«««< HEAD ======= »»»> 16feeb9715e74d0535cda16114d568ab1c4b3b0a «««< HEAD ======= »»»> 16feeb9715e74d0535cda16114d568ab1c4b3b0a

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.

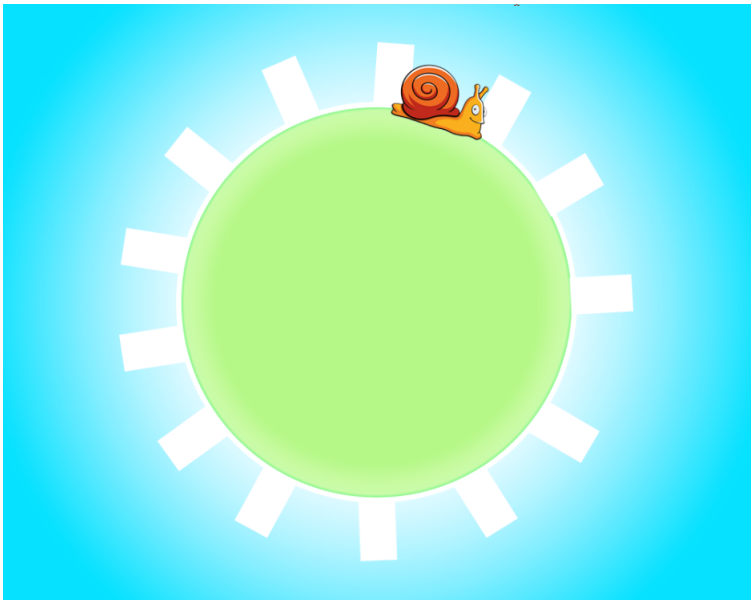


FIGURE 2.1 – Prototype de planète à remplir

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.

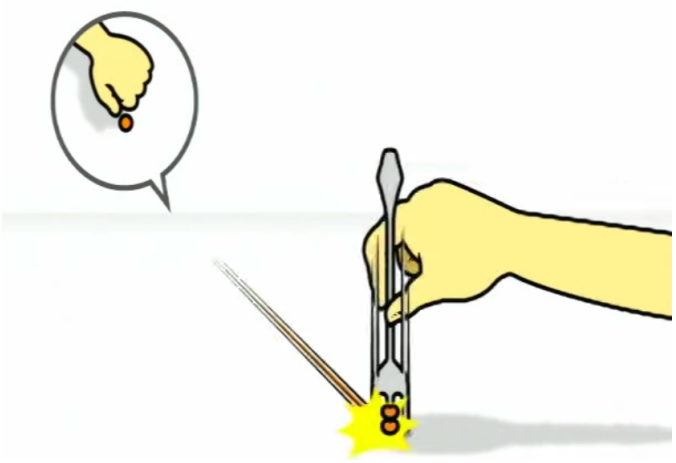


FIGURE 2.2 – Exemple de mini jeu, présent dans le jeu "Rhythm Paradize"

Sur la figure **??**, 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 **??** 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).

