Projet L1 – C.M.I.
Spécialité informatique

Skizzle

Maëlle Beuret
Rémi Cérès
Mattéo Delabre

Année : 2015 – 2016
Soutenu le : 29/04/2016

Réseau Figure
CURSUS MASTER EN INGÉNIERIE

Table des matières

Chapitre 1

Introduction

Chapitre 2

Organisation du projet

2.1 Organisation du travail

Chaque membre du groupe a travaillé en autonomie. Les réunions lors des séances prévues étaient consacrées aux explications sur le travail de chacun ainsi qu'à la répartition des tâches pour la semaine suivante.

Chacun d'entre nous était chargé de tâches spécifiques (voir figure 1). Certaines parties du développement nécessitaient plusieurs personne et étaient ainsi partagées entre certains membres du groupe. La conception et les tests des niveaux, le fond des menus ainsi que les décors furent réalisés par Rémi et Maëlle, et la gestion de projet par tout le groupe.

Tout au long de la réalisation du projet, nous communiquions par Skype afin de s'informer de l'avancement et réfléchir à des solutions lorsqu'un problème était rencontré.

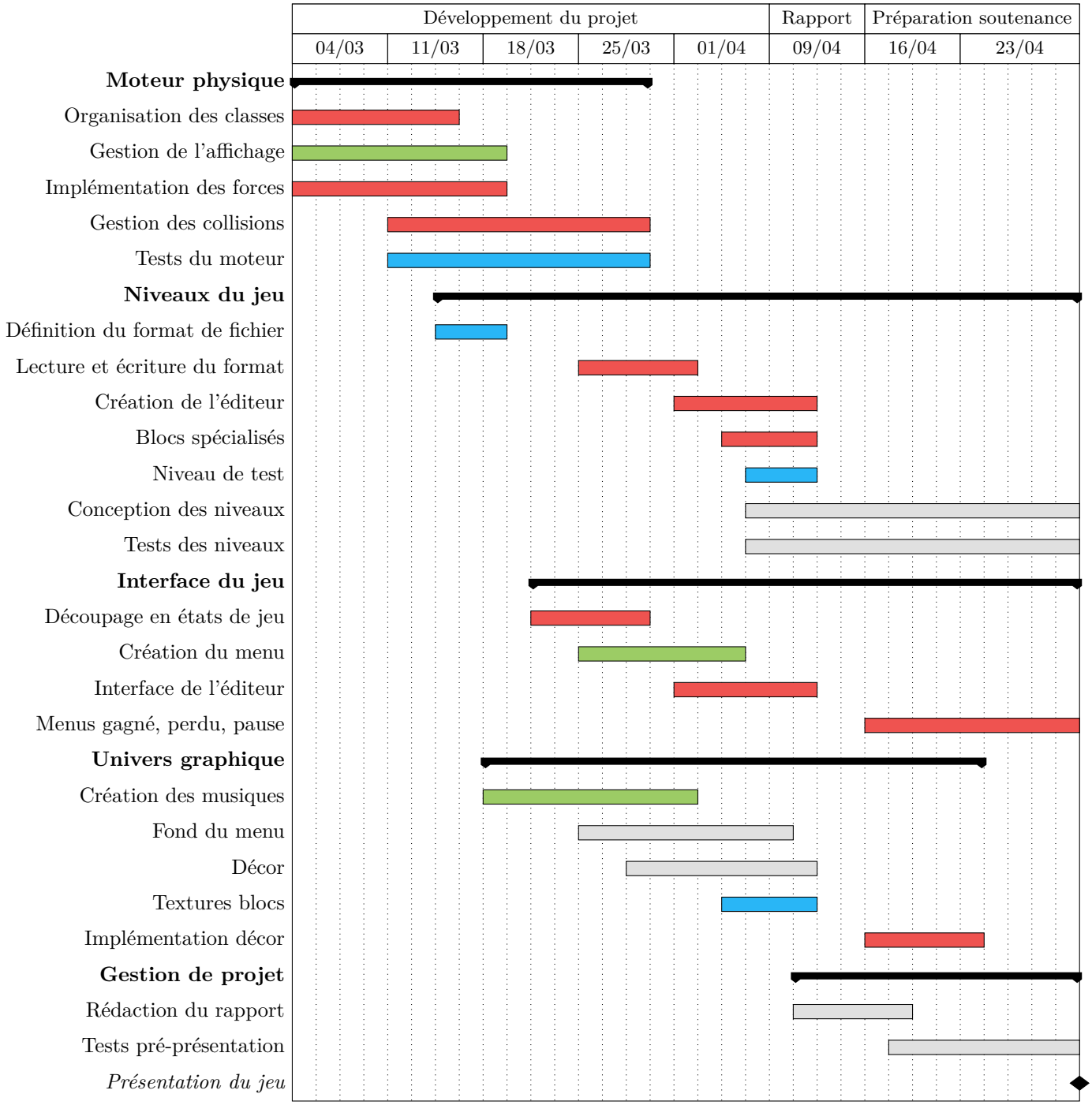

FIGURE 2.1 – Diagramme de la répartition des tâches. En vert, les tâches affectées à Maëlle ; en bleu, les tâches affectées à Rémi ; en rouge, les tâches affectées à Mattéo ; en gris, les tâches résolues en groupe

2.2 Outils de développement

Nous avons choisi le C++ tout d'abord car il s'agit du langage que nous apprenons cette année. De plus, il possède de nombreuses bibliothèques. C'est un langage très utilisé dont le code est élégant. Parmi les bibliothèques graphiques, nous avons choisi la SFML car son utilisation est simple et elle correspondait bien à nos besoins.

Pour écrire le code, nous avons utilisé différents éditeurs de texte (atom et gedit). Nous compilions notre programme avec g++. Pour faciliter la compilation, nous avons utilisé CMake.

Git nous a permis de gérer les versions du programme, avec le gestionnaire de projet GitHub, où nous déposions le code ainsi que les documents tels que le diagramme UML de classes permettant de s'y retrouver plus facilement dans les nombreuses classes que nous avons créées.

