IUT CLERMONT AUVERGNE

Aurillac - Clermont-Ferrand - Le Puy-en-Velay
Montluçon - Moulins - Vichy

Création d'un jeu vidéo 2D – Département Informatique

Jérémy Tremblay
Antoine Viton
Ugo Vignon
Maxime Wissocq
Adrien Coudour

Année Universitaire 2021-2022

Nous autorisons la diffusion de notre rapport sur l'intranet de l'IUT.

Remerciements

Ce projet et ce compte-rendu sont le résultat d'une démarche appuyée par la contribution d'un grand nombre de personnes. Nous tenons sincèrement à les remercier.

Nous souhaitons exprimer notre plus profonde reconnaissance à la personne qui a accepté notre idée de projet et qui a souhaité l'encadrer, M. Laurent PROVOT. Ce sont ses conseils avisés, son appui éclairé et sa généreuse disponibilité tout au long de ce projet qui ont permis l'aboutissement de ce projet.

Nous tenions aussi à remercier tous les enseignants qui nous ont aidés directement ou indirectement à atteindre les objectifs de ce projet. Nous pensons plus particulièrement à M. Cédric BOUHOURS – chef du Département Informatique, Mme Neddra CHATTI, Mme Isabelle GOI, M. Nicolas RAYMOND, M. Olivier GUINALDO et M. Guenal DAVALAN.

Nous souhaitons également remercier l'université et le Département Informatique plus généralement pour la mise à disposition des outils et du matériel nécessaire à la réalisation de ce projet.

Sommaire

1 Introduction 5
2 Présentation du Projet 6
2.1 L'environnement de travail 6
2.1.1 Les participants de ce projet 6
2.1.2 Le matériel nécessaire à la réalisation de ce projet 6
2.1.3 Les outils, logiciels et langages utilisés 6
2.2 Les disciplines adoptées 7
2.3 Ingénierie logicielle 7
2.4 Les objectifs de ce projet 7
3 Approche méthodologique 10
3.1 Ciblage des fonctionnalités attendues 10
3.2 Utilisation de la méthode agile SCRUM Lite 11
3.3 Notre organisation 11
3.3.1 Notre ressenti sur cette manière de travailler 12
3.4 Planification des tâches et délais entre les livrables 13
4 Conception et programmation de notre jeu 15
4.1 Réalisation d'une boucle de jeu 15
4.1.1 Explications générales 15
4.1.2 Ajout d'un système de gestion du temps 15
4.1.3 Système de notification et thread 17
4.2 Création et affichage des cartes 19
4.2.1 Mise en place de l'environnement visuel 19
4.2.2 Chargement et affichage des cartes 22
4.3 Création des entités de notre jeu 25
4.3.1 Un besoin d'organisation et de responsabilisation 25
4.3.2 Des ennemis avec des comportements interchangeables 25
4.3.3 Un joueur aux multiples actions 25
4.4 Gestion d'événements 26
4.4.1 Principe général 26
4.4.2 La nécessité d'un moteur qui centralise ces évènements 26
4.4.3 Les collisions 27
4.4.4 Gestion des événements 28
4.4.5 Les solveurs 29
4.5 Gestion des entrées utilisateur 29
4.6 Elaboration d'une caméra 29

4.6.1 Problématique : 29
4.6.2 Fonctionnement 29
4.6.3 Qualité de l'affichage 30
4.7 Scriptage du jeu 31
4.7.1 L'intention du système de scriptage 31
4.7.2 Création d'un dossier dans le home de l'utilisateur 32
4.7.3 Transition entre les cartes 32
4.7.4 Règles de jeu 33
4.7.5 Configuration des touches 36
5 Bilan Technique 37
6 Conclusion 38
7 Summary of the project in English 39
8 Bibliographie / Webographie 40
9 Lexique 41
10 Annexes 43

1 Introduction

Ce projet relève d'une idée partagée entre 5 étudiants : créer un jeu-vidéo pour ordinateur, de sa conception jusqu'à sa réalisation. Nous avons imaginé notre jeu dans un genre d'action/aventure/réflexion, le tout dans un monde 2D dans lequel il serait possible de réaliser de nombreuses interactions. Par la suite le projet a évolué de manière à se transformer en véritable moteur de jeu, permettant à l'utilisateur de créer ses propres scénarios de jeu de A à Z.

