Rapport de Projet

Jeu 2D pour mobile
Heliko

Auteurs
Thibaut Castanié
Jolan Konig
Noé Le Philippe
Stéphane Wouters

Encadrant
Mathieu Lafourcade
Master
IMAGINA

Année universitaire 2014-2015

Remerciements

Nous remercions tout particulièrement M. Lafourcade de nous avoir encadrés, soutenus et prodigué de bons conseils, tout le long de la réalisation de ce projet.

Nous tenons à remercier M. Bourreau, ainsi que les intervenants de son cours, de nous avoir aidés à y voir plus clair dans notre organisation du projet.

Nous remercions également toutes les personnes qui nous ont apporté des retours, ainsi que de précieux conseils sur nos différentes versions de tests du jeu tout au long de son développement.

Table des matières

Table des figures

Chapitre 1

Introduction

Le projet relève d'une idée partagée entre quatre étudiants : créer un jeu vidéo pour téléphone mobile, de sa conception jusqu'à sa publication. Le thème du jeu de rythme a été choisi et il est développé avec le moteur de jeu *Unity*. Nous nommerons ce jeu **Heliko** (escargot en Espéranto, l'escargot étant la mascotte du jeu).

La complexité du projet réside dans sa liberté. De nombreuses décisions doivent être faites pour le mener à bien et accomplir l'objectif principal : publier un produit final qui soit beau et pleinement fonctionnel. Ayant proposé notre propre sujet de TER, nous devons établir le cahier des charges initial, ainsi que concevoir en équipe le principe du jeu. Le temps est un facteur principal et difficile à gérer, puisque créer un jeu de toutes pièces en demande énormément. De plus, cela rassemble beaucoup de domaines (développement, graphisme, son, monétisation, publication...), qui nécessitent une formation lorsqu'ils ne sont pas maîtrisés.

Après avoir analysé le sujet, en parlant de l'existant et en donnant le cahier des charges, nous présenterons le rapport d'activité rendant compte des méthodes de travail que nous avons utilisées pour mener à bien ce projet. Le rapport technique décrit les choix de conception ainsi que les principaux éléments de ce qui constitue notre moteur de mini-jeux. On pourra également y trouver les outils que nous avons pu développer et des exemples concrets de l'utilisation de notre moteur. Nous conclurons en donnant les résultats, et en faisant un bilan du projet.

Glossaire

Quelques mots techniques sont utilisés dans ce rapport, il est nécessaire de comprendre le sens dans lequel ils seront utilisés.

Mobile Représente l'ensemble des supports des téléphones mobiles et des tablettes.

Feedback Rétroaction en français. Action en retour d'un effet sur sa propre cause. Utilisé dans le jeu de rythme comme un signal de prévention pour le joueur.

Gameplay Jouabilité en français. Terme caractérisant le ressenti du joueur quand il utilise un jeu vidéo.

Sample Échantillon en français. Extrait musical, ou son, réutilisé dans une nouvelle composition musicale, souvent joué en boucle.

Chapitre 2

Analyse du projet

Une bonne analyse est nécessaire pour le pilotage du projet. Il faut réaliser un examen des jeux de rythmes déjà existants, concevoir le concept du jeu, définir les priorités et établir une planification avant de commencer tout développement.

2.1 Analyse de l'existant

Plusieurs jeux au gameplay approchant celui visé par l'équipe ont été testés. Voici ce qu'il en ressort au niveau du gameplay.

Jeux basés sur la musique On y trouve souvent une musique par niveau. Il faut taper sur plusieurs notes en synchronisation avec la musique qui est jouée en arrière-plan. Les notes ont généralement été posées manuellement par les concepteurs, en fonction du niveau. Certains jeux permettent de charger directement ses propres MP3, la génération du niveau est donc faite à la volée en fonction du fichier.
Exemples : Guitar Hero sur console, TapTap sur iOS et applications de promotions d'album.

Jeux basés sur l'habileté rythmique Il existe aussi des jeux basés sur l'habileté de l'utilisateur et sa vitesse de réaction où le principe est de taper sur l'écran au bon moment et au bon endroit. L'accent est mis sur le style graphique, il n'y a pas une grande variété dans les musiques et elles n'ont pas de lien direct avec les niveaux. La musique n'est là que pour apporter du rythme ainsi qu'un élément addictif, si elle est agréable à l'oreille.
Exemples : Geometry Dash, SineWave et Colorace sur smartphones.

Jeu de type "Runner" Enfin, certains jeux ne proposent qu'un gameplay rapide et de beaux graphismes, en ne prêtant que peu d'importance à la musique. Le contenu du jeu est ainsi plus fourni en niveaux, en personnages et fonctionnalités, et les graphismes du jeu sont donc, en conséquence, plus approfondis. L'utilisateur ne joue qu'en fonction de ce qui apparaît à l'écran. La musique peut changer en fonction de l'avancée dans le niveau.
Exemples : Temple Run sur mobile. Bit.Trip sur Wii et PC.

2.1.1 Principes dégagés

Le principe des jeux musicaux nous plait, mais nous ne pouvions pas nous permettre d'utiliser des catalogues de musique existants pour des questions de droits d'auteur. Générer les niveaux de façon procédurale par rapport à de la musique nous semblait intéressant, mais nous avons préféré nous concentrer sur d'autres priorités, alors que créer de nombreux

niveaux à la main aurait été une tâche fastidieuse que nous voulions éviter. Créer un jeu de type *Runner* ne demande que très peu de travail sur le contenu (il y a beaucoup de générations aléatoires), mais le principe du jeu, trop simple, nous semblait peu intéressant sur le plan de l'enseignement. Nous avons ainsi décidé de créer un moteur de mini-jeux rythmique générique, afin de créer des scènes complexes et difficiles qui demandent un apprentissage de la part de l'utilisateur. Ceci nous assure une durée de vie de jeu correcte, sans devoir créer trop de contenu.

2.2 Objectifs du projet

L'objectif premier du projet est de mettre en ligne un jeu de rythme pour téléphone mobile sur les plateformes Google Play et Apple Store. Le jeu doit être le plus intéressant et le plus complet possible.

Contenu du jeu Nous avons dès le départ conçu les parties du jeu pour qu'il soit jouable et en ligne au bout de 4 mois de travail. Étant donné notre légère expérience dans la gestion du temps, nous avons conçu des "couches" à développer pour former le jeu de façon incrémentale et ainsi être certain d'aboutir à un résultat (à condition de prendre en compte les tests, les finitions et la mise en ligne).

Couches incrémentales La liste des fonctionnalités à réaliser en fonction de leur importance pour un jeu jouable :

1. Créer un moteur de jeu de rythme Unity générique et créer un mini jeu jouable sur Android ;
2. Rendre l'application jouable sur tous les supports mobiles ;
3. Créer d'autres mini jeux ;
4. Créer un moteur de tutoriel et les tutoriels associés ;
5. Concevoir et ajouter une monétisation (gain de pièces dans les niveaux, personnalisation du jeu) ;
6. Concevoir et ajouter un esprit communautaire (partage de l'avancement) ;
7. Créer un éditeur public de niveaux.

2.3 Cahier des charges

2.3.1 Principe de progression

Le joueur lance le jeu et découvre un monde totalement vide sans aucune animation ni aucun effet sonore. Un simple métronome qui frappe la mesure retentit, et la mascotte, un escargot, se déplace en rythme sur cette planète de façon circulaire. Le joueur doit débloquer des objets visuels et sonores dans des mini-jeux afin de rendre cette planète plus fournie et joyeuse.

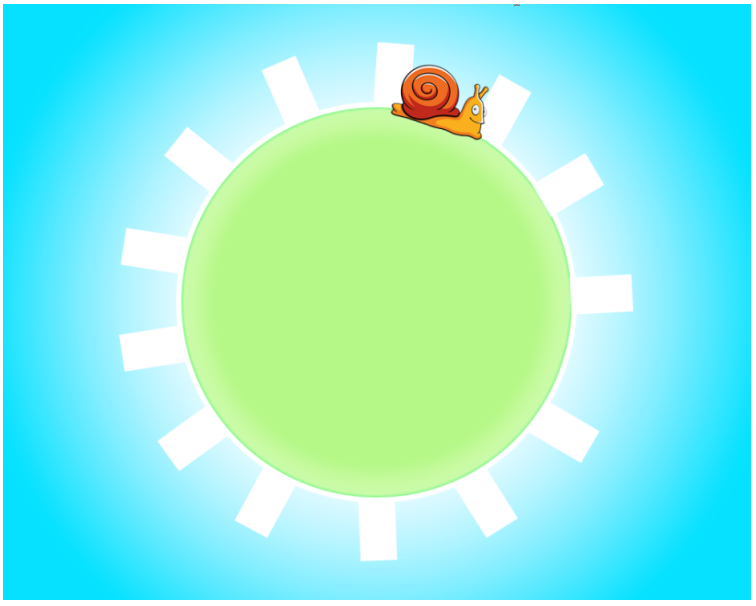


FIGURE 2.1 – Prototype de planète à remplir

Un objet est constitué d'un élément visuel qui est animé en rythme sur le tempo de la planète, et est associé à un sample unique qui est une composante d'une musique propre au jeu.

Ce sample peut être un kick, une basse, une mélodie... Le principe est de débloquer peu à peu chacun des éléments du morceau final. On compte entre 5 et 10 éléments (donc mini-jeux) pour achever le morceau.

Cette planète sert de représentation visuelle pour l'avancement du joueur.

Concept de mini-jeu

Un mini jeu est constitué d'une scène unique et minimaliste animée par une musique de fond entraînante.

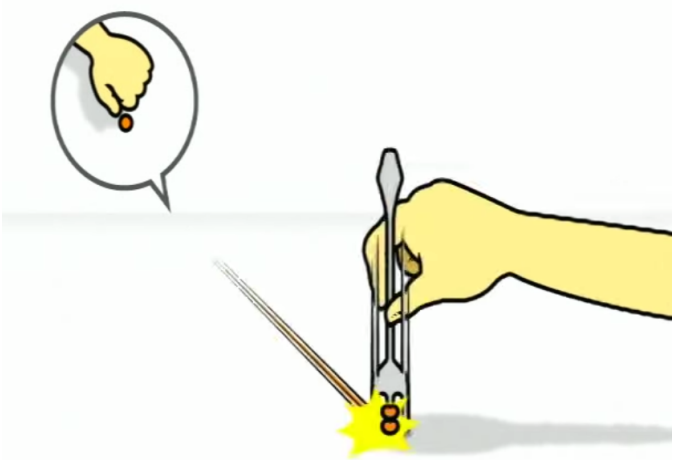


FIGURE 2.2 – Exemple de mini jeu, présent dans le jeu "Rhythm Paradize"

Sur la figure 2.2, le joueur doit attraper des petits pois en abaissant la fourchette au bon moment après un signal. C'est un bon exemple de mini jeu.

Le joueur doit apprendre quelques mouvements propres au mini jeu dans un entraînement. Une fois les mouvements acquis, le joueur peut tenter de réussir le mini jeu. La musique commence, et il doit réussir un maximum de mouvements dans le temps fixé (1mn30 - 2mn environ)

Ce mouvement peut être de nature varié : Il peut s'agir de tout simplement frapper un rythme synchronisé avec la musique, ou de répéter précisément une série de rythmes joués, ou de frapper après un signal avec un délai plus ou moins long, etc.

À la fin du niveau, un score est généré. S'il est supérieur à un certain seuil exigé, le mini-jeu passe en mode "réussi" et le joueur débloque l'objet associé sur sa planète. Il existe plusieurs issues pour un mini-jeu :

- Non réalisé
- Réussi
- Bien réalisé
- Parfait

Ces issues seront représentées par un nombre d'étoiles, allant d'une à trois.

Notions techniques Les mini jeux sont uniques, c'est-à-dire qu'à chaque fois il s'agit d'un nouveau concept rythmique. Il n'est pas possible de les générer à la volée. On mettra en place des mécaniques précises qu'on pourra réutiliser sur chacun des mini jeux pour les créer le plus facilement possible. Pour les graphismes, de très simples dessins en 2D seront utilisés.

2.3.2 Niveau infini

Une fois tous les éléments débloqués, le joueur atteint le niveau final qui est un mode infini arcade sur la musique assemblée dans lequel il peut essayer de réaliser le meilleur score possible en incarnant l'escargot. (Qui s'approche de notre idée de départ, mais de façon moins importante).

La somme de ses scores réalisés au fil du temps est convertie sous la forme d'une monnaie avec laquelle il peut acheter des effets de lumières et décorations supplémentaires à disposer sur sa planète afin de la personnaliser.

2.3.3 Partage

L'avancement des joueurs, et la personnalisation des planètes sont stockés dans le cloud. Un joueur peut montrer sa planète à ses amis et leur faire découvrir son avancement et ses scores.

2.3.4 Monétisation

En fonction de l'avancement final du projet, un système de monétisation sera choisi. Il pourra s'agir de la vente de monnaie supplémentaire pour la personnalisation par exemple,

ou simplement de mettre en place un système de publicités judicieusement mis en place. Cette monétisation s'accompagnera d'une étude des différents services de paiements proposés pour les téléphones mobiles.

Chapitre 3

Rapport d'activité

3.1 Organisation du travail

3.1.1 Rôles dans l'équipe

Vu la charge importante de travail et la diversité des tâches, nous avons préféré nous attribuer des responsabilités fixes :

- **Stéphane Wouters**, chef de projet ;
- **Noé Le Philippe**, responsable développement technique ;
- **Thibaut Castanié**, responsable son et graphismes ;
- **Jolan Konig**, responsable intégration et publication.

Nous avons fait le choix de travailler ensemble sur toutes les parties, et nous nous sommes affectés, au fil du projet, des micro tâches fréquemment mises à jour.

3.1.2 Cycles itératifs

Dès le départ, un développement par cycles itératifs a été choisi. Bien adapté pour le développement d'un jeu vidéo et surtout à cause de notre contrainte principale : notre liberté sur les choix.