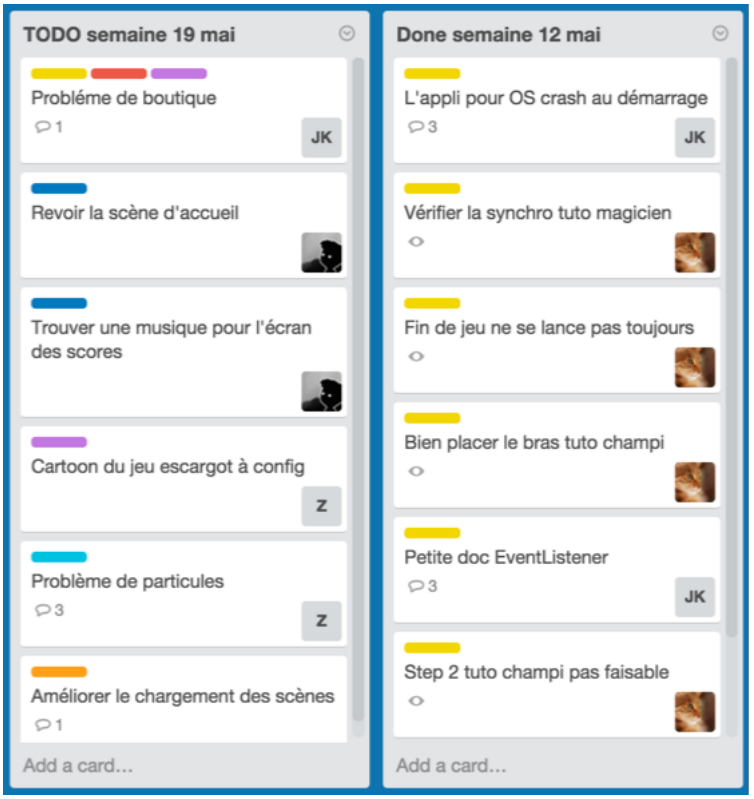


FIGURE 3.2 – Exemple de colonnes en fin de projet

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

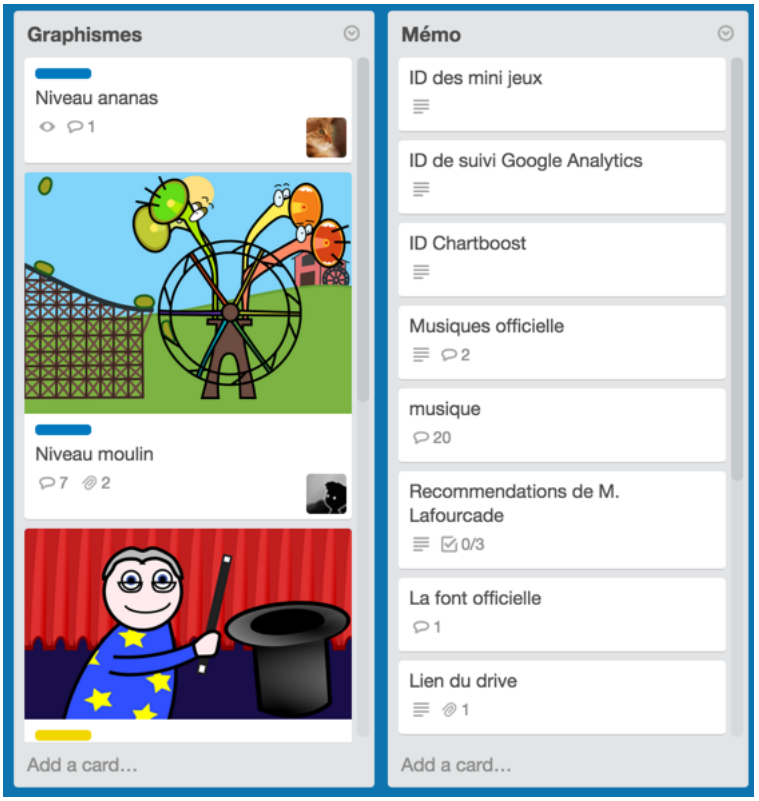


FIGURE 3.3 – Utilisation de Trello comme mémo et archives

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...).