Chapitre 3

Analyse du projet

3.1 Découpage du projet

Le jeu est constitué d'une suite de niveaux organisés de manière semblable à ceux d'un jeu de plateformes. Ces niveaux contiennent des entités.

Les entités du jeu sont multiples : blocs, blocs spéciaux, joueurs, éléments de décor. Ces entités – ou objets, interagissent entre elles par un certain nombre de phénomènes physiques « naturels ». Pour répondre à ce besoin, **un moteur physique** est nécessaire. Le moteur physique est chargé de gérer les forces s'appliquant aux objets du jeu, de répondre aux collisions entre objets et de faire évoluer les objets en conséquence des forces qui leur sont appliquées.

Plusieurs moteurs physiques en 2D existent déjà dans le langage que nous avons choisi, notamment Box2D. [1] Nous avons choisi d'implémenter le moteur physique du jeu par nous-mêmes pour répondre aux besoins particuliers (notamment la force d'attraction) et car cela nous permet de mettre en pratique les savoirs acquis au premier semestre dans le module de physique générale.

Les **niveaux du jeu** sont constitués de ces entités et d'autres métadonnées. Pour pouvoir éditer les niveaux, les sauvegarder et y rejouer plus tard, il est nécessaire de pouvoir les stocker en dehors de la mémoire. Nous avons pour ce faire choisi de définir un format de fichier binaire permettant leur stockage sur le disque. Des fonctions pour coder et décoder ce format devront être écrites.

Skizzle propose différents **états de jeu**, notamment, on peut à tout moment se trouver dans l'éditeur, dans le jeu en lui-même ou sur la vue des règles. Pour pouvoir accéder à ces états, nous devons créer un menu. L'ensemble des états du jeu doit être abstrait pour pouvoir être géré dans la classe principale. Certains états du jeu proposeront des éléments interactifs (boutons, barres d'outils, zones de texte) qui doivent être implémentés.