Les premières semaines ont été dédiées à la réalisation de prototypes et de tests en augmentant toujours la difficulté, dans l'objectif de produire un moteur de jeu de rythme fonctionnel qui correspond aux besoins définis dans le cahier des charges.

Liste du déroulement de nos prototypes :

- **Prototype 1** - Réalisation d'un cube qui bat à un rythme constant (3 jours)
- **Prototype 2** - Réalisation d'un prototype de test de réussite (2 jours)
- **Prototype 3** - Création de la première version du moteur de rythme (6 jours)
- **Prototype 4** - Réalisation d'un prototype d'animations, connectés au moteur de rythme (4 jours) -> *Le moteur n'est pas assez précis et doit être revu*
- **Prototype 5** - Deuxième version du moteur de rythme (3 jours)
- **Prototype 6** - Nouvelle tentative de connexion à des animations (1 jour) -> *Test OK. On s'aperçoit que nous avons besoin de quart de temps dans le modèle et qu'il faut recommencer une nouvelle fois sa conception*
- **Prototype 7** - Troisième version du moteur de rythme (4 jours)
- Etc.

FIGURE 3.1 – Les différents projets Unity de tests

On voit sur la figure 3.1 l'ensemble de nos tests. À chaque nouveau test, un nouveau projet Unity. Au **test 14**, nous avons jugé que le moteur correspondait à nos attentes et nous avons ensuite itéré directement sur ce projet. C'est à ce moment (environ 1 mois après le début du projet) que le système de fonctionnement a changé et que nous avons fonctionné en micro tâches.

Les dernières semaines ont été dédiées aux finitions sur le projet afin de rendre le jeu agréable avec un aspect "fini".

3.2 Méthodes et outils

3.2.1 Méthode de communication

Gestionnaire de tâches

Pour faire avancer le projet, nous avons utilisé toutes les capacités d'un gestionnaire de tâches en ligne, **Trello**, qui est un outil inspiré par la *méthode Kanban*. Nous nous sommes imposé de visiter ce tableau tous les jours et ses outils de notifications nous ont permis d'être continuellement connectés.

Trello permet une gestion des tâches dans le Cloud avec de nombreuses fonctionnalités :
- Création de tâches, avec titre et description ;
- Choix d'une date de fin sur une tâche ;
- Affectation de membres à une tâche ;
- Labels (tags) personnalisés ;
- Commentaires sur chaque tâche pour discussion asynchrone entre les membres du groupe sur une tâche ;
- Application Android et iOS avec notifications par *push*.

Un système de rangement vertical a été adopté pour les types de tâches, et des labels de couleurs en fonction de l'avancement des tâches :
- **Bloquante** (tâche à réaliser rapidement, bloquant l'avancement du projet) ;
- **À discuter / en recherche** (tâche en cours de discussion, pour prise de décision) ;
- **À attribuer** (Tâche correctement spécifiée, en attente d'affectation à un membre de l'équipe ;

— **En réalisation** (tâche en cours de réalisation par un ou plusieurs membres de l'équipe) ;
— **À tester / à contrôler** (tâche réalisée, à tester pour confirmation) ;
— **Fait** (tâche réalisée et fonctionnelle, prête à être archivée).

Les tâches sont classées dans des colonnes "TODO" triées par thèmes (développement, graphismes...), ou par cycle itératif (d'un jour à une semaine).

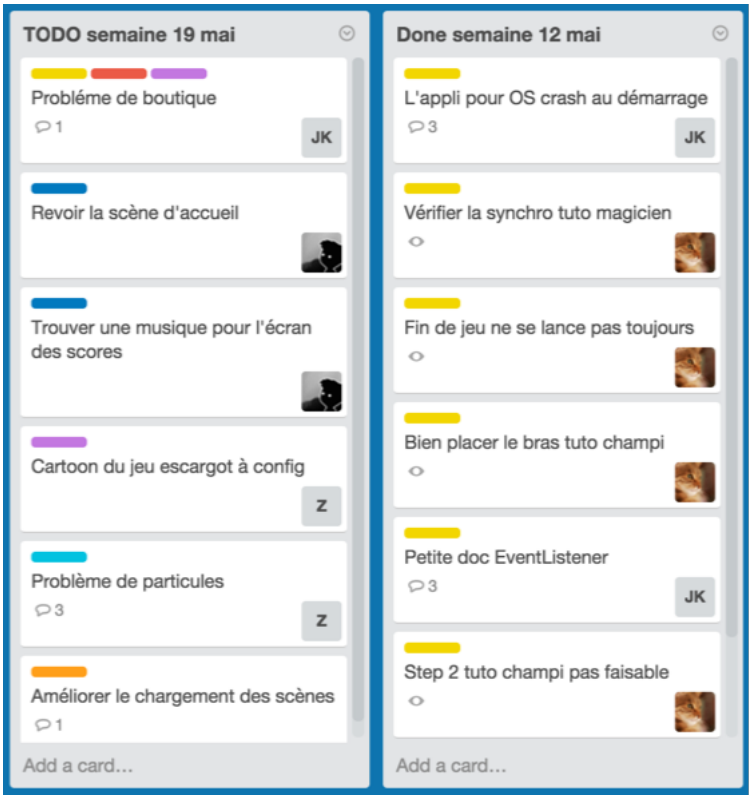


FIGURE 3.2 – Exemple de colonnes en fin de projet

Trello a aussi été utilisé comme mémo et pour archiver les ressources graphiques et sonores (figure 3.3).

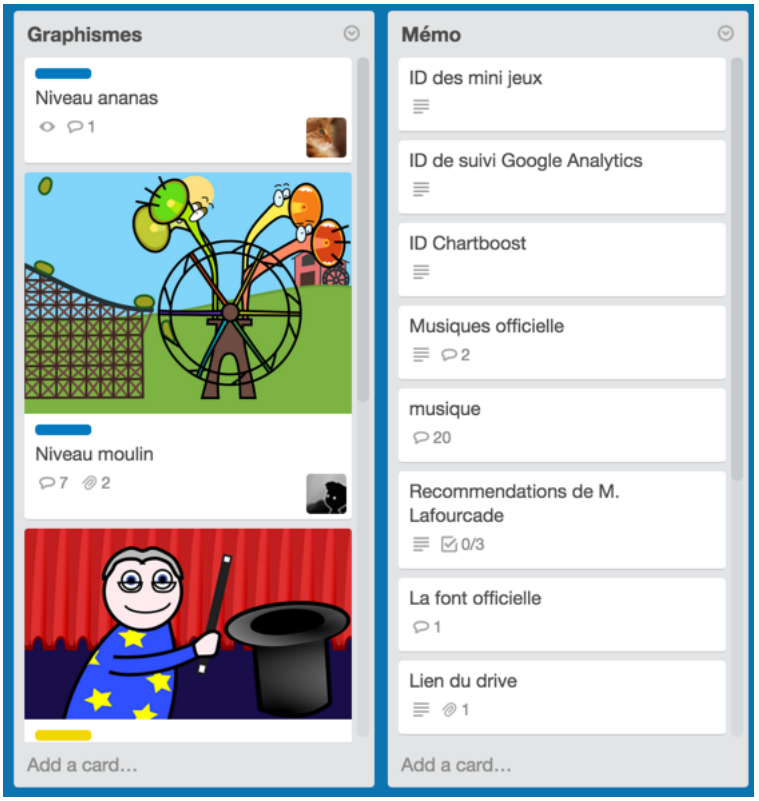


FIGURE 3.3 – Utilisation de Trello comme mémo et archives

Cette méthode de travail avec Trello nous a permis d'être efficace 100% du temps au travers d'Internet. Il n'y avait jamais de temps mort et nous étions toujours clairs dans notre direction tant que quelqu'un (chef de projet) s'occupait de créer des tâches et de les organiser.

Réunions

Même si *Trello* nous permet de travailler de façon indépendante, nous nous sommes réunis très régulièrement pour travailler ensemble et fixer les nouveaux objectifs, une à quatre fois par semaine.

Pendant les longues séances de travail en collaboration (de 9h à 18h par exemple), nous avons utilisé un véritable tableau blanc pour noter les tâches en cours et les affectations. Exemples de tâches :

- Réadapter la taille du logo *pause* sur Android v4.1 en écran 16 :10 ;
- Couper les 150ms du début du son `tic.wav` ;
- Revoir l'animation du bras gauche du champignon.

Les tâches étant à 80% de ce type (courtes et rapides), l'organisation est très importante pour être efficace.

De nombreuses heures de réunion étaient dédiées à la recherche de concept ou de remise en question des objectifs. Par exemple abandonner le développement d'un mini jeu trop complexe, ou en inventer de plus simples.

3.2.2 Github

Un gestionnaire de version pour notre projet a bien sûr été utilisé et nous avons choisi le gestionnaire *Github* qui offre de nombreux outils de statistiques très intéressants.

Déjà habitués à Git, nous l'avons utilisé pleinement et nous avons envoyé des *commits* pour chaque modification fonctionnelle. Nous sommes ainsi arrivés en fin de projet avec un cumul de **1020 commits**, globalement bien répartis entre nous quatre. (Avec plus de 2 millions de modifications dans les fichiers...).

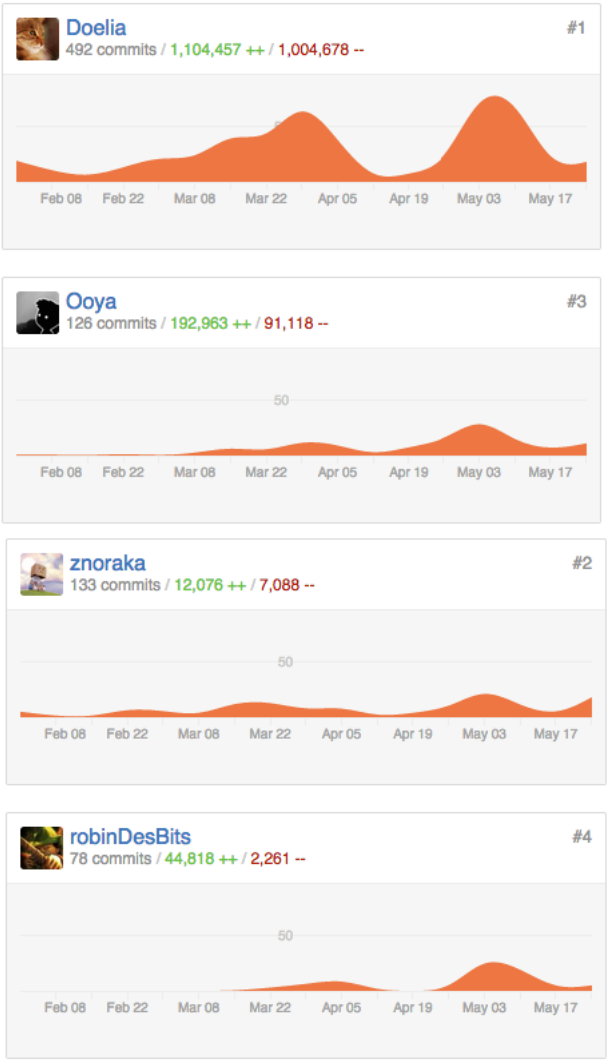


FIGURE 3.4 – Répartition des commits dans le temps

On remarque d'après la figure 3.4 de nombreuses suppressions (presque autant que d'additions), prouvant l'évolution du projet et l'application des cycles itératifs. Le développement s'est fait continuellement dans le temps.

Ponctualité hebdomadaire

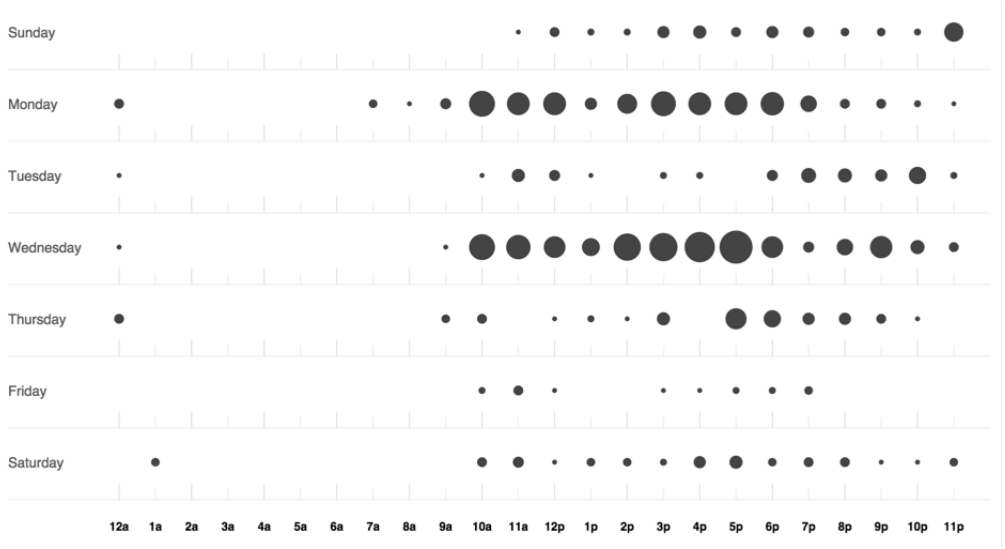


FIGURE 3.5 – Punch card de Github

D'après les statistiques *Github* (figure 3.5), nous avons travaillé tous les jours de la semaine, avec une préférence pour le lundi et le mercredi en après-midi et jusqu'en fin de soirée.

Modélisation Gource

Gource est une application qui permet de tracer en vidéo l'historique des commits qui construit l'architecture des fichiers d'un projet GIT.

Cette capture (figure 3.6) représente l'état final de l'architecture des fichiers du projet. La taille de la branche des tests démontre qu'ils ont effectivement constitué une grande partie du projet (Environs 30%).

