c'est ici que les choses se compliquent car il faut relayer les informations à la scène qui se trouve sur le thread *GameLoop*. La solution de base a été d'appeler directement *touchDown(...)* et *touchUp(...)* de la classe *GScene*. Mais forcer de constater qu'ensuite une partie des fonction de GScene se désynchronise car elle ne sont plus exécuter sur le thread *GameLoop* mais dans la Thread *main*(principale). La correction apporté à été de stocker les informations dans une variable de *GScene* et de créer une fonction qui vérifira les changements à chaque passage de la boucle infinie. **Tous ce procédés est bien évidement masqué à l'utilisateur final, il verra seulement ça fonction *touchDown()* et *touchUp()* être appelé.**

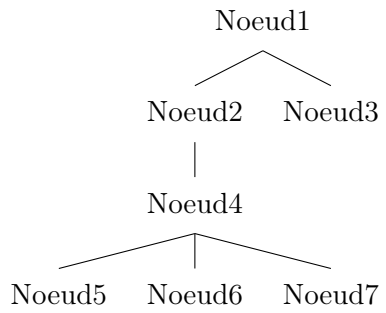
Processus de rendu Pour mener à bien ça tâche, et surtout afficher les élément aux bonne endroit et dans le bonne ordre, la partie rendu est elle même composer de plusieurs étapes à savoir dans l'ordre :

- Le regroupement de tous les élements de la scène par leur position en Z. Répond à la question : *'est ce qui est dessiner au dessus de quoi ?*
- Une fois regrouper on tri la liste liste obtenue, pour ranger les éléments du plus éloigné aux plus proche
- Une fois ce travail préalable effectué, la scène(*GScene*) délègue la suite du processus de rendu à chaque élément implémentant l'interface *IGDrawable*(représente des éléments pouvant être dessiner sur la scène).
- Les éléments *IGDrawable* ont pour mission d'être correctement dessiné sur la scène. Pour cela on va calculé la relativité des items par rapport à leur parent. Prenons un exemple : On a un vaisseau qui a pour enfant une barre de vie, qui elle même pour enfant un fond gris. On souhaite que quand le développeur du jeu déplace le vaisseau, tous ses enfants le suivent. On a pas besoin de déplacer chacun d'eux manuellement. De même pour les rotations et les positions en Z.
- Uniquement quand la position relative d'un élément par rapport à tous ces parents est calculés, là on dessiné cette éléments à l'écran.

taille d'écran unique Si on regarde attentivement le code du jeu, toutes les valeurs sont codée en dur(à pars quelque une). Pourtant si l'on teste sur plusieurs écran en mode portrait tous s'ajuste parfaitement. Le procédé qui se cache derrière permet de coder en toute simplicité sans se soucier du chargement de taille d'écran, mais augmente grandement les performances. Le principe est qu'au lieu d'utiliser tout les pixels de l'écran on en utilise qu'une partie et on zoom dessus afin que ça ai l'air naturel. On doit d'abord définir une résolution (w, h) qui correspond le plus à l'aspect ratio(16 :9, 4 :3) du périphérique et on dit au canvas de n'utiliser que la partie converné de l'écran. Avec ce procéde le jeu Crash it est passé de 10 image/sec à 25 images/sec sur android[1] lollipop, et jusuq'à 35 image/sec sur android[1] oréo.

5 Quelques points délicats/intéressants

Les points délicats on été sous android[1] dans le moteur "*Graphics*" Le point le plus difficile à aborder à été la prise en charge d'un arbre de rendu. Supposons l'arbre ci-dessous, qui représente la relation parent/enfant qui peut être créé par dans un jeu



Imaginons que l'on veuille déplacer le Noeud1 et l'on veut aussi qu'il fasse une rotation. Suposont que le noeud1 représente un vaisseau ennemie et qu'il doit viser (rotation) et se

diriger(changement de position) vers ça cible(en l'occurence vous). Le noeud2 peux représenté l'élément vert d'une barre de vie, et le noeud4 le fond de cette même barre de vie(Il s'agit la que d'un exemple, pour rammener un cas abstrait vers un cas réel)

Quand ce vaisseau(Noeud1) fera le moindre déplacement ou rotation, l'ensemble de ses enfants doit pouvoir le suivre. Donc la position(relative) du noeud2,3,4,5,6,7 devra être transformé en position absolue par un convertisseur(*GRelativeRender* prenant en compte les paramètres de chacun de leur parent respectif). Il a donc fallut créer une fonction récussive pour que la classe *GRelativeRender* puissent appliqué toutes les transformations

```
public void processChildRelativity(@NonNull GNode current) {...}
```

6 Conclusion

Ce projet a été une belle expérience. Le rapport n'a pas été très orienté vers iOS[2] car tous été relativement simple a créé car ils donnent tous les outils nécessaire au fonctionnement optimal d'un jeu. Seul quelques étapes comme le ciblage du joueur, créer une architecture modulaire, assez générique et suffisament abstraite pour facilité la réutilisation du code ont été plus complexe à mettre en place.

Pour ce qui est d'android[1], rien n'est founis. La solution la plus efficace pour réutilisé tout le code swift a été de créer un moteur de jeu qui aurait quelques fonctionnalitées de base de SpriteKit[4]. Faire 2 jeux vidéo + un moteur de jeu en si peu de temps a été un véritable défi. Mais heureux de dire qu'il a été accompli.

Références

[1] Android developers. https://developer.android.com.

[2] ios developer. https://developer.apple.com.

[3] Android opengl documentation. https://developer.android.com/guide/topics/graphics/opengl.html.

[4] Spritekit official. https://developer.apple.com/spritekit/.

[5] Cours trigonométrie. http://www.educastream.com/formules-trigonometriques-calcul-angles-3eme.

[6] Android vulkan doc. https://developer.android.com/ndk/guides/graphics/index.html.