Ce jeu a été nommé MAUJA Adventures en référence aux initiales de nos prénoms, et afin de souligner le côté ludique et attrayant de celui-ci.

Nous avons proposé ce sujet de projet car il nous plaisait car nous étions curieux de découvrir comment réaliser ce genre de jeu tout en nous posant la question : comment outiller efficacement nos connaissances afin de concevoir et programmer un jeu évolutif sans partir d'un moteur de jeu préexistant ?

L'objectif de ce projet était ainsi de fournir un jeu 2D interactif aux utilisateurs, jouable au clavier sur ordinateur, tout en le concevant extrêmement évolutif de manière à ce que l'utilisateur puisse lui aussi créer son propre jeu (ses environnements, comportements, décors, événements et interactions).

Cet accomplissement a été réalisé du 8 novembre 2021 au 21 mars 2022 dans le cadre du projet tuteuré à l'IUT de Clermont-Ferrand. Il nous a permis de découvrir de nouveaux modes de conception et de philosophie de programmation que nous allons vous détailler dans la suite de ce rapport.

Pour commencer, nous allons vous présenter notre projet dans les lignes suivantes, en commençant par vous décrire l'environnement dans lequel celui-ci s'est déroulé ainsi que les objectifs attendus. Après avoir détaillé notre méthodologie et l'organisation mise en place durant l'entièreté de ce travail, nous vous exposerons nos réalisations et nos interrogations dépeintes à travers différentes figures et diagrammes. Nous réaliserons un bilan technique afin de détailler tous les livrables concrétisés ainsi qu'une conclusion dans laquelle nous exprimerons notre point de vue sur le dénouement de ce projet et sur ce que nous avons appris.

2 Présentation du Projet

2.1 L'environnement de travail

2.1.1 Les participants de ce projet

Ce projet a été réalisé par 5 étudiants de deuxième année de l'IUT de Clermont-Ferrand : Antoine Viton, Jérémy Tremblay, Ugo Vignon, Maxime Wissocq et Adrien Coudour. Jérémy Tremblay a eu le rôle de chef de projet durant l'entièreté de ce travail. Notre professeur encadrant et ayant accepté notre proposition de sujet est M. Laurent PROVOT, il a également joué le rôle du client.

2.1.2 Le matériel nécessaire à la réalisation de ce projet

Pour mener à bien ce projet nous avons eu besoin de 5 ordinateurs avec des systèmes d'exploitation différents tel que Windows ou une distribution de Linux (Debian et Ubuntu par exemple). Nous nous sommes servis de nos ordinateurs personnels, mais nous avons également utilisé les ordinateurs mis à disposition par l'IUT. Nous avons également recouru à l'utilisation de tableaux dans les salles de travail afin de débattre sur la conception de notre projet.

2.1.3 Les outils, logiciels et langages utilisés

Ce projet a été développé en utilisant le langage Java (un langage orienté objet souple permettant de réaliser des jeux) et le Framework *JavaFX* qui est une bibliothèque d'interface utilisateur permettant la création d'interfaces graphiques.

Nous avons exploité l'environnement de développement intégré *d'IntelliJ IDEA* (développé par *JetBrains*) pour réaliser la programmation de l'entièreté de notre projet. Il s'agit d'un des IDE les plus connus, mais également un des plus flexible et ergonomique.

La conception de notre projet a pu être réalisée en continu avec le logiciel *StarUML*, aussi bien dans les diagrammes que nous réalisions avant de programmer, mais aussi dans ceux que vous trouverez tout au long de ce rapport.

Pour créer nos ressources graphiques pour notre jeu, nous avons utilisé un logiciel de sélection d'images que nous avons développé et qui est détaillé plus loin.

Vous verrez par la suite de ce rapport que nous avons dû créer des cartes pour notre jeu. Afin de le faire efficacement, nous avons utilisé *Tiled* qui est un logiciel d'édition de cartes 2D (permettant de réaliser aussi bien des cartes isométriques qu'orthogonales).

Pour assurer la lecture de ces cartes créées, nous nous sommes servis de la bibliothèque *TiledReader* pour charger le contenu des fichiers que l'on venait de créer.

Nous avons assuré le versionnage de notre projet via l'outil Git en déposant notre projet sur le *GitLab* de l'IUT.