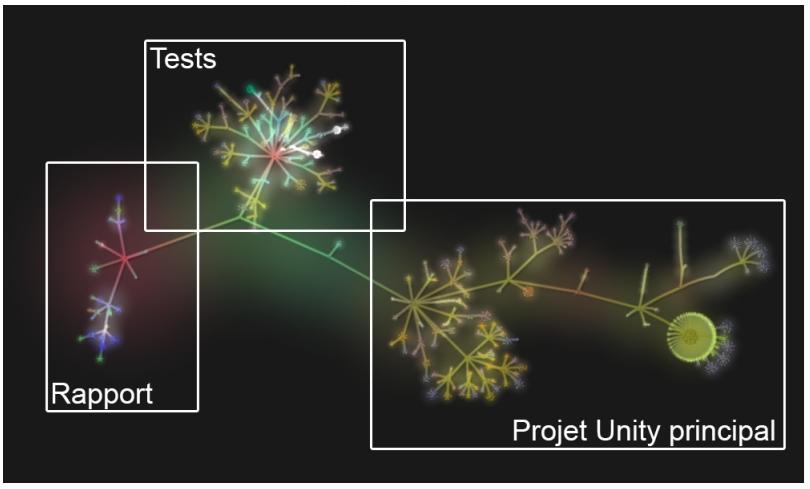


FIGURE 3.6 – Modélisation Gource en fin de projet

3.2.3 Unity

Unity est un logiciel orienté pour le développement de jeux vidéo intégrant un moteur physique 2D et 3D. Sa particularité réside dans la possibilité d'exporter l'application développée sur de nombreuses plateformes (Web, Android, iOS, consoles...). Nous avons réalisé notre projet avec la version 5 du produit.

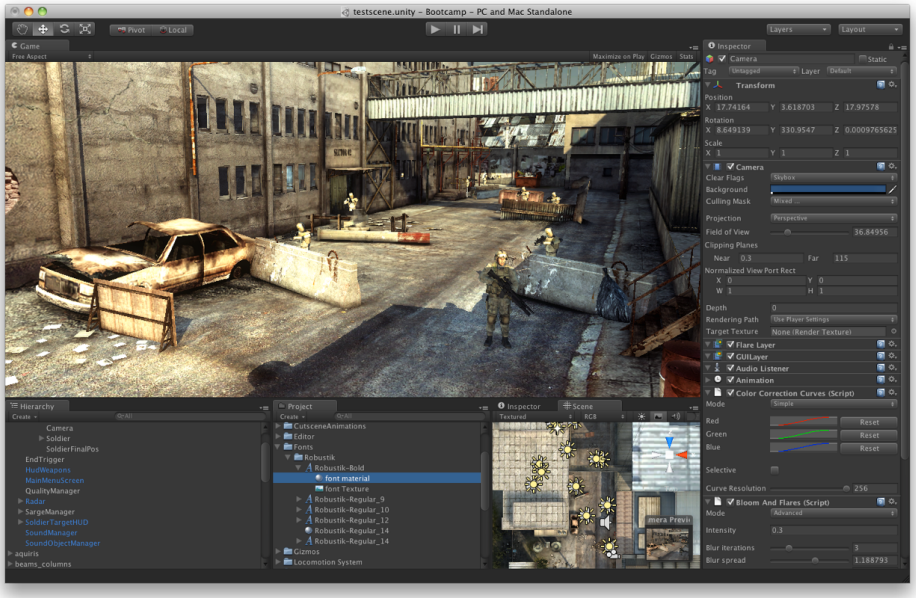


FIGURE 3.7 – Interface de Unity

Pourquoi utiliser Unity ? Nous avons choisi de développer en utilisant Unity afin de pouvoir déployer notre jeu facilement sur les plateformes mobiles populaires (Android, iOS et Windows Phone), sans avoir à développer trois fois la même application dans un langage différent. De plus, l'utilisation du moteur de Unity nous permet d'économiser le temps nécessaire à la création d'un moteur spécifique au projet, qui serait probablement mal optimisé. Enfin, la technologie Unity est de plus en plus populaire et de plus en plus de jeux voient le jour grâce à elle, nous avons donc profité du projet pour apprendre à l'utiliser.

Fonctionnalités de Unity Dans notre projet, nous utilisons principalement Unity pour :

- Afficher des *sprites*
- Réaliser des animations
- Jouer des sons
- Créer l'interface utilisateur
- Gérer les évènements clavier (ou tactiles)
- Exporter sur de multiples plateformes

Ce que nous n'utilisons pas avec Unity :

- Le moteur physique (gravité, collisions, squelettes...)
- La 3D

Ainsi, il y a de nombreuses fonctionnalités, nécessaires au développement de notre jeu, qui ne sont pas gérées par Unity et que nous devons développer.

Principe de Unity Lorsqu'un nouveau projet Unity est démarré, il faut d'abord ajouter un ou plusieurs objets à une scène. Cet objet peut être de tout type : un solide, une lumière, une caméra, un son, des particules... Ensuite, il est possible de greffer un script à cet objet. Ce script peut être développé en JavaScript ou en C#. Il est possible d'exécuter des instructions lors de la création de l'objet, de sa destruction ou en encore à chaque rafraîchissement de l'écran. Chaque composante de l'objet (taille, position, rendu...) est accessible et paramétrable directement dans le script. Enfin, les objets peuvent communiquer entre eux par le biais des scripts qui leur sont attachés. Les objets peuvent être connectés entre eux par de simples glissé-déposé.

Coût de Unity Unity possède deux types de versions : une version personnelle, et une version professionnelle. Les deux versions comportent les mêmes fonctionnalités, c'est à qu'il n'est pas forcément nécessaire d'acheter la version professionnelle pour développer un jeu vidéo de grande envergure. Cependant, l'achat de la licence Unity est nécessaire lorsque l'entreprise réalise un revenu brut sur une année de plus de 100 000$. L'achat de la version Unity Pro revient à 1500$ (ou 75$ par mois). Il est à noter qu'il est aussi nécessaire d'acheter les modules permettant d'exporter sous iOS ou Android, qui reviennent à 1500$ chacun (ou 75$ par mois). La version personnelle affichera néanmoins un *splash-screen* au démarrage du jeu vidéo, et ce, sur n'importe quelle plateforme.

3.3 Estimation du temps passé

Estimation globale Nous nous sommes réunis au moins une fois par semaine pour travailler, sur des séances d'environ 8h sur une journée. Sur 4 mois, cela représente un total de 128 heures par personne, et 512 heures au total. À cela, nous pouvons y ajouter toutes les heures de travail individuel que nous chiffrons facilement à 30% du projet. Soit **665 heures** au total.

Estimation assistée Étant donné que nous envoyons très régulièrement des commits, nous avons essayé d'utiliser un outil de calcul : **git-hours**. Il calcule les heures de travail en fonction des écarts entre les commits, mais ne compte évidemment que le temps de développement pur, les moments passés à discuter et débattre autour du projet en équipe ne sont donc pas comptabilisés. L'application retourne un total de **29822** minutes, soit **497 heures**. Ce qui semble totalement cohérent avec notre première estimation, en ajoutant les heures de débats en réunion.

Chapitre 4

Rapport technique

4.1 Moteur de jeu rythmique

Notre objectif premier est de réaliser un moteur de jeu de rythme générique sous Unity afin de pouvoir créer des mini-jeux aisément.

Nous définissons un jeu de rythme par les composantes suivantes :
- Une musique, avec un tempo et une durée fixée ;
- Des actions à réaliser par l'utilisateur, qui peuvent être de différents types ;
- Un décor animé et synchronisé sur la musique ;
- Un taux de réussite en % calculé sur les performances du joueur.

Pour réaliser cela, les fonctionnalités attendues du moteur sont les suivantes :
- Synchronisation parfaite d'un battement sur le tempo d'une musique, entrée de façon numérique manuellement ;
- Détection des temps entiers sur la musique, des demi-temps et des quarts de temps ;
- Scripting dans un fichier texte pour définir des actions sur les temps voulus. Que ce soit pour définir les comportements de l'environnement ou le comportement attendu de l'utilisateur ;
- Analyse des actions de l'utilisateur, et détection de sa réussite à partir du niveau scripté, avec une certaine tolérance ;
- Connexion de tous ces évènements à des objets de type Unity, pour pouvoir déclencher des animations et autres effets désirés.

4.1.1 Le *beater*

Le *beater* lit la musique et doit déclencher des évènements à chaque "tick" enregistré. Pour notre moteur, nous allons jusqu'à une précision d'un quart de temps. Un "tick" correspondra donc à chaque quart de temps de la musique.

Note technique sur la musique
Un fichier son numérique est enregistré sur l'unité de temps du sample. À chaque sample, on récupère une valeur numérique en décibels. Sur un son classique, la fréquence est de 44100Hz, ce qui correspond à 44100 samples par secondes. Dans ce projet nous convertissons toutes nos durées en samples pour une précision optimale.
Le tempo d'un morceau se mesure en BPM (battements par minutes). Il varie entre 80 et 160 BPM en moyenne sur des morceaux classiques, mais reste fixe tout au long de la musique.

Le principe du Beater est de déclencher un "tick" tous les quarts de temps. Sur un morceau à 120 BPM, on récupère la durée d'un tick en samples avec le calcul suivant :

```
samplePeriod = (60f / (tempo * scalar)) * audioSource.clip.frequency;
```

Ainsi, notre *beateur* boucle continuellement dans le temps et mesure le temps passé pour envoyer des évènements tout les X *samples* passés.

```
IEnumerator BeatCheck () {
    while (true) {
        if (audioSource.isPlaying) {
            float currentSample = audioSource.timeSamples;
            if (currentSample >= (nextBeatSample)) {
                this.beat();
                nBeat++;
                nextBeatSample += samplePeriod;
            }
        }
        yield return new WaitForSeconds(loopTime / 1000f);
    }
}
```

Note : Le code source complet du Beater est disponible en annexe.

Difficulté technique Sur Unity chaque itération de la boucle s'effectue à chaque *Update* du moteur, soit 60 fois par secondes sur PC, et 30 fois sur téléphone. 1 divisé par 30 = 0.03333333, soit 30ms entre chaque tick.

Sur un tempo à 120 BPM un quart de temps dure 107 ms ce qui offre une marge de manœuvre faible, d'où la difficulté de synchronisation. Sans une bonne pratique et des calculs correctement réalisés, le jeu perd rapidement la synchronisation avec la musique.

Pour des raisons de performances, nous nous devons de limiter le nombre de calculs par seconde. Des paramètres sont disponibles dans notre moteur afin d'affiner ce genre de détails avant la mise en production.

Dans le cadre de ce projet, nous avons passé de nombreuses heures avant d'arriver à détecter les *beats* parfaitement sans aucune perte d'information sur une longue durée. La méthode que nous présentons ici semble simple et fonctionnelle, mais nous en avons essayé beaucoup d'autres avant d'arriver à ce résultat, qu'il serait inintéressant d'expliquer.

De plus, nos exigences ont changé avec le temps. Au départ, nous pensions qu'avoir la précision du double temps été suffisant, mais rapidement nous avions besoin de quarts de temps... Nous retenons ici qu'il vaut mieux voir au plus compliqué dans la conception, pour être certain que toutes les possibilités futures soient couvertes.

4.1.2 Autres fonctionnalités du *beateur*

Une musique commence rarement sa première seconde au temps 0 de son tempo (il peut y avoir un léger blanc), un paramètre "offset" `start` est disponible dans notre moteur pour définir le temps à attendre avant de détecter le premier *beat*. Ceci permet de synchroniser parfaitement les animations avec la musique.

Pour les calculs de réussite ou calcul du score, on peut avoir besoin d'autres fonctionnalités de calculs, par exemple pour récupérer le numéro du tick le plus proche à un instant T.

```
// Retourne le numero du step le plus proche au temps T
public int getStepClosest() {
    float currentSample = audioSource.timeSamples - sampleOffset;
    float score = currentSample % samplePeriod;
    int step = (int) (currentSample / samplePeriod);
    if (score > samplePeriod/2) {
        step++;
    }
    return step;
}
```

Tous ces calculs prennent en compte le `sampleOffset`.

4.1.3 Le niveau (*LevelScripted*)

Pour pouvoir réaliser des niveaux intéressants, il nous faut pouvoir les scripter afin de définir sur quels "ticks" le joueur doit frapper. Il peut y avoir de longs blancs, ou des enchaînements rapides (avec une fréquence maximale équivalente au quart de temps).

Nous avons décidé de représenter un niveau dans un fichier texte de la façon suivante :

```
1 1 0 0 2 1 2 0 0 1 2 0 2 1 0 0 3 1 0 0 0
0 0 0 1 0 0 0 1 0 0 0 2 0 0 0 1 0 0 0 0 0
1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Le fichier se lit dans le temps de haut en bas (pour chaque quart de temps), puis de gauche à droite (pour chaque temps plein). Les différentes colonnes correspondent aux différents quarts de temps. Par exemple, si l'on veut simplement déplacer un cube à chaque battement (temps fort), on écrit le fichier suivant :
Un "0" correspond à "aucun évènement", et un chiffre correspond à un type d'évènement. Nous nommerons les chiffres supérieurs à zéro les **actions**.
Une action peut être de type différent afin de pouvoir varier les attentes du joueur. En effet, il peut y avoir plusieurs mouvements différents au niveau du tactile. Par exemple, le "1" peut représenter un tap simple, le "2" un appui long, le "3" un relâchement, etc.

```
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

FIGURE 4.1 – Exemple de niveau

Ce type de fichier peut servir à scripter les actions attendues de la part du joueur, mais aussi à scripter n'importe quel autre élément dans un mini jeu, comme un personnage animé ou des déclenchements exceptionnels d'animations à certains instants clés de la musique.

La classe `LevelScripted` est connectée au *beater* pour recevoir les ticks, et filtre en lisant le fichier, pour envoyer les actions de type 1, 2, 3... D'autres objets peuvent ensuite se connecter à un `LevelScripted` pour recevoir ces évènements.

La longueur du fichier dépend de la longueur de la musique. On peut écrire de petits fichiers pour les boucler et créer des patterns, car le moteur répète automatiquement le fichier tout au long de la musique.

4.1.4 Évènements du joueur et calcul de la réussite

`EventPlayerListener` Une classe est destinée à écouter les évènements du joueur, afin de détecter des actions de type 1, 2, 3, 4... L'endroit où l'utilisateur appuie sur l'écran n'a aucune importance.