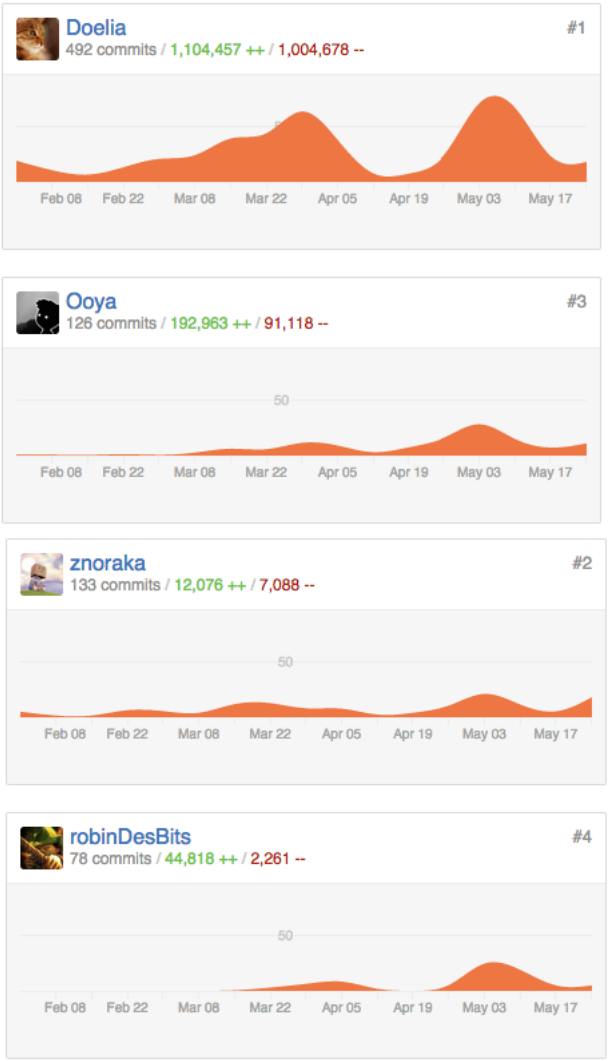


FIGURE 3.4 – Répartition des commits dans le temps

On remarque d'après la figure **??** 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

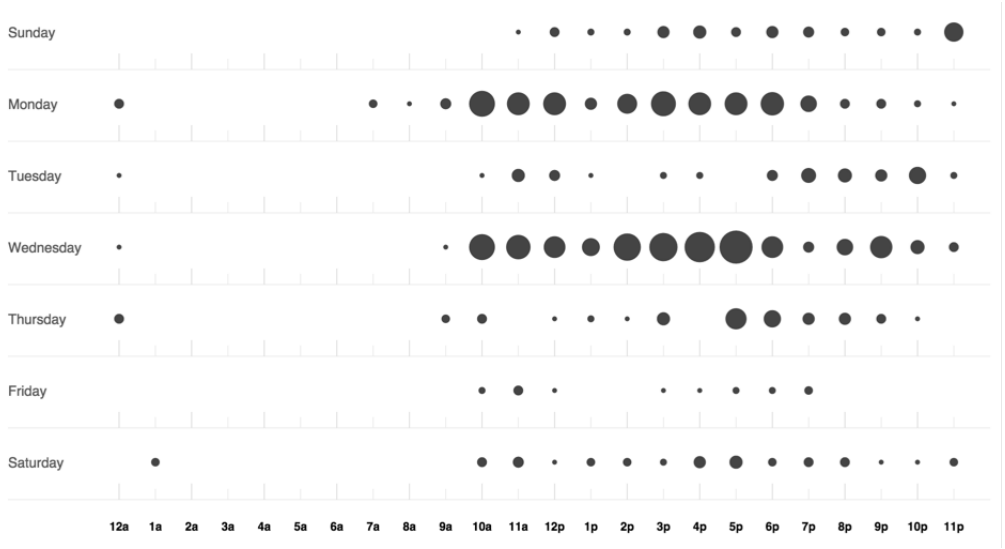


FIGURE 3.5 – Punch card de Github

D'après les statistiques *Github* (figure **??**), 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 **??**) 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%).