Les différents documents et la maintenance du *backlog* de produits ont été réalisés avec LibreOffice (*Writer* et *Calc*).

Pour effectuer la rédaction de ce rapport, l'outil Word a été exploité, pour les différentes présentations et soutenances google Drive, google Doc et Slide ont été mobilisés.

2.2 Les disciplines adoptées

Ce projet n'aurait pu être réalisé sans certains concepts essentiels qui nous ont été enseignés ou que nous avons appris durant le déroulé de ce projet. Voici les disciplines principales dont nous nous sommes servis :

- **Modélisation / Conception** : Il s'agit là d'un élément essentiel dans ce projet, et c'est certainement la partie sur laquelle nous avons passé le plus de temps. Cette discipline nous a aidé à réfléchir sur la manière d'agencer nos classes, notre programme de manière à ce qu'il apporte les fonctionnalités attendues tout en restant souple, maintenable et cohérent. De la même manière, la modélisation a facilité la manière de cibler les besoins des utilisateurs, et a rendu notre code bien plus compréhensible pour quelqu'un n'ayant jamais participé au projet.
- **Programmation orientée objet** : Après la conception, nous avons passé beaucoup de temps à programmer notre petit moteur de jeu 2D. Nous nous sommes servis de l'objet qui est le paradigme de Java.
- **Design et conception graphique** : Utilisé pour la réalisation du visuel de notre jeu : la création et l'utilisation de nos ressources graphiques, la réalisation des interfaces graphiques (menus, fenêtre de jeu) depuis *JavaFX* et l'incorporation du système de calque pour ajouter de la profondeur au jeu.

2.3 Ingénierie logicielle

Tout au long de ce projet, nous avons utilisé UML pour nos phases de conception et Scrum en tant que méthode agile.

- **UML** : Il s'agit d'un langage de modélisation unifié à base de formes et pictogrammes permettant de fournir une méthode normalisée pour visualiser la conception de différentes vues statiques ou mêmes dynamiques d'un système.
- **Scrum** : Scrum est une méthode agile adapté pour répondre à des besoins changeants tout en livrant en continu des produits et fonctionnalités de qualité.

2.4 Les objectifs de ce projet

L'objectif principal de notre projet est de réaliser un jeu 2D permettant d'allier l'aspect d'aventure de combat et de réflexion. Ainsi, nous voulons offrir un produit capable de livrer aux joueurs une expérience complète et captivante en termes de fonctionnalités et de graphismes et d'énigmes.

L'idée est donc de réaliser un jeu 2D de type RPG pour ordinateur, dans lequel l'utilisateur pourra se déplacer librement à travers différentes cartes. Elles seront reliées entre elles, et l'utilisateur pourra ainsi explorer divers environnements. Ces environnements seront créés avec une grande variété d'images différentes, et les cartes se baseront ainsi sur un système de *tile-mapping**. Un système de calque sera présent afin de superposer les images les unes par-dessus les autres, afin de donner une impression de profondeur au jeu.

Les contrôles seront gérés par les touches du clavier. Une caméra sera présente et suivra les mouvements du joueur, tout en le gardant au centre de l'écran (sauf quand ce dernier se rapproche des bords et des coins).

Des ennemis avec différents comportements seront présents et auront pour objectif de faire perdre le joueur via des attaques physiques ou par des tirs de projectiles. Si les points de vie du joueur tombent à zéro, alors celui-ci aura perdu. Il pourra bien évidemment se défendre et les attaquer en retour à l'aide d'une épée. Muni d'un bouclier, il pourra parer les attaques à distance et les renvoyer contre eux.

Bien évidemment, le personnage sera soumis aux collisions du monde qui l'entoure. Il ne pourra pas traverser les obstacles, tout comme les autres ennemis et projectiles.

Notre jeu sera décomposé en plusieurs menus. Il sera possible de choisir de jouer ou de modifier des paramètres depuis le menu principal. En pleine partie, il sera possible à tout instant de mettre le jeu en pause afin de modifier les paramètres, de revenir au menu ou bien à la partie en cours.

Figure 2-1 Capture d'écran du jeu Alundra sur PS1. Il s'agit d'un jeu qui nous a inspiré dans notre idée de projet. On y retrouve le système de cartes, de tile-mapping, de combat, d'ennemis, d'énigmes et d'exploration.