Les mouvements suivants sont définis :

1. Tap bref (le plus commun) ;
2. Commencement appui long ;
3. Relâchement appui long ;
4. Swipe (lancer).

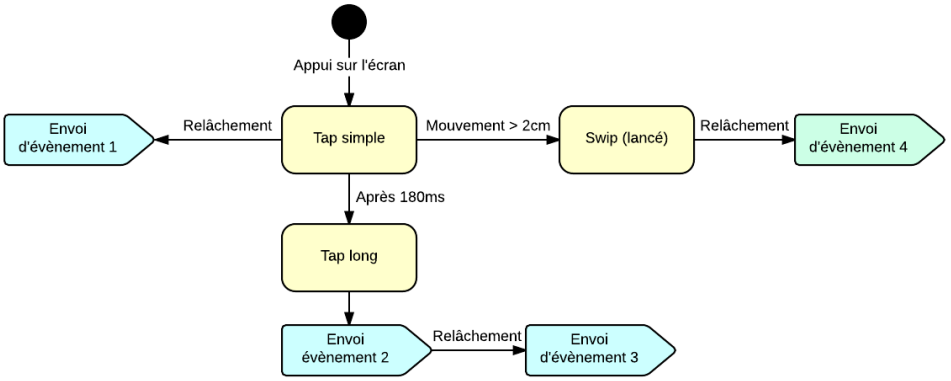


FIGURE 4.2 – Diagramme d'états-transitions des évènements

La gestion des évènements est délicate, dans le sens où elle se joue à quelques millisecondes près. Un temps d'attente trop long, et le joueur retrouve un décalage entre le

moment où il appuie, et le moment où l'évènement est envoyé. Trop court, et le tap bref est interprété par un tap long, et le glissé n'a pas le temps de se faire. De même, une distance trop courte, et le moindre mouvement du doigt est interprété comme un lancement. Trop longue, et le risque qu'il soit interprété comme un évènement long, ou qu'un décalage se crée, apparaît. Ainsi de nombreux tests ont dû être effectués, nécessitant un mouvement infime des paramètres, et demandant à chaque fois de faire de nouveau une compilation, un transfert sur le téléphone et un test, puisque seule l'expérimentation permet de savoir si le réglage est bon.

Détection de la réussite Une fois les évènements convertis en numéros d'action, il faut vérifier si ces numéros sont en cohésion avec le niveau chargé. On ne compte que les échecs, et à la fin du niveau on soustrait le nombre total d'actions à réaliser par le nombre d'échecs pour obtenir le % de réussite sur le niveau. On ne met pas en place d'échelle de réussite comme on peut voir dans les autres jeux (mauvais, bon, parfait...). On considère que le joueur réussit ou rate un évènement.

Il y a deux types d'évènements à tester :

- Au moment où le joueur appuie, il faut vérifier que le numéro d'action tapé correspond au numéro d'action courant du niveau. On incrémente le nombre d'échecs si ça ne correspond pas.
- Quand un évènement est passé, il vérifie si le joueur l'a réussi. Dans le cas où le joueur ne joue pas, il faut compter des erreurs.

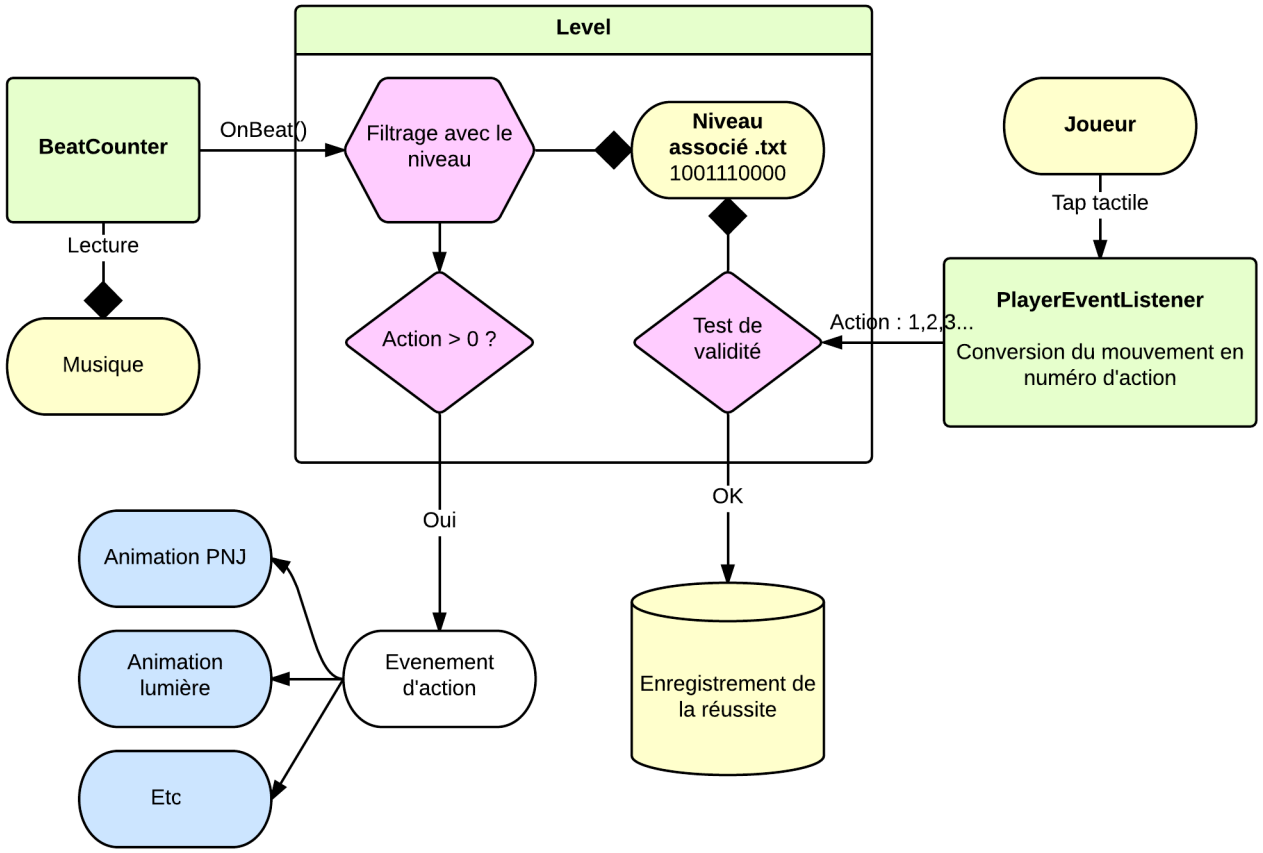
La phase complexe est de déterminer quand un évènement est trop tôt ou trop tard. Un joueur ne frappera jamais pile au moment réel de l'évènement : il faut mettre en place un système de tolérance.

Après de multiples tests, nous avons conclu que le délai d'un quart de temps est suffisant comme négligence. C'est-à-dire que le joueur dispose d'un quart de temps complet pour réaliser une action demandée (représente environ 100ms).

On stocke dans une liste les numéros des ticks où le joueur a réussi. Quand le *beateur* envoie un évènement, on regarde si le joueur a réussi le précédent. Sans quoi, on envoie un évènement `onFailure`.

N'importe quel objet Unity peut se connecter à ce contrôleur. Ceci permet de construire le feedback visuel, en jouant par exemple une animation quand on reçoit un évènement `onFailure`.

4.1.5 Résumé du fonctionnement du moteur



Pour résumer en s'appuyant sur la figure 4.1.5 :

- Le `BeatCounter` lit la musique et envoie des évènements `OnBeat()` à chaque quart de temps.
- Le `LevelScripted` filtre ces évènements en lisant le fichier, et envoie des évènements d'actions à tous les objets Unity qui l'écoutent
- Quand le joueur tape sur l'écran, le `PlayerEventListener` convertit le mouvement en numéro d'action
- Les actions du joueur sont validées par le `PlayerActions` qui enregistre si l'action correspond au fichier

4.2 Outil de construction des niveaux

Même si notre système de fichier texte est suffisamment clair pour être lu et modifié manuellement, il est tout de même fastidieux de construire des niveaux directement. Il est beaucoup plus intéressant de créer les actions du niveau dans un logiciel de musique. Nous avons donc développé un outil de conversion pour générer un niveau à partir de fichiers midi.

4.2.1 Choix du langage

La programmation est la représentation des données et la manipulation de ces représentations. Les fichiers midi sont représentés par des listes de listes. Quel meilleur langage

qu'un LISP pour effectuer des opérations sur des listes ? Le Scheme s'est imposé comme choix naturel.

4.2.2 Structure d'un fichier midi

Les spécifications du midi sont lourdes et complexes, elles ne seront pas détaillées ici, seules seront détaillées les informations importantes à notre convertisseur de niveau. (Un extrait est fourni en annexe).

Un fichier midi est composé d'une suite de "chunks", eux-mêmes composés d'évènements, il existe un certain nombre d'évènements différents, tels que le nom de la piste, le tempo, le nombre de pistes... Nous détaillerons uniquement les évènements pertinents à notre programme.

`fichier midi = <header chunk> + <track chunk> [+ <track chunk> ...]`

Un fichier midi commence par un *header* formé de la manière suivante :

`header chunk = "MThd" + <header length> + <format> + <n> + <division>`

```
;; lecture de l en-tete du fichier
(define (read-header in)
    (header (to-string (read-n-bytes in 4))
            (to-int (read-n-bytes in 4))
            (to-int (read-n-bytes in 2))
            (to-int (read-n-bytes in 2))
            (to-int (read-n-bytes in 2))))
```

`track chunk = "MTrk" + <length> + <track event> [+ <track event> ...]`

Un *event* se présent sous la forme suivante :

`track event = <delta time> + <midi event> | <meta event>`

Les *meta events* servent à donner des informations telles que le tempo, la division du tempo, ou la fin d'une *track*.

```
;; events utilises
(define time-signature-event '(255 88 4))
(define set-tempo-event '(255 81 3))
(define sequence-name-event '(255 3))
(define instrument-name-event '(255 4))
(define key-signature-event '(255 89 2))
(define smpte-offset-event '(255 84 5))
(define midi-channel-prefix-event '(255 32 1))
(define end-event '(255 47 0))
```

Les *midi events* sont quant à eux des évènements directement en rapport avec la musique, le début ou la fin d'une note, ou encore le changement de canal. `midi event = <status byte> + <data byte> + <data byte>`

Un fichier midi est un fichier binaire, à sa lecture, c'est une suite de valeurs hexadécimales, il est représenté dans notre programme comme une suite de valeurs de 0 à 255 : '(20 255 88 4 60 100 ...).

En parcourant le fichier, on peut reconnaître par exemple la suite d'octets "255 88 4", qui fait partie de nos évènements connus, on connaît également la taille de cet évènement,

on sait donc que les 4 octets suivants formeront un *event* de type "time signature".

```
;;vrai si toutes les valeurs de l1 sont dans l2
(define (sublist? l1 l2)
    (andmap (lambda (i j)
        (= i j)) l1 (take l2 (length l1))))

;;vrai si les premiers octets de data
;;forment un event connu : e
(define (known-event? e data)
    (or (sublist? e data) (sublist? e (cdr data))))
```

Le delta time est codé avec une quantité à longueur variable. Le delta time n'est pas par rapport au début de la piste, mais par rapport à l'évènement précédent. C'est lui qui permettra à la musique dans le fichier midi d'avoir un rythme, par exemple, plus le delta time est long entre un évènement de début de note et un évènement de fin de note, plus la note sera tenue longtemps.

Note technique sur la quantité à longueur variable (*variable-length quantity*)
La vlq permet de représenter de manière compacte des quantités supérieures à un octet.

VLQ Octet

7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
A	B_n						

Si `A` est égal à 0, c'est que c'est le dernier octet de la quantité. Si c'est 1, un autre octet vlq suit.
`B` est un nombre de 7 bits, et `n` est la position de l'octet où `B0` est l'octet de poids faible.

Certains éditeurs de fichiers midi utilisent une technique appelée le "running status" pour réduire la taille de leurs fichiers. Pour clairement comprendre son fonctionnement, une explication supplémentaire sur les *midi events* s'impose.

Le *status byte* a une valeur comprise entre 128 et 255, les *data bytes* ont, quant à eux, une valeur comprise entre 0 et 127.

Le "running status" consiste à ne pas répéter le *status byte* s'il est identique à l'évènement précédent. L'utilisation du "running status" est triviale à détecter et implémenter. En lisant le fichier, si l'octet lu est inférieur à 128 alors que l'on attendait un *status byte*, c'est qu'il faut utiliser le dernier *status byte* rencontré.

4.2.3 Conversion en niveau

En possession des ces informations, et avec la table des codes midi (voir annexe), convertir le fichier midi en niveau n'est alors plus qu'une succession de transformation de représentations. D'abord en *chunks*, puis en *tracks*, et enfin en *events*, en filtrant les évènements inutiles à notre cas d'utilisation.
Une fois les évènements extraits du fichier midi, nous sommes en mesure de les convertir en actions pour notre jeu.

L'action 1 correspond à la note C, l'action 2 à la note C#, et l'action 3 à la note D. À chaque évènement avec le *status byte* "début de note" (de l'octet 0x90 à l'octet 0x9F), l'action correspondant à la note de l'évènement est ajoutée au niveau. Ces actions sont

séparées avec des zéros, eux donnés par le *delta time*.

```
;; transforme un evenement en donnees de niveau (0, 1, 2...)
;; division est le nombre de frames par secondes
;; delta-sum est la somme des delta depuis le dernier event utile
(define (event-to-level event division delta-sum)
    (let ([n (/ (+ delta-sum (vlq->int (midi-event-delta event))) (/ division 4))])
         ; n est le nombre de temps ou rien ne se passe (0)
         ; midi-event-arg1 est la note
         ; notes est la hashmap ou sont faites les correspondances
         (append (make-list n 0) `(,(hash-ref notes (midi-event-arg1 event) ?)))))
```

4.2.4 Mise en pratique

On utilise dans cet exemple le logiciel Logic Pro X sous Mac OS X. N'importe quel autre séquenceur gérant les fichiers MIDI peut être utilisé.