Enfin, les différents **objets du jeu** sont représentés à l'écran en dessinant des textures. Nous avons également choisi d'ajouter des musiques au jeu pour le rendre plus convivial. D'autres éléments graphiques doivent être créés, par exemple le fond du menu. Tous ces éléments sont regroupés dans l'univers graphique du jeu.

3.2 Découpage du code

Nous avons choisi d'organiser notre code selon le paradigme objet. La plupart du code est sorti en dehors du `main`, dont la seule fonction est d'instancier la classe `Manager` qui gère de manière abstraite le jeu et de démarrer le premier état du jeu : le menu.

3.2.1 États, gestion des états et des ressources

Un état du jeu modélise un écran pouvant être affiché. Une classe abstraite `State` chapeaute toutes les classes d'états et permet de requérir l'implémentation d'une interface commune :

— `enable()` : cette méthode initialise l'état avant qu'il commence à être affiché. L'état implémentant cette méthode doit mettre en place les ressources globales utilisées comme la lecture de la musique, le titre de la fenêtre, les éléments de l'interface quand cette méthode est appelée ;
— `processEvent(event)` : cette méthode est appelée avec un événement lorsque celui-ci est extrait par la SFML lors de la boucle principale. L'état est censé décider s'il souhaite traiter cet événement et, si oui, modifier ses variables en conséquence ;
— `frame()` : cette méthode est appelée lorsque l'état doit dessiner une frame à l'écran. Pour éviter d'encombrer la boucle principale, l'état doit dessiner sa frame le plus rapidement possible.

Les états suivants sont implémentés et descendent de la classe `State` : `Rules` pour afficher les règles du jeu, `Menu` pour afficher le menu du jeu, `Level` pour afficher les niveaux (soit l'éditeur, soit le jeu en lui-même).

On définit `Manager` la classe qui gère les éléments principaux du jeu. Notamment, `Manager` maintient une pile d'états qui est initialisée contenant une seule instance de la classe `Menu` et peut être empilée ou dépilée par les états. Par exemple, le menu peut empiler un nouvel état instance de `Rules` pour « démarrer » la vue affichant les règles. En tout temps, l'état en haut de la pile est celui qui est actif (il reçoit les événements et est dessiné).

La librairie SFML permet de charger les ressources comme la musique, les images et les polices. Cependant, recharger ces ressources à chaque utilisation serait inefficace. La classe `ResourceManager` permet de mutualiser ces ressources : les états lui demandent les ressources à obtenir et le gestionnaire de ressources s'arrange pour ne charger la ressource qu'à la première demande et à la garder en mémoire par la suite. Le gestionnaire des ressources mutualise l'accès aux polices, textures et à la lecture de la musique.

La figure 3.1 résume les classes de gestion d'états et de ressources présentées.