L'objectif principal était donc de réaliser notre propre moteur de jeu, adapté spécialement pour des jeux 2D utilisant le système de *tile-mapping*. Cela signifie que nous allons devoir créer notre propre boucle de jeu, mettre en œuvre notre système d'évènements et de notifications, et générer un gestionnaire d'interactions chargé de valider ou d'invalider les scénarios du jeu.

Au fur et à mesure de l'avancée du projet, nous avons souhaité étendre cet objectif, et pour cela nous avons voulu offrir un jeu scriptable. Autrement dit, l'utilisateur doit être en mesure d'ajouter ses propres cartes, ses propres scripts d'évènements (ou règles du jeu),

ses propres transitions entre les cartes, et doit pouvoir choisir comment il veut configurer ses touches. Il doit pouvoir *modder* le jeu, le modifier en profondeur de la manière dont il le souhaite.

Ainsi, un utilisateur doit pouvoir, sans même devoir programmer, créer ses propres cartes, les relier, les incorporer au projet aisément (en les ajoutant simplement dans un dossier spécifique) et jouer dessus. Les collisions et déplacements doivent être fonctionnels, la caméra également. Il doit être en mesure d'ajouter ses propres règles du jeu (ses propres évènements) en éditant simplement des fichiers XML qui eux aussi seront automatiquement récupérés et lus. Il doit par exemple être libre de générer des leviers qui vont apparaître dans le jeu et qui vont réaliser des actions lorsqu'ils seront activés.

Enfin, nous aspirons à améliorer et tester nos compétences dans la conception et le développement, tout en découvrant et assimilant de nouvelles connaissances et compétences. Ce projet en équipe sera pour nous l'occasion d'avoir un aperçu sur le fonctionnement qu'il faut adopter en entreprise pour que le travail d'équipe soit efficace, que les tâches soient bien partagées et que le projet aboutisse.

3 Approche méthodologique

3.1 Ciblage des fonctionnalités attendues

Nous nous sommes réunis avant le début du projet pour débattre ce que nous souhaitions rendre et aussi pour lister les fonctionnalités que nous voulions implémenter.