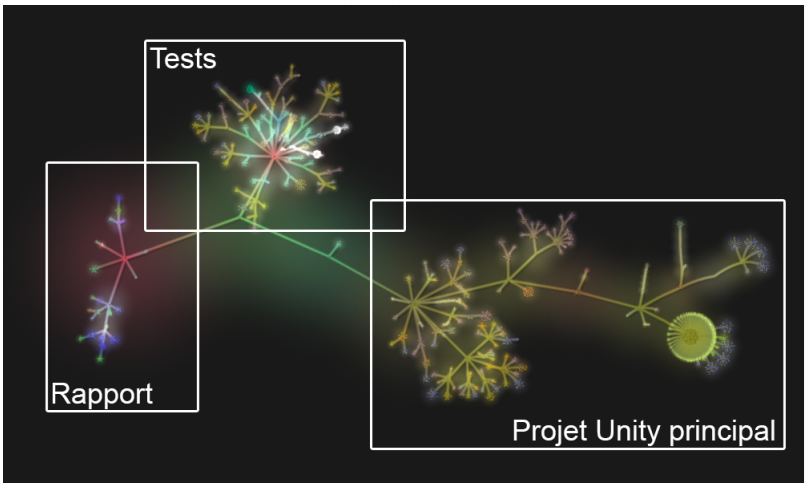


FIGURE 3.6 – Modélisation Gource en fin de projet

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.

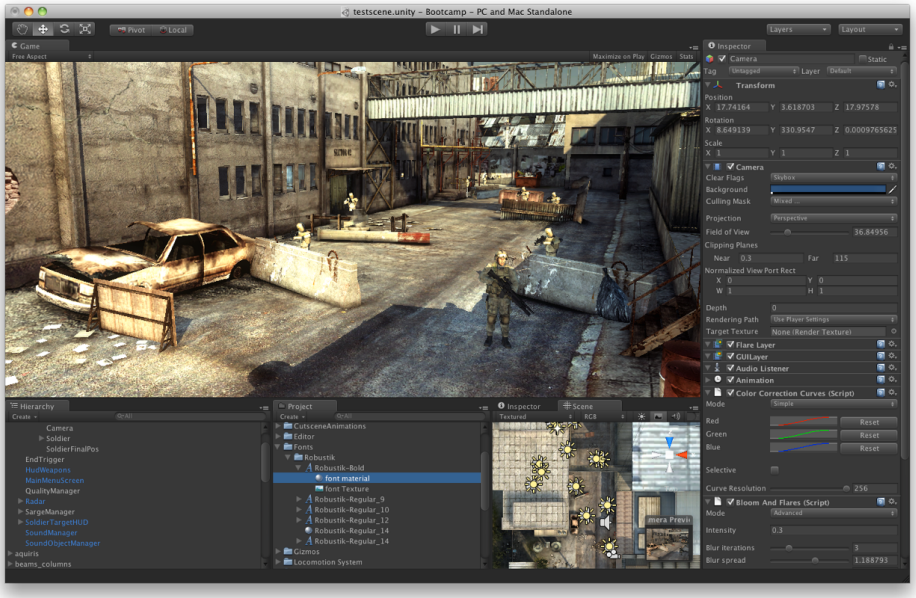


FIGURE 3.7 – Interface de Unity

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).

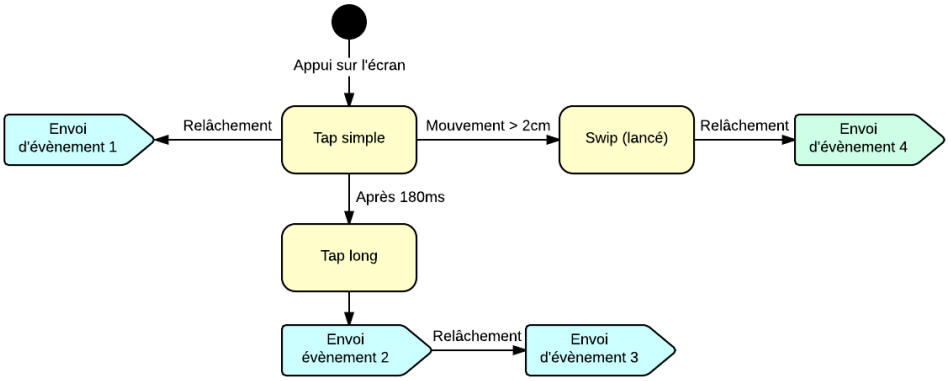


FIGURE 4.2 – Diagramme d'états-transitions des évènements

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.