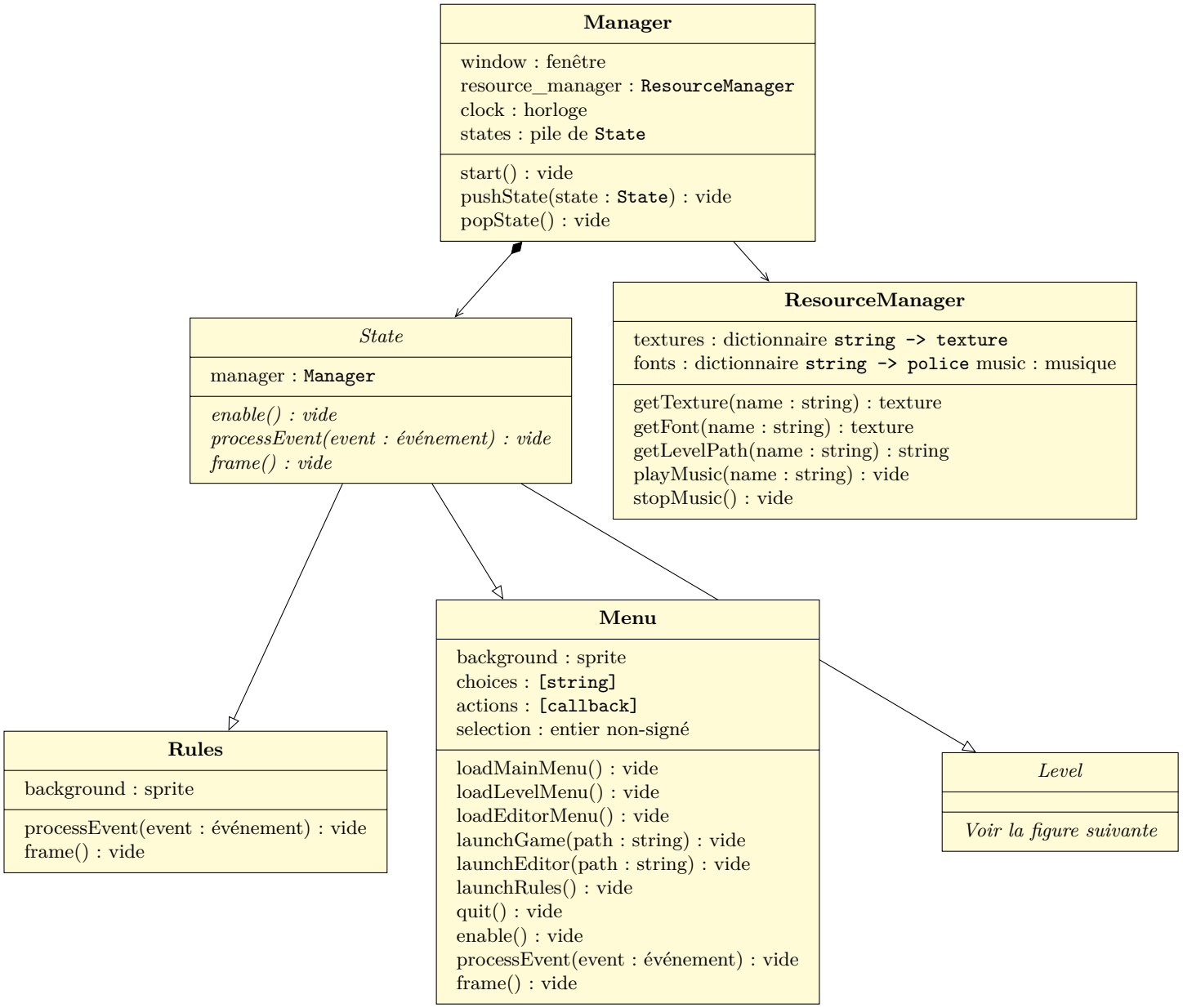

FIGURE 3.1 – Gestion des états et des ressources dans le jeu

3.2.2 Niveau et objets

La classe `Level` définit les niveaux, qui sont des collections d'objets. Elle définit la méthode pour dessiner tous les objets d'un niveau, le charger, le sauvegarder dans un fichier, ajouter ou supprimer des objets. Elle ne définit pas la méthode `frame()` que tous les états doivent implémenter, elle n'est donc pas un état en tant que tel.

Deux classes dérivent de `Level` : `Game` pour jouer aux niveaux et `Editor` pour les éditer. L'abstraction en `Level` permet d'éviter la duplication de code notamment en ce qui concerne la gestion des objets contenus.

Les classes de niveaux manipulent des collections d'objets. Les objets modélisent toutes les entités du jeu : les joueurs, les blocs et les blocs spéciaux. Une classe abstrait les fonctionnalités de tous les objets, `Object`.

Les classes `Block`, définissant l'apparence et le comportement des blocs, et `Player`, définissant l'apparence et le comportement des joueurs, descendent directement d'`Object`. Enfin, on définit des blocs spéciaux, qui peuvent réaliser des actions particulières :

— le bloc de gravité modifie la direction de la gravité dans un niveau lorsqu'une entité entre en contact avec lui. Il ne peut être activé qu'une seule fois par partie ;
— le bloc changeur échange la polarité de l'entité entrant en contact avec lui. Il ne peut être activé qu'une seule fois par partie ;
— le bloc tueur tue le joueur entrant en contact avec lui et fait perdre la partie (le niveau passe en mode « perdu ») ;
— le bloc d'arrivée tue le joueur entrant en contact et lorsqu'il ne reste plus de joueurs fait gagner la partie (le niveau passe en mode « gagné »).