On importe la musique sur une piste, et on indique au logiciel son tempo (qui doit être connu, on peut utiliser un logiciel commit *Audacity* pour le détecter).

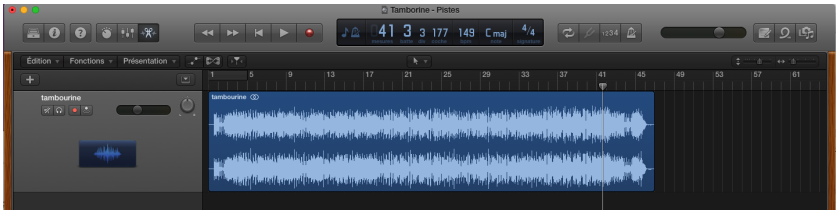


FIGURE 4.3 – Piste sonore

On ajoute sur une seconde piste vide un instrument qui représente le niveau à créer, sur laquelle on va ajouter les actions.

FIGURE 4.4 – Piste MIDI

On pose ensuite les notes qui représentent les actions sur cette piste, puis on écoute le tout en temps réel pour superposer proprement chacune des actions sur la musique. On utilise des notes différentes pour chaque type d'action. Ici le DO pour une action 1, le DO# pour une action 2, etc.

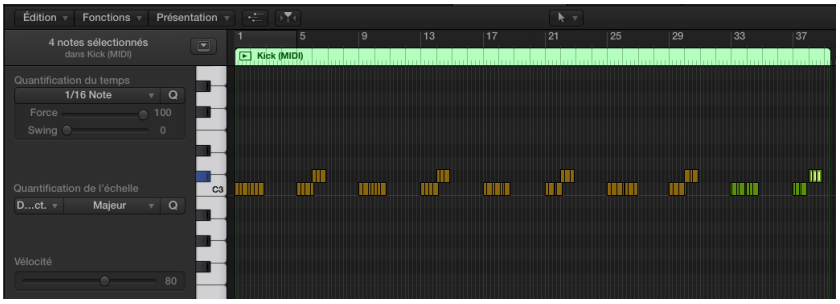


FIGURE 4.5 – Éditeur de notes de Logic Pro

Une fois le niveau construit, il ne reste plus qu'à exporter la piste au format .midi, et de le passer au convertisseur pour obtenir le niveau au format .txt.

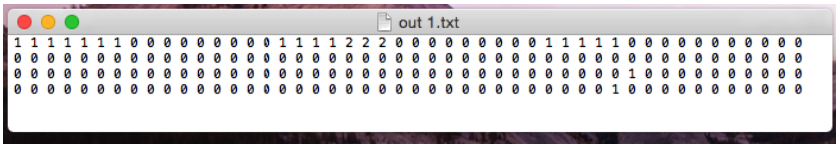


FIGURE 4.6 – Fichier .txt du niveau en sortie

4.3 Avancement du joueur

4.3.1 Calcul du score dans un mini-jeu

Le calcul du score est fait sous la forme d'un pourcentage. En fait, seul le nombre d'erreurs est regardé, puisque peu importe que le joueur fasse tout bon, s'il se contente d'envoyer des évènements dans tous les sens en appuyant sur son tactile. C'est pourquoi le nombre de réussites n'est pas regardé à proprement parler. Cependant est considéré comme un échec :

- Un évènement envoyé à un moment non attendu
- Un évènement envoyé alors que c'est un autre qui est attendu
- Ne pas envoyer d'évènement alors qu'un évènement est attendu

Ainsi, même si les succès ne sont pas considérés, étant donné que l'absence de succès est considérée comme un échec, ne rien faire ou spammer le tactile ne donne en aucun cas un bon score, pas moyen de "tricher" pour gagner, la seule manière étant de jouer correctement. Ensuite, une fois les échecs additionnés, on se contente de calculer le pourcentage en divisant le nombre d'échecs par le nombre d'évènements attendus. Ainsi, 0 correspond au fait de n'avoir fait aucun échec, 1 correspond au fait d'avoir fait autant d'échecs que d'étapes à effectuer (et voulant dire qu'on a fait plus d'erreurs que d'étapes). On soustrait ce nombre à 1, et on multiplie par 100 pour obtenir un pourcentage. On se retrouve donc à avoir 100% si on n'a fait aucune erreur, 50% si on a loupé la moitié des évènements, et on renvoie 0% si on fait autant d'erreurs ou plus que d'évènements à faire.

4.3.2 Enregistrement des scores

Même si le calcul des scores est fait sous la forme de pourcentage, ce qui est affiché au joueur et ce qui est enregistré est sous la forme d'étoiles. 3 étoiles correspondent à 100% de réussite, 2 étoiles à un score supérieur à 90% et enfin 1 étoile à un score supérieur à 75%. Le score est enregistré de manière très simple, Unity permettant d'enregistrer des

entiers, des chaînes de caractères et des nombres flottants. Ainsi le score est enregistré dans une variable "int" s'appelant "etoileLevel[numéro du mini jeu]" et sa valeur correspond au nombre d'étoiles. À chaque fin d'un mini-jeu, on vérifie si le joueur n'a pas amélioré le nombre d'étoiles sur ce mini-jeu, et si oui, alors on écrase le précédent score avec le nouveau qui est meilleur.

FIGURE 4.7 – Score du joueur affiché

4.3.3 Déblocage des mini-jeux

Certains jeux sont débloqués de base, d'autres nécessitent la réussite d'un autre mini-jeu pour cela. Un mini-jeu est considéré comme réussi du moment que le joueur a obtenu au moins une étoile sur ce jeu. Ainsi, si un mini-jeu n'est pas débloqué, dans le menu de sélection des mini-jeux, son icône est grisée, et il est non cliquable. De plus, en dessous de l'icône de chaque mini-jeu est affiché le plus grand nombre d'étoiles obtenu dans le jeu correspondant.

FIGURE 4.8 – Sélection des mini-jeux

4.4 Création d'un mini-jeu

Afin d'obtenir un niveau cohérent, le processus de création d'un mini-jeu doit se faire étape par étape. Le travail effectué sur le premier niveau a permis de peaufiner ce processus et de réaliser les mini-jeux suivants avec plus d'efficacité et de rapidité. Dans un premier temps, nous avons commencé à développer le niveau des champignons en suivant des étapes qui nous paraissaient logiques. Puis, après avoir développé la majeure partie du niveau, nous avons réalisé qu'il manquait des éléments importants au gameplay, tels que des sons cohérents correspondants au rythme et à l'image, ou un *feedback* visuel montrant la réussite ou l'échec du joueur.

Pour illustrer ces différentes étapes, des exemples seront tirés du mini-jeu des champignons.

4.4.1 Les graphismes

Un des éléments les plus importants d'un jeu est son aspect visuel. Il est préférable de commencer par avoir une base solide au niveau de ce que l'on veut que le joueur fasse. Ensuite, il s'agit d'imaginer une scène simple dans laquelle l'action du joueur ne serait pas aberrante. Par exemple, si le joueur doit répéter un motif sonore, alors il vaut mieux que les graphismes représentent au moins deux personnages frappant sur une surface, un représentant le modèle, et l'autre le joueur.

Dans notre projet, nous avons voulu mettre en avant le côté simple et divertissant de notre thème en utilisant des graphismes 2D de type *cartoon*. Nous avons créé nos propres graphismes, en utilisant des couleurs vives et des traits de contours très épais.

FIGURE 4.9 – Exemple visuel de fichier graphique vectoriel

Pour cela, nous avons utilisé Adobe Illustrator, qui permet de réaliser des créations graphiques vectorielles. L'intérêt de travailler sur du vectoriel est qu'on peut rendre l'image

dans la dimension voulue, et, de plus, il est plus facile d'apporter rapidement une petite modification un objet ou sa couleur, sans avoir tout à recommencer.

4.4.2 Les animations

Les animations utilisées dans l'application sont gérées directement par Unity. Leur mise en place est classique, elle se fait via l'utilisation de clés sur une ligne de temps.

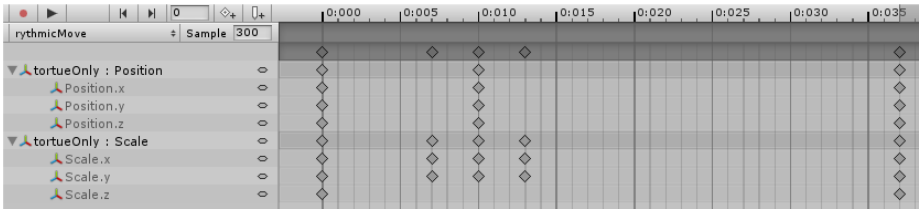


FIGURE 4.10 – Exemple d'utilisation de clés sur une ligne temporelle

Le principe étant de donner les caractéristiques de l'objet à animer (position, taille, rotation ...) à un temps donné, et de les stocker dans une clé. Une fois que deux clés ont été créées, une interpolation linéaire est ensuite appliquée entre les deux clés afin d'obtenir un déplacement fluide sur les images intermédiaires. La courbe d'interpolation peut être modifiée afin d'obtenir l'effet voulu.

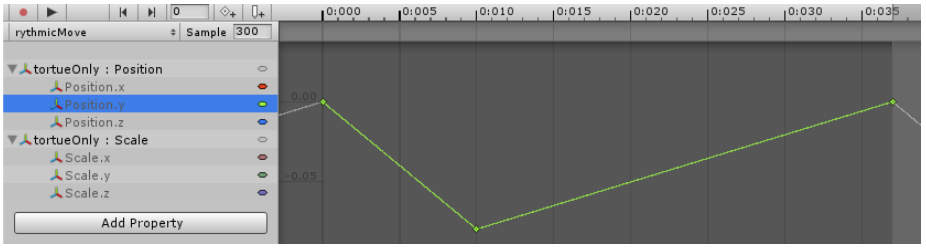


FIGURE 4.11 – Interpolation simple entre trois clés définissant la position y de l'objet

Unity possède la caractéristique de pouvoir créer un diagramme d'état afin de jouer les différentes animations dans l'ordre désiré. Dans notre projet, nous avons utilisé cette fonctionnalité pour nous aligner sur le rythme de la musique. Pour cela, nous avons créé un arbre simple qui possède un état *immobile* et un état *mouvement*. À chaque fin d'une animation, c'est l'animation suivante qui sera jouée, en synchronisation avec les battements par minute de la musique.

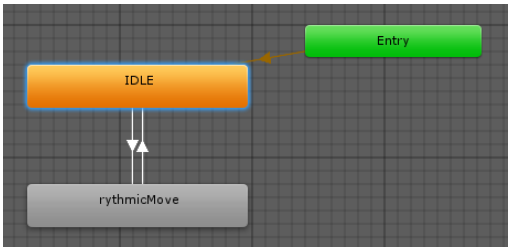


FIGURE 4.12 – Diagramme d'état utilisé pour l'animation

4.4.3 Assemblage avec le moteur

On va donc brancher les différents éléments composant le jeu au moteur. Les composants du moteur sont détaillés partie 4.1.5.

Le passage de message

Plusieurs façons de faire passer des messages sont utilisées dans le moteur.

Le passage de message par *trigger* est employé pour déclencher les animations (voir partie 4.4.2). Il est possible d'ajouter des *triggers* aux animations comme montré figure 4.13. Lorsqu'un trigger est déclenché, on passe à l'état suivant de l'animation. Cela permet, entre autres, de ne pas spécifier qui est connecté à qui, mais de simplement envoyer des messages d'un coté et de les récupérer de l'autre sans se soucier de comment il y est arrivé.

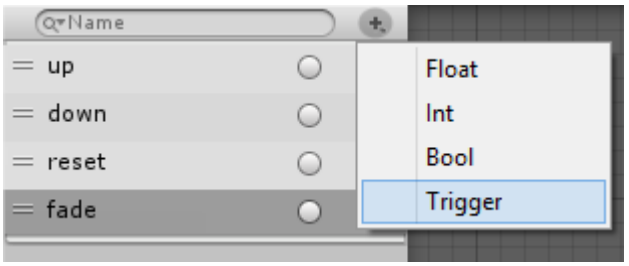


FIGURE 4.13 – Exemple d'ajout de trigger

Il est également possible de connecter directement des objets entre eux, pour que l'objet qui se connecte reçoive des notifications de l'objet auquel il s'est connecté. C'est par exemple ainsi que le *BeatCounter* notifiera le *LevelScripted* qu'il y a eu un beat.

```
public void connect (LevelScriptedReceiver r) {
  this.observers.Add (r);
}

public void Disconnect (LevelScriptedReceiver r) {
  this.observers.Remove (r);
}

private void notifyChildren (int type) {
  foreach (LevelScriptedReceiver e in this.observers) {
    e.OnAction (type);
  }
}
```

FIGURE 4.14 – Méthodes servant à la communication entre les objets

L'assemblage des éléments

Commençons par les éléments du décor et les personnages non joueurs. Ils seront reliés à un *LevelScripted* auquel on passera en paramètre le fichier texte correspondant à leur mouvement. Les mouvements sont et les animations (voir partie 4.4.2) déclenchés à l'aide de *trigger* que le *LevelScripted* va envoyer en fonction des évènements lus.
Typiquement on aura un *LevelScripted* avec un fichier texte ressemblant à celui fig. 4.1 qui

servira à battre la mesure et à indiquer le tempo au joueur.
On aura ensuite un *LevelScripted* avec un vrai niveau qui va scripter les mouvements d'un personnage non-joueur comme on peut le voir par exemple dans le niveau des champignons où le joueur doit répéter les mouvements du PNJ.
Les *LevelScripted* sont évidemment reliés au *BeatCounter* pour que tous les mouvements du décor et des PNJ soient effectués en rythme.

Les animations du personnage joueur fonctionnent globalement de la même manière. La seule différence est que l'on va ajouter un *PlayerActions* et un *PlayerEventListener* qui vont se charger de récupérer les actions du joueur et d'interroger le *LevelScripted* pour savoir si l'action du joueur correspond à l'action attendue, et ainsi envoyer le trigger correspondant.