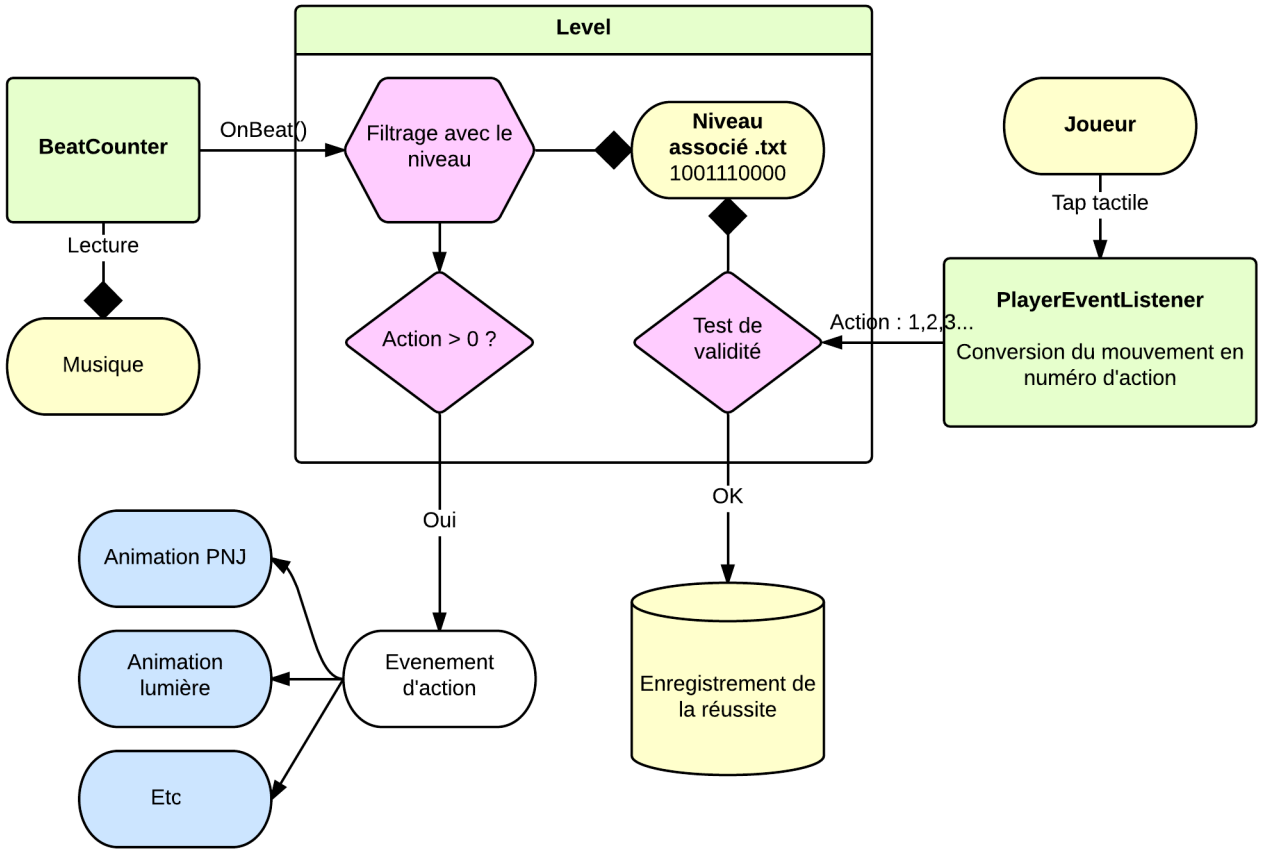
La phase complexe est de déterminer quand un évènement est trop tôt ou trop tard. Un joueur ne frappera jamais pile au moment réel de l'évènement : il faut mettre en place un système de tolérance.

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 **??** :

- 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