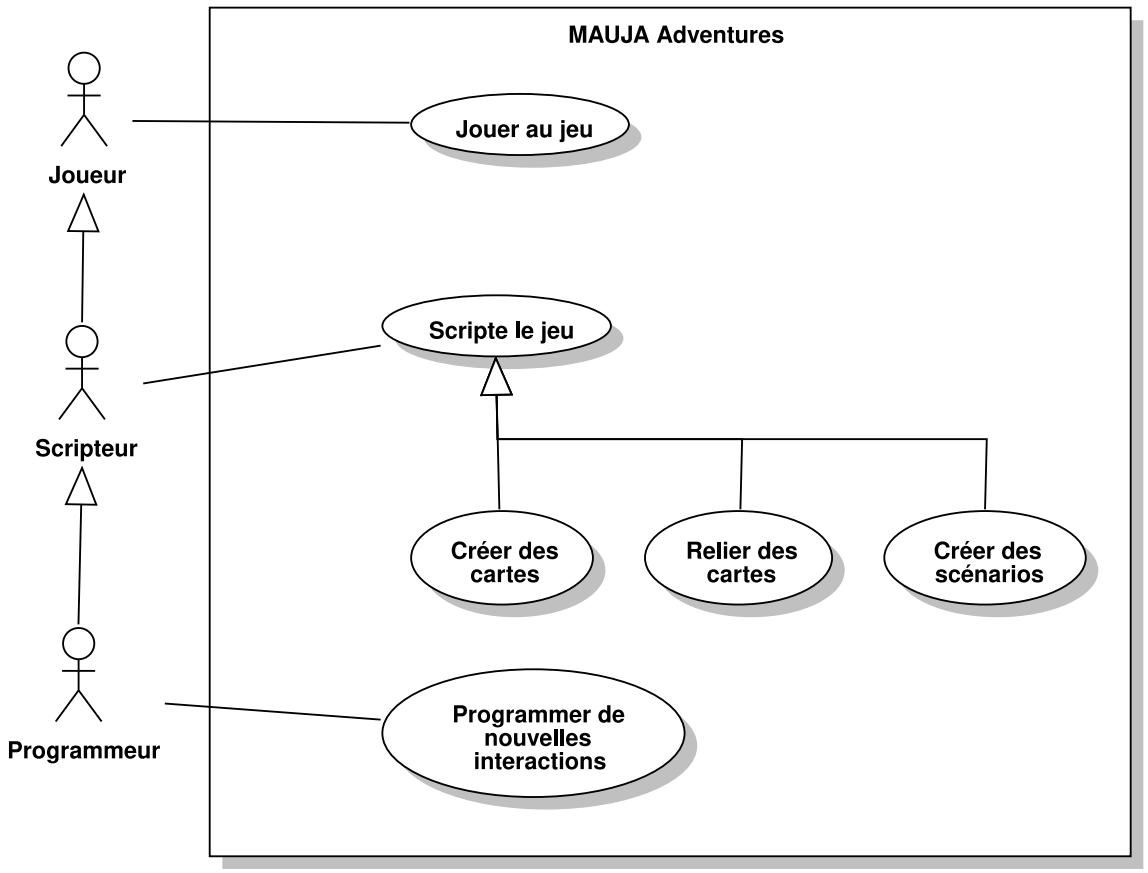


Figure 3-1 Diagramme de cas d'utilisation de notre jeu MAUJA Adventures.

Bien évidemment, nous voulions produire un jeu *jouable*, l'idée principale était de réaliser un moyen de divertissement permettant aux utilisateurs de se détendre et de s'amuser.

Cependant, le projet a par la suite évolué, et il nous est venu à l'esprit d'intégrer un système pour externaliser nos données permettant à l'utilisateur d'intégrer ses propres cartes au jeu. Il pourrait également les relier de la manière dont il le souhaite, de manière à pouvoir complètement personnaliser son jeu. Nous avons eu envie d'intégrer un système de *scriptage*, l'objectif étant que l'utilisateur puisse définir ses propres règles du jeu, c'est-à-dire, qu'il puisse préciser la manière dont il veut que le jeu se déroule si telle situation se produit. Cela lui laisse alors la possibilité de réaliser des cinématiques, des actions spécifiques lorsque certaines conditions sont vérifiées, et bien plus.

Enfin, nous ciblons aussi des développeurs même si cela peut paraître surprenant. En effet, l'idée un système de *scriptage* en tant que tel est intéressant, mais il faudra bien programmer ces interactions et règles à un moment donné. Nous ne pourrons pas en implémenter des dizaines, mais nous voulons laisser la possibilité à d'autres de le faire. Ainsi la difficulté est de construire un jeu évolutif : il doit être simple de pouvoir ajouter

des comportements ou des règles dans notre jeu sans devoir modifier notre implémentation en interne.

Cela laisse ainsi la possibilité aux passionnés (les programmeurs) de développer de nombreuses interactions et fonctionnalités. Les scripteurs pourront alors utiliser le travail des développeurs afin de mettre en place leurs règles du jeu. Enfin, les joueurs pourront alors profiter des créations des autres et s'amuser.

Pour ce qui est de la planification du projet, nous avons mis en place une gestion en décomposant le projet en taches, pour cela nous avons réalisé un WPF et des diagrammes de Gantt personnels et collectifs. A la mi-projet (début janvier 2022), nous avons effectué une analyse entre les prévisions et ce qui a été réalisé dans le but de comprendre pourquoi des écarts entre les prévisions et les réalisés sont apparus.

3.2 Utilisation de la méthode agile SCRUM Lite

3.3 Notre organisation

Avant que le projet ne débute, nous avons tous lus le livre *SCRUM et XP depuis les tranchées*. Notre enseignant M. PROVOT a souhaité que nous utilisions une version plus lite de la méthode agile *Scrum* que nous appellerons par la suite *Scrum lite*.

Ce livre explique la manière dont fonctionne la méthode agile *Scrum*. La personne qui écrit ce livre ne se contente pas de l'expliquer mais donne également son ressenti sur la manière dont il l'a vécu.

Son fonctionnement est plutôt simple. Nous avons tout d'abord réuni toutes nos fonctionnalités que nous avons séparé en tâches. Ces fonctionnalités ont ensuite été incorporées dans un tableau nommé le *backlog de produit*, un outil de la méthode agile Scrum.