4.4.4 Le *feedback*

Le *feedback* est l'ensemble des signes visuels ou sonores que recevra le joueur en fonction de son action. Dans notre application, il permet de signaler la réussite ou l'échec d'une action de l'utilisateur. Ainsi un coup réussi jouera un son cohérent avec la scène et une animation gratifiante sera jouée, afin de récompenser rapidement le joueur. À l'inverse, une action ratée résultera à un son lié à un échec et des graphismes montrant le mécontentement. Le feedback retourné par l'application s'est avéré être un des points les plus difficiles à imaginer dans la création des scénarios des mini-jeux.

4.4.5 Le choix des sons

Maintenant que la plupart des éléments sont placés et fonctionnels dans la scène, il s'agit d'ajouter le plus important : la musique. Son rôle est important, car c'est sur son rythme que l'utilisateur devra se synchroniser pour réussir le niveau. Elle se doit donc de comporter des rythmes prononcés et un thème en rapport avec la scène créée. Ne pouvant pas composer nous même notre propre musique, nous avons fait le choix d'ajouter dans notre jeu des musiques proposées gratuitement et libres de droit sur Internet.

4.4.6 La difficulté

Après avoir développé tout le contenu d'un mini-jeu, il faut évaluer sa difficulté en se mettant dans la peau d'un joueur qui le découvre pour la première fois. Celle-ci ne doit être, ni trop faible, ni trop élevée, pour ne pas se décourager, ni trop simple, pour ne pas s'ennuyer. Pour répondre à ce problème, nous avons d'abord placé des motifs simples à réaliser, puis augmenté peu à peu leur difficulté au fur et à mesure de l'avancement du niveau. Ensuite, nous avons fait tester nos ébauches de niveau à des personnes externes au développement qui ont pu juger de la difficulté du jeu.

4.5 Les tutoriels

Le tutoriel est crucial dans la perception que le joueur a du jeu, c'est sa première interaction avec le gameplay, la plus importante, celle qui dictera si le joueur restera ou sera rebuté par un gameplay désagréable, inintéressant ou trop difficile. Un tutoriel se doit donc d'être ludique et accessible, tout en enseignant correctement les bases du jeu.

Nos tutoriels placent de plus le joueur dans le contexte du jeu, en lui donnant un semblant d'histoire. Le tutoriel est donc divisé en étapes successives, du texte, pour l'histoire et les explications, et une phase de jeu, dans laquelle le joueur applique ce qu'il vient d'apprendre dans la phase précédente. Cette boucle se répète autant de fois qu'il y a de notions à apprendre pour le jeu.

Comme expliqué section 4.1.3, les niveaux sont scriptés par des fichiers textes. La même technique est utilisée pour les tutoriels, qui ne sont rien d'autre que des niveaux normaux, un fichier texte par étape. À la première étape, le joueur apprendra par exemple le tap court, le fichier texte sera donc uniquement composé de ces évènements, puis lorsque cet élément sera appris, le moteur chargera un autre fichier texte contenant un autre élément de gameplay jusqu'à ce que le joueur les ait tous appris.

Pour être certain que le joueur maîtrise un élément de gameplay, et ainsi pouvoir passer à l'étape suivante du tutoriel, il lui est demandé de le répéter trois fois. C'est pour s'assurer que le joueur n'a pas réussi par chance et est donc incapable de compléter un niveau, ce qui pourrait entraîner de la frustration.

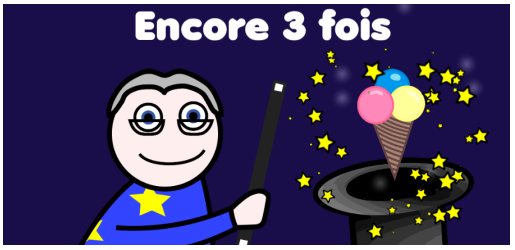


FIGURE 4.15 – Compteur du nombre de répétitions restantes

Nous avons fait le choix de proposer le tutoriel à chaque fois que le niveau est lancé, désagrément minime pour un joueur expérimenté, mais obligatoire pour un jeu destiné à être partagé et montré. Désagrément minime, mais seulement s'il est possible de passer le tutoriel. Il est donc donné au joueur la possibilité de passer le tutoriel après qu'il l'ait complété au moins une fois.

FIGURE 4.16 – Bouton skip verrouillé

FIGURE 4.17 – Pop up correspondante

FIGURE 4.18 – Bouton skip déverrouillé

FIGURE 4.19 – Pop up correspondante

En pratique, un tutoriel est composé d'un tableau de *StepTuto*, chaque *StepTuto* correspond à un élément du tutoriel : un texte ou une phase de jeu. Chaque classe implémentant *StepTuto* possède une méthode *endStep()* qui permettra au tutoriel de passer à l'étape suivante. Cette méthode est appelée lorsqu'un step est terminé, c'est-à-dire que le joueur a touché l'écran après avoir lu le texte ou qu'une phase de jeu a été réussie. Lorsque tous les éléments du tableau ont été visités, c'est-à-dire que toutes les étapes du tutoriel sont achevées, le tutoriel est terminé.

4.6 Assemblage des jeux

Chaque jeu est en réalité une scène de Unity. La navigation entre les scènes se fait suivant le diagramme fig. 4.20. L'écran principal est un "hub" où sont regroupés tous les jeux, on accède bien évidemment au jeu en cliquant dessus. Le tutoriel est ensuite chargé, puis enfin le jeu, on peut remarquer que l'on peut à tout moment revenir à l'écran principal.

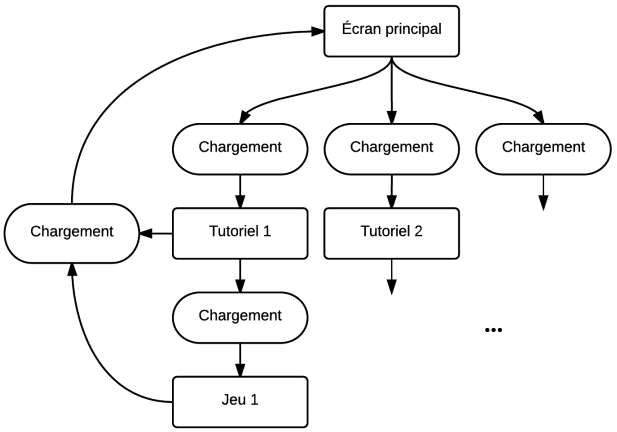


FIGURE 4.20 – Transitions entre les scènes du jeu

Les scènes ne se chargent pas instantanément, en Unity indique le pourcentage de la scène qui a été chargée, cela permet entre autres de réaliser des écrans de chargement animés. En plus de donner au joueur l'avancement du chargement, cela rend le jeu bien plus fluide que si le chargement était simplement un écran noir, comportement par défaut de Unity.

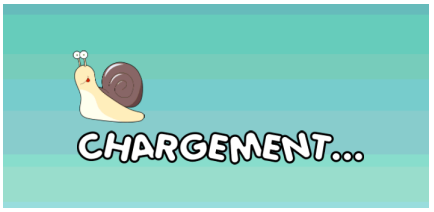


FIGURE 4.21 – Écran de chargement

Les transitions entre les scènes se font donc par des écrans de chargements, mais il serait un peu brusque d'interrompre directement le jeu lorsqu'il est terminé. Nous avons décidé de faire une fermeture en cercle (voir fig. 4.23) sur un élément de la scène pour donner un effet cartoon.

FIGURE 4.22 – Scène normale

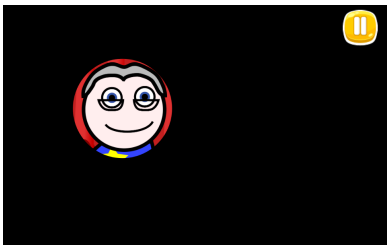


FIGURE 4.23 – Fermeture en cercle

Il faut savoir qu'il n'y a pas de manière simple de réaliser cet effet dans Unity. L'approche logique aurait été de coller de grandes zones noires aux bords de l'image du cercle vide, mais il n'est pas possible de manipuler la position et la taille des éléments dans Unity, ces données sont modifiées par le *scale* (échelle) de l'image. Difficile donc d'avoir un effet précis et fluide en multipliant la taille de l'image par un scalaire. C'est la solution de la physique qui a finalement été choisie, un centre de gravité au centre du cercle, et en soumettant les grandes zones noires à la physique, elles se déplacent de façon fluide en suivant le bord du cercle, qui lui a son scale diminué.

4.7 Services externes

Outre le jeu à proprement parler, nous avons intégré d'autres services externes afin de monétiser notre jeu, et d'obtenir des statistiques intéressantes.

4.7.1 Statistique et Analyse

La priorité étant de pouvoir par la suite améliorer le jeu, et pour ce faire, nous avions besoin de savoir comment les utilisateurs interagissent avec notre application. Il se trouve que Google fournit une très bonne interface pour fournir des statistiques d'utilisation : "Google Analytics". Et un plug-in permettant d'utiliser ces services avec Unity existe. "Google

Analytics" est un service qui permet d'analyser le comportement des utilisateurs sur les applications.

Il fournit différents outils, qui permettent de connaître la zone géographique de nos utilisateurs, le matériel utilisé, la durée moyenne d'utilisation de notre application, la manière dont ils ont trouvé l'application, mais aussi leurs habitudes d'utilisation, les écrans qu'ils visitent, leur passage ou non des tutos et pleins d'autres informations utiles. Bien utilisé, il permet donc de connaître vraiment le rapport qu'entretiennent les joueurs avec le jeu, et ainsi de donner des pistes sur les améliorations possibles, tant en terme de marketing que sur le jeu lui-même. En gros, un outil indispensable pour qui souhaite améliorer son produit, et augmenter le nombre de téléchargements.

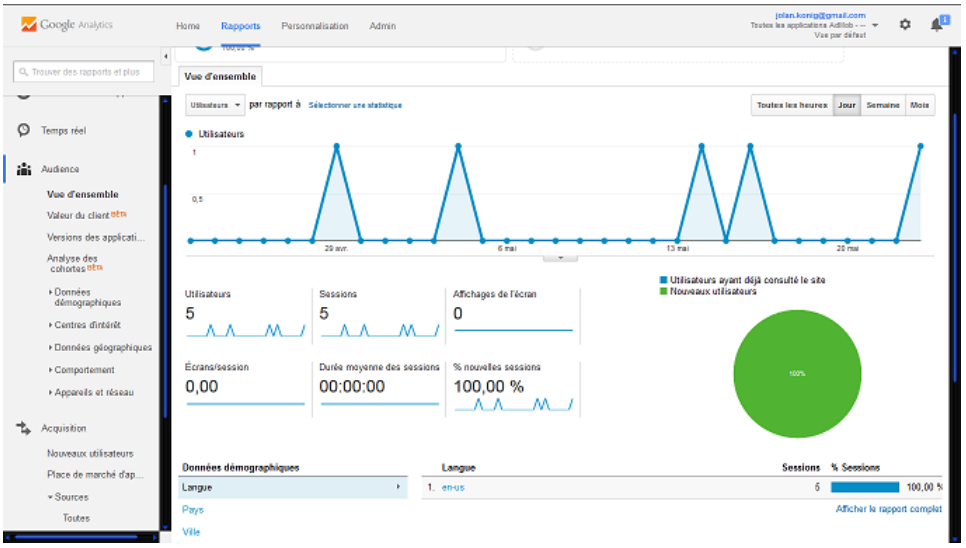


FIGURE 4.24 – Aperçu des statistiques fournies pas Google Analytics

4.7.2 Publicité

Ensuite, la monétisation à proprement parler. Nous avons opté pour un jeu gratuit, sans boutique, mais monétisé à partir de la publicité. Sur le support mobile, 3 grandes régies publicitaires dominent, Admob (Google), Iad (Apple) et Chartboost. Nous avons choisi Chartboost, pour plusieurs raisons. La première étant qu'elle est réservée pour les jeux mobiles, ainsi nous étions sûrs d'avoir des publicités ciblées, et adaptées à nos utilisateurs. Ensuite leur système de rémunération est assez intéressant, les gains étant plutôt importants, bien que seulement au téléchargement (quand un joueur installe un autre jeu à partir du nôtre). Enfin, la possibilité de développer facilement des partenariats avec d'autres concepteurs de jeu, et ainsi de nous permettre d'avoir de nouvelles sources d'utilisateurs gratuitement.

Nous ne voulions pas de pubs envahissantes, qui gênent l'utilisateur dans son utilisation de l'application, et perturbent la navigation dans le jeu. C'est la raison pour laquelle nous avons privilégié l'utilisation d'interstial en fin de partie, plutôt que d'une bannière tout au long du jeu. Une partie durant relativement longtemps, et sachant qu'un joueur fait plutôt peu de parties par session, nous avons décidé d'en mettre une à la fin de chaque partie, juste avant l'écran de fin lui indiquant le résultat obtenu lors du mini-jeu.

4.7.3 Version payante sans publicité

Enfin, sachant que certains utilisateurs sont réticents à la publicité (ce qui est compréhensible), mais qu'ils peuvent vouloir soutenir les développeurs, nous avons donné la possibilité de payer une version payante afin d'obtenir le jeu, mais sans aucune publicité. Nous avons utilisé un plug-in utilisé normalement pour la conception de boutique (Achats in-app), afin de créer un bouton sur l'écran d'accueil, qui permet (contre la modeste somme de 1,20€) de supprimer définitivement toute publicité. Grâce à l'utilisation du plug-in, l'achat est simplifié, puisqu'il utilise directement les systèmes de paiement d'Android et d'iOS.

L'idée n'étant pas d'escroquer les joueurs, simplement d'amortir les coûts de mise en ligne de l'application (voir "Mise en ligne"). Le plug-in utilisé est Soomla Store, un plug-in intéressant, puisqu'il est développé de manière collaborative par qui veut apporter sa contribution, et qu'ils sont connus pour de nombreux plug-ins Unity très intéressants.