La figure 3.2 résume les classes de niveaux et d'objets.

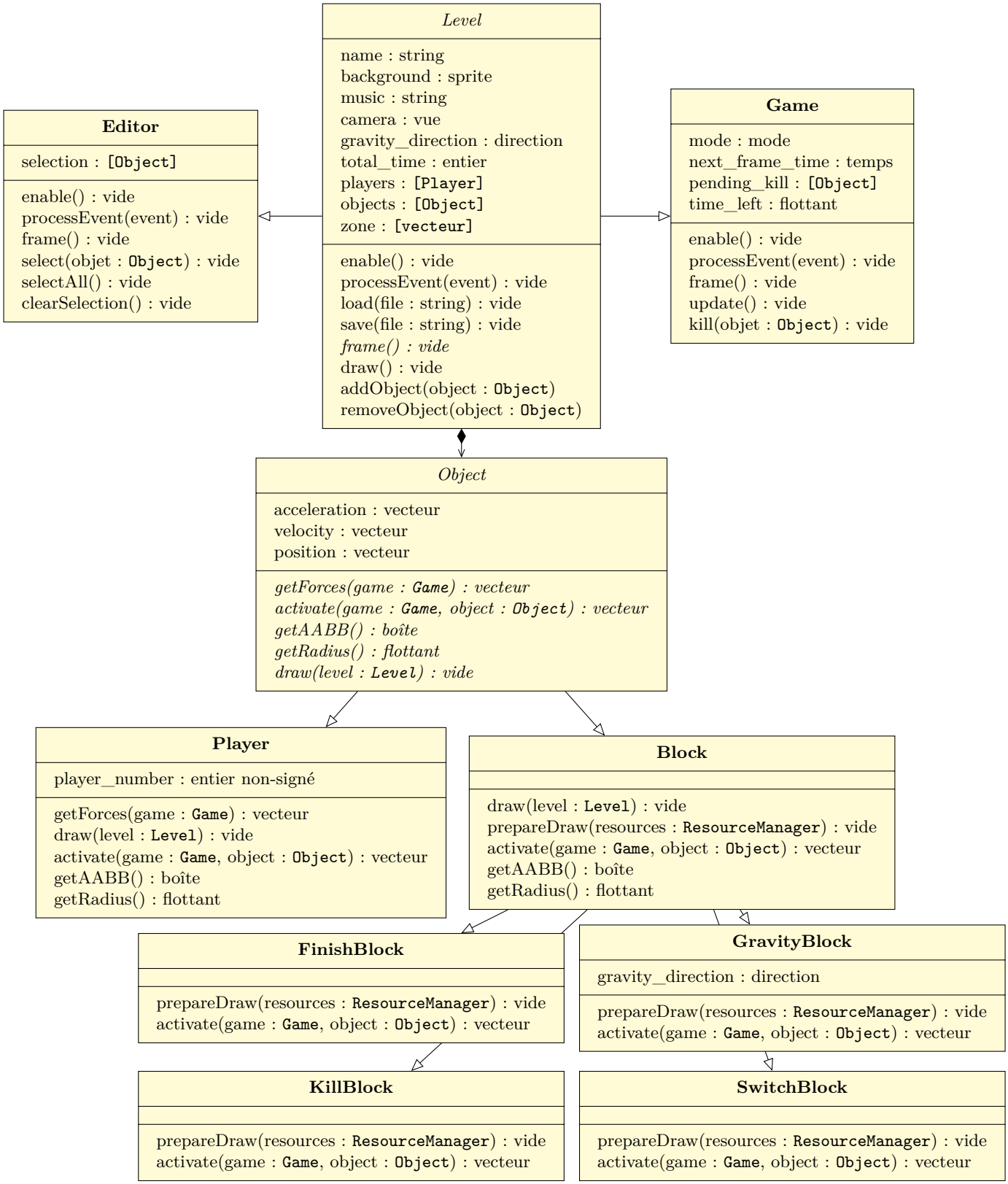

FIGURE 3.2 – Classes du niveau

Chapitre 4

Développement

4.1 Moteur physique

Cette partie du projet est basée sur l'algorithme du pré-rapport. La première tâche fut l'organisation des classes, réalisée par Mattéo à partir du diagramme de classe. Lors de cette première étape, le projet était constitué de trois classes : la classe Engine, contenant le moteur physique et gérant l'affichage du jeu, la classe Ball qui contenait les propriétés de l'objet ball, ainsi que la classe Block qui contenait les propriétés des blocs. Des classes furent rajoutées plus tard dans le développement du projet afin de mieux répondre aux besoins d'affichage et de moteur physique.

L'affichage fut géré par des fonctions draw() permettant de boucler sur les différents objets afin de mettre à jour le dessin affiché en fonction de leur position. Il fallut également gérer la caméra avec l'objet view de la SFML, et l'adapter au redimensionnement de la fenêtre. Enfin, pour un meilleur rendu graphique, nous avons chargé et appliqué des textures aux objets (textures provisoires lors de cette étape). Pour cela, nous avons décidé de créer la classe ResourceManager afin de permettre de charger les ressources depuis tout dossier où se trouverait l'exécutable.

Lors de l'implémentation des forces et des collisions, nous avons rencontré quelques difficultés, notamment au niveau des collisions. En effet, grâce aux tests du moteur dans différentes situations, nous avons pu repérer des erreurs de collisions. Cela mena à la création d'une nouvelle classe Object réunissant les balles et les blocs dans un même tableau et définissant les propriétés communes des objets. Nous avons donc passé plus de temps que prévu sur cette partie du projet.

4.2 Niveaux du jeu