Dans ce tableau, on retrouve le nom de la fonctionnalité, un indice représentant son importance et une estimation du nombre d'heures pour réaliser la fonctionnalité. Nous rédigeons dans ce tableau ce qui est attendu du point de vue du client mais également du point de vue d'un développeur pour chaque tâche. En réalité, nous étions à la fois les développeurs de ce projet et à la fois les clients car nous avions des attentes. M. PROVOT jouait le rôle du client de manière à nous pousser en tant que développeurs à produire un projet de qualité.

Durant ce projet, nous avons fonctionné en découpant notre projet en « sprints », des périodes de travail d'une durée de 2 semaines. Nous planifions une démo à chaque sprint, ce qui correspond à un objectif à tenir et à montrer au client. Pour cette démo, nous définissions les tâches que nous souhaitons réaliser afin de présenter certaines fonctionnalités.

Nous choisissions les tâches depuis le *backlog* de produit en fonction de leur niveau de priorité (on prenait toujours les tâches avec le plus haut niveau de priorité).

Les tâches étaient ensuite sélectionnées en fonction d'un nombre de points d'histoire. Nous avions un nombre limité de ces points (qui représentait le nombre d'heures de travail par semaine, soit 10 points d'histoire par personne et par semaine). Nous utilisions ainsi nos 100 points d'histoire (pour deux semaines de sprint) sur les tâches les plus importantes. Ces points d'histoire fluctuaient en fonction de notre niveau d'implication et du travail à côté que nous avions à réaliser.

Nous nous répartissions ensuite le travail à réaliser entre les différents membres du groupe. Pour ce faire, nous utilisions les issues de *GitLab* pour lister les tâches et fonctionnalités à réaliser, et nous les assignons à des personnes.

Durant le sprint, nous nous réunissions régulièrement pour discuter de notre avancée, de nos problèmes et éventuellement pour s'entraider lors de réunions appelées *mêlées* dans la méthode Scrum. A la fin du sprint nous présentions les fonctionnalités réalisées au client (M. PROVOT) sous forme d'une démo puis nous démarrions un nouveau sprint.

Vous pourrez voir en annexes des captures d'écrans de notre organisation sur *GitLab* ainsi que sur des parties du *backlog* de produits.

3.3.1 Notre ressenti sur cette manière de travailler

Cette manière de travailler nous a été très bénéfique. En effet elle permet d'avoir du recul sur les fonctionnalités désirées, en termes d'ordre de priorité, de durées, … Cela permet de se fixer des objectifs de manière régulière et de s'adapter aux problèmes et ralentissements que l'on rencontre.

Au début nous avons rencontré des difficultés à maintenir le rythme soutenu de la méthode Scrum. Il y avait beaucoup de choses auxquelles penser entre mettre à jour notre Wiki régulièrement, modifier le backlog de produits, réaliser les mêlées, réaliser de la conception, programmer, prendre en main les différents outils (dont JavaFX).

Les mêlées ont également été source de retards. Nous n'avons pas assez communiqué, en nous fixant un compromis de réaliser des mêlées non pas une fois par jour mais une fois tous les deux jours. Cependant, vers la fin du projet, nous réalisions en moyenne une mêlée tous les 3-4 jours. Cela avait pour conséquence d'allonger les mêlées lorsque l'on en faisait (car forcément nous avions plus de choses à dire), et ainsi cela nous donnait moins envie d'en faire (cercle vicieux). Nous aurions dû maintenir un rythme fréquent mais léger, plutôt que d'en faire plus rarement et des très longues.

Nous n'avons pas non plus réussi à bien estimer les durées des tâches (dû à un manque d'expérience). Elles duraient souvent bien plus de temps que ce qui était prévu, ce qui nous a retardé dans nos plannings. Nous avons aussi sous-estimé le temps que prenait la conception, car il s'agissait d'une des étapes les plus dure du projet, et nous y passions parfois un grand nombre d'heures avant d'arriver à avoir quelque chose que nous pouvions programmer.

Malgré ces déconvenues, nous avons tout de même réussis à réaliser un projet dont nous sommes fiers. Nous n'avons pas intégré toutes les fonctionnalités espérées, mais nous avons élargi le périmètre de fonctionnalités initiales (le système de *scriptage* ainsi que la grande évolutivité).

Nous avons tout de même approuvé cette méthode agile par la versatilité qu'elle propose. Le *backlog* de produit nous a tous convaincus par son efficacité, et nous a permis de bien moins avancer à l'aveugle. Nous savions quelles tâches effectuer et dans quel ordre les réaliser. Il s'agit vraiment d'un point que nous avons beaucoup aimé dans ce projet.