4.8 Mise en ligne

La mise en ligne d'une application tant sur Android que sur iOS n'est pas gratuite. Les deux systèmes utilisent la même formule, à savoir l'achat d'un compte développeur qui permet de mettre autant d'application souhaitée, pendant un an. Une fois une application mise en ligne, elle reste disponible à vie, mais passé les un an, on ne peut ni rajouter d'application ni faire de mise à jour (à moins de renouveler le compte développeur). Le prix étant de 25$ pour Android, et de 99$ pour iOS ! À noter aussi que pour Android le compte est nécessaire uniquement pour la publication, alors que chez iOS, il est impossible de tester la moindre application en cours de développement sans payer.

Chez Android, la publication d'une application est assez simple, il suffit de compiler l'application et d'uploader le fichier apk obtenu. Une ou deux heures après, l'application est disponible sur le store. Pour iOS, c'est un peu plus fastidieux, chaque application/ou mise à jour est scrupuleusement vérifiée. Ainsi, les délais de publication (à condition d'être accepté) peuvent durer selon les périodes entre 5 et 14 jours ! Il faut donc être bien prudent avant de publier une mise à jour, la moindre erreur laissée pouvant être extrêmement longue à corriger.

De plus, alors qu'une application Android peut être développée à partir de n'importe quel système, Apple étant un peu plus fermé, nécessite l'utilisation d'un Mac, sans quoi ce n'est pas possible. Fort heureusement nous en possédions un, nous permettant d'éviter l'installation d'un émulateur Mac sur PC (contournement possible). Enfin, alors que la compilation Android est extrêmement simple, pour iOS c'est encore un peu compliqué, puisqu'il nécessite l'ajout de bibliothèques externes, qui parfois comportent quelques conflits entre elles.

4.8.1 Problème de synchronisation

Lors de nos premiers tests sur mobile, nous nous sommes aperçus un peu tard de très gros problèmes de synchronisations. Quand le joueur frappait sur l'écran, il y avait un décalage entre le moment où il appuyait sur l'écran et le moment ou les actions se jouaient. Si au début on pensait que ce n'était qu'une question d'optimisation de code, ou une habitude à prendre, il s'est avéré que c'était un très gros problème dans la jouabilité.

Après plusieurs heures de recherche sur les éléments de notre création qui pouvaient provoquer ces décalages, nous avons terminé par créer une simple démo qui joue un son quand on tape l'écran. La conclusion était effrayante : le décalage était réel (entre 300 et 500ms environ). Le problème nous semblait venir de Unity, il ne serait peut-être pas adapté pour les jeux de rythme de précision. (Un peu tard à ce stade du développement).

Finalement nous avons terminé par trouver la raison du problème : Une simple case à cocher dans l'une des centaines de menus de Unity. Le moteur devient optimisé pour jouer les sons rapidement, et tous les problèmes de synchronisation disparaissent. Ce n'est qu'un simple exemple démontrant l'importance de la configuration de Unity pour l'exportation d'un jeu optimisé sur toutes les plateformes.

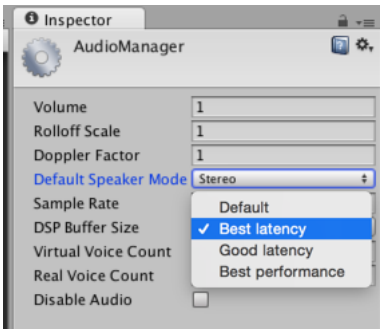


FIGURE 4.25 – Paramètre pour la synchronisation des sons

4.8.2 Optimisation du poids de l'application

Une application ne doit pas avoir un poids trop élevé si elle veut rester cohérente sur une vitrine. Pour cela, Unity propose plusieurs façons de compresser les fichiers composant l'application. Ainsi, pour réduire le poids des images utilisées dans les *sprites*, il est possible de les compresser en diminuant leur taille initiale ou bien en choisissant un algorithme de compression différent.

Bien entendu cela impacte la qualité de jeu de l'utilisateur sur des périphériques à grand écran comme les tablettes ou les derniers smartphones, c'est pour cela que Unity propose d'appliquer les compressions voulues seulement sur un déploiement en particulier. Par exemple, il est possible de compresser les textures sur les systèmes Android et iOS, mais de garder les textures en haute définition pour un déploiement PC.

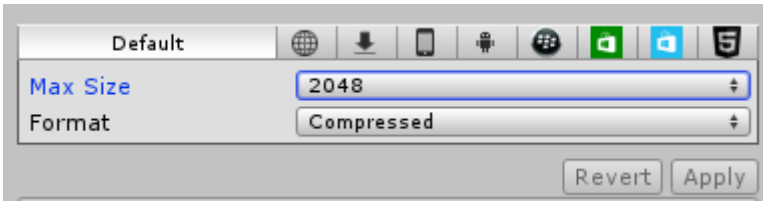


FIGURE 4.26 – Aperçu des réglages de compression des textures

Chapitre 5

Résultat

L'application que nous avons publiée est l'aboutissement du projet. Elle contient de nombreuses fonctionnalités, qui forment une base permettant l'ajout de nouveau contenu de façon simple. Dans la plupart des rapports, une partie est dédiée à un mode d'emploi. Il était important pour nous qu'il n'y ait pas besoin de lire un guide avant de jouer, cette partie n'est donc pas nécessaire. Nous avons préféré présenter les résultats obtenus.

L'écran d'accueil Le principe étant d'avoir un écran principal permettant d'accéder aux fonctionnalités principales de l'application. Il possède ainsi le titre de l'application, un bouton qui permet d'avoir accès à la grille des niveaux, un bouton permettant d'enlever les publicités en échange de $1,20$ euros, et enfin, un bouton qui donne les crédits à l'auteur des musiques ainsi que les noms des développeurs.

FIGURE 5.1 – L'écran d'accueil de l'application

Choix des niveaux Cet écran permet à l'utilisateur de sélectionner le niveau de son choix. Au démarrage, seul un niveau est actif, les autres sont grisés et inaccessibles. Pour les débloquer, le joueur doit réussir le niveau précédent.

FIGURE 5.2 – L'écran de choix du niveau

Ce menu permet aussi d'afficher le nombre d'étoiles obtenues sur chaque niveau. L'utilisateur peut ainsi facilement retrouver les niveaux qu'il n'a pas encore réussi à 100%.

Le menu pause À n'importe quel moment de n'importe quel mini-jeu, le joueur peut décider de mettre en pause sa partie. Ce menu permet de reprendre la partie au moment où le bouton pause a été pressé, de recommencer le jeu en cours depuis le début, ou bien de quitter le niveau pour revenir à l'écran de sélection.

L'écran de fin de partie À la fin d'un mini-jeu, l'application affiche le résultat de la partie, en attribuant au joueur une note en fonction de son degré de réussite sur le niveau, ainsi qu'une rapide appréciation. Il propose ensuite un bouton pour recommencer le niveau et un autre pour retourner à l'accueil (Voir figure 4.7).

Les mini-jeux

La marche de l'escargot Ce mini-jeu est proposé en premier, car sa difficulté est moindre. Il consiste à suivre le rythme de la musique et de taper en même temps que les coups de trompette qui battent la mesure. Le motif à jouer est répétitif et n'est donc pas compliqué à mémoriser.

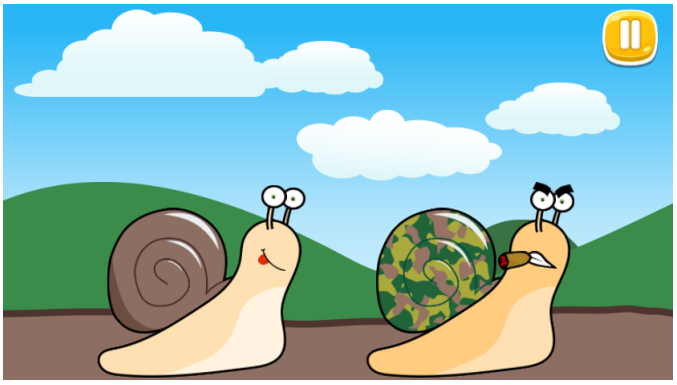


FIGURE 5.3 – Le niveau de la Marche des escargots

Lors de l'échec, un son différent est joué, et le sergent-escargot n'est pas content !

Le magicien La difficulté de ce niveau est plus élevée et repose sur l'action de maintenir le doigt sur l'écran pendant un temps précis. Lorsque l'objet apparait au-dessus du chapeau du magicien, le joueur doit appuyer sur l'écran pour l'abaisser dans le chapeau. En fonction de l'objet, il doit rester appuyé un certain temps, puis relâcher lorsque ce temps est terminé. S'il réussit, l'objet sera transformé en un autre contenant le même jeu de couleurs, sinon un escargot triste apparaitra et un son triste sera joué. La durée pendant laquelle il faut maintenir l'objet dans le chapeau est indiquée dans le tutoriel en début de partie.

FIGURE 5.4 – Le niveau du magicien

Ce niveau comporte des effets de particules pour renforcer l'aspect magique de l'ambiance.

Les champignons Malgré le fait que ce niveau ait été développé en premier, il se retrouve en dernier dans la liste des mini-jeux proposés. Cela est dû à sa difficulté qui est élevée. En effet, pour le réussir, il faut avoir déjà pris l'habitude de battre le rythme sur son smartphone au travers des autres jeux. De plus l'attention du joueur doit être totale pendant toute la durée du jeu.

FIGURE 5.5 – Le niveau des champignons

Le principe consiste à répéter à l'identique l'action du papi champignon, après que celui-ci nous donne le signal. Le joueur est représenté par un champignon plus jeune, et donne ainsi l'impression de recevoir des instructions d'un grand sage champignon. Des champignons fans sont présents en arrière-plan et battent la mesure de la musique. La difficulté est croissante au fur et à mesure de l'avancée du niveau, ainsi au début du jeu seuls les temps pleins sont joués, puis on rajoute des demi-temps et même des quarts de temps. Le tutoriel permet de s'initier au jeu en douceur.

Chapitre 6

Bilan du projet

6.1 Autocritique

Nous avons pu valider quatre fonctionnalités parmi celles qui étaient prévues au départ dans le cahier des charges.

— Créer le moteur de jeu de rythme avec Unity, et un jeu l'utilisant
— Rendre l'application jouable sur tous les supports
— Créer d'autres mini-jeux utilisant le moteur
— Créer un moteur de tutoriels, et créer les tutoriels pour les jeux

Les trois premières fonctionnalités sont les plus importantes et donc nécessaires à la réussite du projet. Elles ont été implémentées avec succès.

Cependant, plusieurs idées initiales ont dû être abandonnées en cours de projet. Le principe de la "planète d'accueil" que le joueur devait personnaliser grâce aux objets qu'il aurait gagnés dans les mini-jeux, ainsi que le niveau qui se joue à l'infini, n'ont pas étés développés. En effet, après avoir réalisé l'ampleur de la tâche que représentait la création d'un seul mini-jeu, de sa conception, jusqu'à son intégration finale avec le moteur, nous avons décidé de nous impliquer pleinement dans ce processus afin d'obtenir des mini-jeux ergonomiques et cohérents pour un utilisateur. C'est ainsi que plusieurs prototypes de mini-jeux n'ont pas dépassé le stade de la conception, car jugés peu compréhensibles pour le type de gameplay facile et amusant que nous désirions.

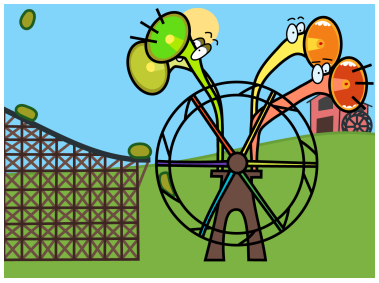


FIGURE 6.1 – Concept du jeu du moulin

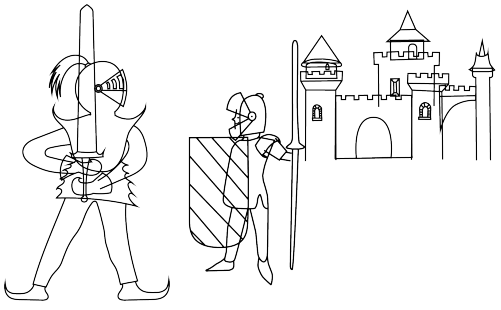


FIGURE 6.2 – Concept d'un jeu de chevaliers

Le principal objectif que nous nous étions fixé au sein du groupe était d'avoir une application fonctionnelle, propre, bien codée et en ligne. Cet objectif est accompli, l'application

est fonctionnelle et présente sur les marchés d'applications pour les systèmes Android et Apple.

Nous sommes globalement satisfaits de notre gestion du temps, et nous pensons que notre organisation y a beaucoup contribué. L'expérience de créer un jeu de A à Z nous a beaucoup appris et nous a permis de mettre en oeuvre beaucoup de concepts sur la gestion de projets.

6.2 Enseignements tirés

Les difficultés que nous avons rencontrées tout au long du développement de l'application nous ont permis de tirer des leçons et de gagner de l'expérience.

Dans un premier temps, la maitrise de l'environnement d'Unity nous a demandé beaucoup de temps, et pour cela nous avons créé de nombreuses scènes de test au démarrage du projet afin de pouvoir nous familiariser avec l'outil, avant de commencer à travailler sur le véritable projet. Malgré ces précautions, nous avons rencontré quelques difficultés sur le développement principal. Ainsi, en travaillant en collaboration, via GitHub, nous avons constaté qu'il est impossible de travailler à deux en même temps sur la même scène, Unity ne sachant pas gérer la fusion de deux scènes. Pour résoudre ce problème, nous avons dû travailler sur des copies de scènes en leur donnant des noms de version. D'autres difficultés liées à Unity sont apparues, comme la non-compatibilité entre les versions ou la nécessité d'utiliser Windows. De plus, Unity est un logiciel demandant beaucoup de ressources, ce qui a posé des petits problèmes d'efficacité lorsque nous nous retrouvions en groupe pour travailler, avec des ordinateurs portables aux capacités de calculs faibles pour travailler efficacement.