Tout d'abord, il fallut définir le format des fichiers de niveaux, puis lire et écrire le format. Afin de pouvoir créer des niveaux plus facilement, mais également permettre aux utilisateurs de jouer à leurs propres niveaux, nous avons choisi de créer un éditeur de niveau en mode graphique. Cela nous prit plus de temps que prévu, c'est pourquoi il nous restait peu de temps pour la création de niveaux. Nous avons donc réfléchi à des niveaux courts mais demandant de la réflexion afin de les rendre intéressants malgré le manque de temps.

4.3 Interface du jeu

Afin de mieux gérer l'affichage entre les menus, l'éditeur et les niveaux, nous avons décidé de découper le jeu en états. Nous avons créé une classe abstraite gérant tous ces états, ainsi qu'une classe par état : Menu pour le menu principal, Rules pour les règles du jeu, Editor pour l'éditeur de niveaux et Game pour le jeu, ainsi qu'une classe abstraite Level pour gérer les niveaux.

Le menu principal se basait ainsi sur ce découpage. Chaque option renvoyait à un état différent ou modifiait l'état du menu si cela envoyait sur un autre menu (choix de niveaux).

Interface de l'éditeur ?

Menus pause, gagné, perdu ?

4.4 Univers graphique

Nous avons décidé d'ajouter des musiques au jeu afin de le rendre plus vivant. Pour cela, nous avons fait appel à un étudiant de l'IUT de Montpellier, Maxime Petitjean, pour aider Maëlle à créer des musiques pour chaque niveau ainsi que pour le menu à l'aide du logiciel Ableton.

Nous avons également décidé de créer nos propres textures pour un rendu du jeu plus personnel. Après les dessins préalablement réalisés par Maëlle sur papier, Rémi s'est chargé de les numériser en repassant les contours pour créer une image vectorielle sur Inkscape. Puis Maëlle les a colorées avec le même logiciel, pendant que Rémi se chargeait des textures des blocs que nous avions au préalable définies à l'oral en groupe.

Nous n'avons pas eu le temps d'implémenter le décor avant le rendu du code, mais il est prévu que cela soit fait avant le passage à l'oral.

Chapitre 5

Manuel d'utilisation

Chapitre 6

Conclusion

Webographie

[1] Erin Catto. Box2d : A 2d physics engine for games. `http://goo.gl/uTnXH4`.