Nous avons aussi beaucoup aimé le concept des démos. Elles permettaient ainsi de voir où nous en étions, d'éventuellement déceler des bugs, et nous forçaient à avancer.

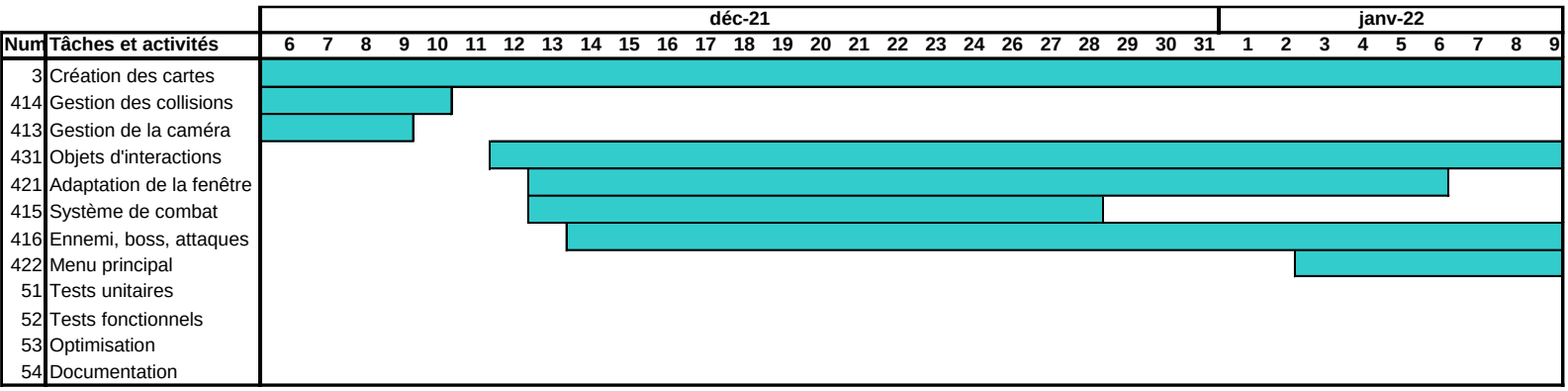
Nous avions des objectifs réguliers et soutenus, et ces objectifs nous motivaient à avancer de manière à arriver fier le jour de la démo, avec un projet fonctionnel.

La méthode agile Scrum reste une référence dans le monde du travail. Nous sommes contents d'avoir pu l'utiliser, car il s'agit de compétences en plus qui nous seront utile par la suite. De plus, nous ne reproduirons pas les erreurs que nous avons faites dans ce projet.