Du point de vue du travail en équipe, nous avons dû apprendre à mieux connaitre les capacités de chacun afin d'être efficaces sur l'organisation. L'avantage d'avoir passé du temps à analyser le projet avant de commencer à le développer, est que nous avons pu choisir les activités sur lesquelles nous étions chacun le plus performant.

Ce projet a été un challenge pour nous quatre, qui nous a conduits à un véritable enrichissement.

6.3 Perspectives

Ce projet peut avoir un avenir et continuer à être développé. La base étant créée, il est possible d'y ajouter autant de mini-jeux que nécessaire, il suffit de suivre simplement la démarche que nous avons appliquée pour créer chaque niveau. Il est aussi possible de rajouter de nouvelles fonctionnalités, telles que des options communautaires, ou la possibilité d'ajouter du contenu personnalisé créé par les joueurs. Néanmoins, l'ajout de telles fonctionnalités nécessite d'avoir un contenu de base solide.

Ainsi, en fournissant un travail conséquent, l'application peut avoir la possibilité de devenir virale, ce qui nous pousserait à développer un système monétaire en jeu, incitant les joueurs à dépenser de l'argent réel pour obtenir un avantage, ou une apparence différente.

Les perspectives de développement de cette application sont donc multiples, et l'expérience acquise durant ce projet nous permet d'affirmer qu'elles sont réalisables dans des temps honnêtes.

Chapitre 7

Conclusion

La création d'un jeu vidéo de bout en bout demande énormément de temps et une grande motivation. Nous en étions conscients en nous lançant dans cette tâche que nous avons pris beaucoup de plaisir à accomplir. Seulement l'ambition ne suffit pas pour aboutir et peut conduire facilement à un échec si elle est trop grande.

La réalisation de ce premier jeu nous a permis de découvrir toutes les facettes qui composent un tel projet, ainsi que leur lot de problèmes. Nous avons consacré beaucoup de temps à apprendre à utiliser les différents outils nécessaires, comme la mise en place d'une boutique ou la publication sur des marchés, que nous pensions annexes en début de projet.

Nous retenons de ce projet que la création (conception et réalisation) du contenu pour un jeu prend autant, voire plus de temps, que le développement pur du jeu, surtout lorsque l'on utilise un moteur de jeu comme Unity. Il est très important de prendre cela en compte lors de l'analyse, par rapport aux capacités de chaque membre d'une équipe.

Après avoir passé un grand nombre d'heures sur le moteur Unity, nous ne regrettons pas de l'avoir choisi. Il est de plus en plus utilisé sur le marché du mobile, et nous avons maintenant, grace à cette riche expérience, de très bonnes bases avec cet outil.

Le travail de groupe a été une réussite par une bonne organisation et une bonne méthode de travail accompagnés d'outils performants. Les capacités de chacun ont été pleinement exploitées.

Le jeu est maintenant en production et jouable sur la plupart des téléphones du marché. Nous considérons que nous avons atteint l'objectif, et nous prendrons maintenant plaisir à analyser les statistiques et les retours des utilisateurs.

Chapitre 8

Annexes

Timer.cs

```
using UnityEngine;
using System.Collections;

public abstract class Timer : HelikoObject {

        public bool startCountAtLoad = true;
        protected bool stopIt = false;
        private bool loop;

        public AudioSource audioSource;
        public float loopTime = 30f; // Temps d'attente entre chaque boucle en MS

        private float nextBeatSample; // Le numéro du prochain sample á attendre
            pour un nouveau beat
        protected float samplePeriod; // Le temps en samples d'un beat
        protected float sampleDelay; // Le temps d'attente en sample avant de
            compter le premier beat

        protected MusicTempo music;

        private int myMsDelayStartCount = 0;
        private int myDelayTicks = 0;
        private int nBeat = 0;

        // Methode á implémenter, décrira le comportement lors d'un beat
        protected abstract void beat();

        // Methode á implémenter, décrira le comportement á la fin de la musique
        protected abstract void endMusic();

        protected void setSampleDelay(int msDelayStartCount, int delayTicks) {
                myMsDelayStartCount = msDelayStartCount;
                myDelayTicks = delayTicks;
        }

        public MusicTempo getMusic() {
                return music;
        }

        // Retourne le numéro du beau
        public int getNBeat() {
                return this.nBeat;
```

```
        }

        public new void Start() {
                if (isStart) return;

                base.Start();

                if (constantes.instantCalcul)
                loopTime = 0;

                music = audioSource.GetComponent<MusicTempo>();

                if (music == null) {
                        Debug.LogError ("MusicTempo component not found");
                        return;
                }

                float delayMS = myMsDelayStartCount +
                    (myDelayTicks*music.GetTimePeriod()*1000f);

                sampleDelay = ((float) delayMS / 1000f) * music.GetFrequency();
                samplePeriod = music.GetSamplePeriod();
                nextBeatSample = sampleDelay;

                if (startCountAtLoad) {
                        StartCount();
                }
        }

        // Lance le beater
        public void StartCount() {
                audioSource.Play();
                StartCoroutine(BeatCheck());
        }

        // Stop complétement le thread de comptage
        public void stopCount() {
                this.stopIt = true;
        }

        // Remet la musique au début en réinitialisant toutes les valeurs
        public void reset() {
                Debug.Log("reset!");
                audioSource.Stop();
                audioSource.Play();
                nBeat = 0;

                float delayMS = myMsDelayStartCount;
                sampleDelay = ((float) delayMS / 1000f) * music.GetFrequency();
                samplePeriod = music.GetSamplePeriod();
                nextBeatSample = sampleDelay;
        }

        // Active ou desactive le bouclage de la musique
        public void setLoop(bool loop) {
                this.loop = loop;
        }

        // Thread principal qui se chargera d'envoyer les beats au bon moment
        IEnumerator BeatCheck () {
                while (!stopIt) {
```

```
                        if (audioSource.isPlaying) {
                                float currentSample = audioSource.timeSamples;
                                if (currentSample >= (nextBeatSample)) {
                                        this.beat();
                                        nBeat++;
                                        nextBeatSample += samplePeriod;
                                }
                        }
                        if (audioSource.timeSamples + 1000 >=
                            audioSource.clip.samples) {
                                if(loop) {
                                        reset();
                                } else {
                                        this.endMusic();
                                }
                        }
                        yield return new WaitForSeconds(loopTime / 1000f);
                }
        }
}
```

BeatCounter.cs

```
using UnityEngine;
using System.Collections;

public class BeatCounter : Timer {

        public int delayInMS = 0; // Délai avant de lancer le compteur
        public int nbrTicksDelay = 0; // Nombre de beat á attendre avant de
            commencer á compter

        void Awake() {
                this.observers = new ArrayList ();
                this.setSampleDelay(delayInMS, nbrTicksDelay);
        }

        /**
        * @return L'écart avec le beat le plus proche
        */
        public float getScore() {
                float currentSample = audioSource.timeSamples - sampleDelay;
                float score = currentSample % samplePeriod;
                if (score > samplePeriod / 2)
                return Mathf.Abs(score - samplePeriod);
                else
                return score;
        }

        /**
        *       @return Le numéro du step le plus proche á ce temps
        */
        public int getStepClosest() {
                float currentSample = audioSource.timeSamples - sampleDelay;
                float score = currentSample % samplePeriod;
                int step = (int) (currentSample / samplePeriod);
                if (score > samplePeriod/2) {
                        step++;
                }
                return step;
```

```
        }

        /**
        *       @action Notifie les enfants connectés qu'il y a eu un beat
        */
        protected override void beat() {
                this.NotifyChildren();
        }

        /**
        *       @action Notifie les enfants connectés que la musique est terminée
        */
        protected override void endMusic() {
                this.NotifyChildrenEndMusic();
        }

        /**
        *       @return Si la musique est en pause
        */
        public bool isInPause() {
                return stopIt || !this.audioSource.isPlaying;
        }


        // SYSTEME DE NOTIFICATION
        ArrayList observers;

        public void Connect (TempoReceiver r) {
                this.observers.Add (r);
        }

        private void NotifyChildren () {
                foreach (TempoReceiver e in this.observers) {
                        e.OnStep (this.getNBeat());
                }
        }

        private void NotifyChildrenEndMusic () {
                foreach (TempoReceiver e in this.observers) {
                        e.OnEndMusic();
                }
        }

}
```

Extrait de la table de code midi

Table 1: MIDI 1.0 Specification Message Summary

Status D7----D0	Data Byte(s) D7----D0	Description
Channel Voice Messages [nnnn = 0-15 (MIDI Channel Number 1-16)]		
1000nnnn	0kkkkkkk 0vvvvvvv	Note Off event. This message is sent when a note is released (ended). (kkkkkkk) is the key (note) number. (vvvvvvv) is the velocity.
1001nnnn	0kkkkkkk 0vvvvvvv	Note On event. This message is sent when a note is depressed (start). (kkkkkkk) is the key (note) number. (vvvvvvv) is the velocity.
1010nnnn	0kkkkkkk 0vvvvvvv	Polyphonic Key Pressure (Aftertouch). This message is most often sent by pressing down on the key after it "bottoms out". (kkkkkkk) is the key (note) number. (vvvvvvv) is the pressure value.
1011nnnn	0ccccccc 0vvvvvvv	Control Change. This message is sent when a controller value changes. Controllers include devices such as pedals and levers. Controller numbers 120-127 are reserved as "Channel Mode Messages" (below). (ccccccc) is the controller number (0-119). (vvvvvvv) is the controller value (0-127).
1100nnnn	0ppppppp	Program Change. This message sent when the patch number changes. (ppppppp) is the new program number.
1101nnnn	0vvvvvvv	Channel Pressure (After-touch). This message is most often sent by pressing down on the key after it "bottoms out". This message is different from polyphonic after-touch. Use this message to send the single greatest pressure value (of all the current depressed keys). (vvvvvvv) is the pressure value.
1110nnnn	0lllllll 0mmmmmmm	Pitch Bend Change. 0mmmmmmm This message is sent to indicate a change in the pitch bender (wheel or lever, typically). The pitch bender is measured by a fourteen bit value. Center (no pitch change) is 2000H. Sensitivity is a function of the transmitter. (lllllll) are the least significant 7 bits. (mmmmmm) are the most significant 7 bits.

Table 2: Expanded Status Bytes List

STATUS BYTE		DATA BYTES	
1st Byte Value Binary |Hex| Dec	Function	2nd Byte	3rd Byte
10000000= 80= 128	Chan 1 Note off	Note Number (0-127)	Note Velocity (0-127)
10000001= 81= 129	Chan 2 Note off	Note Number (0-127)	Note Velocity (0-127)
10000010= 82= 130	Chan 3 Note off	Note Number (0-127)	Note Velocity (0-127)
10000011= 83= 131	Chan 4 Note off	Note Number (0-127)	Note Velocity (0-127)
10000100= 84= 132	Chan 5 Note off	Note Number (0-127)	Note Velocity (0-127)
10000101= 85= 133	Chan 6 Note off	Note Number (0-127)	Note Velocity (0-127)
10000110= 86= 134	Chan 7 Note off	Note Number (0-127)	Note Velocity (0-127)
10000111= 87= 135	Chan 8 Note off	Note Number (0-127)	Note Velocity (0-127)
10001000= 88= 136	Chan 9 Note off	Note Number (0-127)	Note Velocity (0-127)
10001001= 89= 137	Chan 10 Note off	Note Number (0-127)	Note Velocity (0-127)
10001010= 8A= 138	Chan 11 Note off	Note Number (0-127)	Note Velocity (0-127)

FIGURE 8.1 – Extrait des spécifications des évènements midi

Documentation

Tout au long de ce projet, nous avons eu recours à la lecture de divers tutoriels et forums pour parvenir à surmonter les différentes difcultés apparues au fur et à mesure du développement de l'application. Voici une liste des sites qui nous ont aidés durant le projet :

- La documentation officielle d'Unity : http ://docs.unity3d.com/
- Les tutoriels officiels d'Unity : http ://unity3d.com/learn/tutorials/
- Un tutoriel pour créer un jeu 2D simple avec Unity : http ://pixelnest.io/tutorials/2d-game-unity/
- La documentation officielle pour racket : http ://docs.racket-lang.org/
- L'outil *git-hours* pour estimer le temps passé sur le projet : https ://github.com/kimmobrunfeldt/git hours
- Le site de Questions-Réponses stackoverflow : http ://stackoverflow.com/

Bibliographie

- **Defining Game Mechanics**, by Miguel Sicart
- **Sonic Mechanics : Audio as Gameplay**, by Aaron Oldenburg
- **Subjective Measures of the Influence of Music Customization on the Video Game Play Experience : A Pilot Study** by Alexander Wharton, Karen Collins
- **Play Along - An Approach to Videogame Music** by Zach Whalen

Résumé

Ce rapport est le compte rendu du projet Oufmania, exécuté par les auteurs et encadré par Mathieu Lafourcade pour l'unité d'enseignement GMIN250 du premier semestre de master Informatique de la Faculté de Sciences de Montpellier en 2014-2015. Il décrit la conception et la réalisation d'un jeu mobile en 2D en utilisant le moteur de jeu Unity. Une partie analyse commence par décrire le contexte et rendre compte de l'existant tout en parlant des objectifs et en fixant le cahier des charges. Un rapport d'activité décrit ensuite les méthodes de travail et un rapport technique se charge de mettre en avant la conception, l'architecture et la mise en oeuvre de ce projet. Pour finir, il fait état du résultat final de l'application et de son bilan.

Mots clés : Unity, musique, mobile, rythme, jeu

Abstract

This report is an account of the Oufmania project, implemented by its authors and supervised by Mathieu Lafourcade for the first semester of course GMIN250 for the computer science degree for the Faculty of Montpellier in 2014-2015. It describes the design and implementation of a mobile phone 2D game using the game engine Unity. An analysis part describes the context and the already existing work while talking abount our goals and setting the specifications. An activity report then describes our work methods and a technical report will complete this report by showing the design, architecture and implementation of our project.

Mots clés : Unity, music, mobile phone, rhythm, game