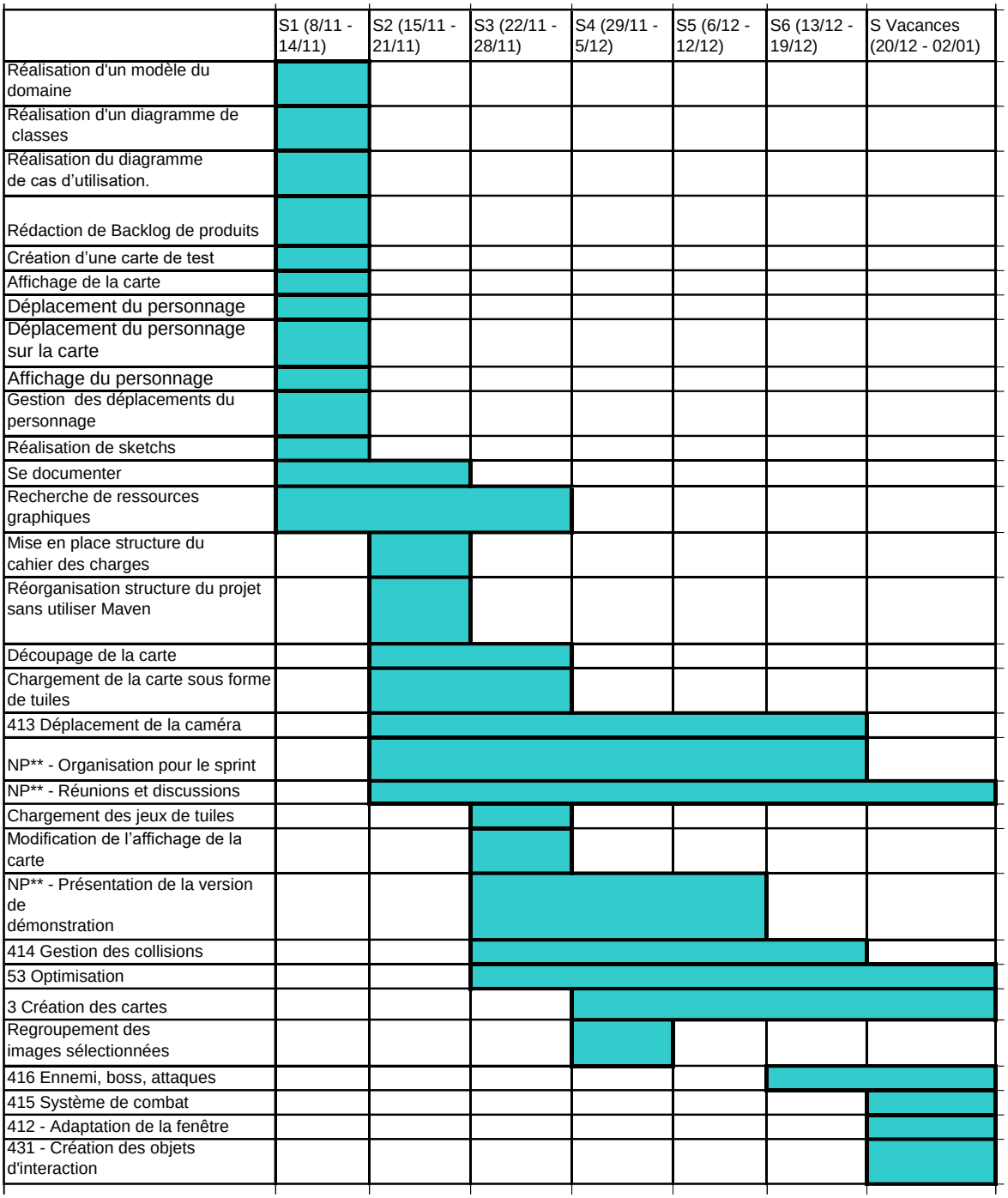
3.4 Planification des tâches et délais entre les livrables

Pour ce qui est de la planification du projet, nous avons mis un place une gestion en décomposant le projet en taches, pour cela nous avons réalisé un WPF et des diagrammes de Gantt personnels et collectifs. A la mi-projet (début janvier 2022), nous avons effectué une analyse entre les prévisions et ce qui a été réalisé dans le but de comprendre pourquoi des écarts entre les prévisions et les réalisés sont apparus.

Prévisionnel commun P2



Réalisé commun P2



La différence entre le prévisionnel et le commun s'explique de différentes façons. La réalisation des taches relatives à la création des cartes a pris du retard à cause de problèmes lors de l'utilisation de la librairie TiledReader, nous avons dû chercher comment cette dernière fonctionne.

Les mêmes taches que cités précédemment ont pris du retard car un logiciel a dû être créé pour réaliser des tilesets personnalisés.

Le temps dépensé pour ce logiciel a également décalé certaines autres taches comme la création des IA et attaques ou encore la création des projectiles.

Enfin, la prise en main de la méthode Scrum nous a retardé comme expliqué plus haut. Enfin, nous avons voulu étendre les fonctionnalités de notre jeu pour faire un jeu scriptable, et cela nous a pris beaucoup plus de temps que prévu, ce qui est la cause d'un grand retard sur le projet.

4 Conception et programmation de notre jeu

Nous allons ici vous décrire quelles ont été les grandes étapes qui ont été réalisées pour mener à bien nos objectifs.

4.1 Réalisation d'une boucle de jeu

4.1.1 Explications générales

Nous allons tout d'abord vous détailler comment nous avons implémenter le point central de notre jeu, qui est la boucle de ce jeu. Il ne s'agit pas d'une simple boucle, il s'agit d'un patron de conception essentiel à la programmation de n'importe quel jeu.

Ce patron nommé la *Game Loop* consiste à découpler la progression du temps de jeux des entrées utilisateurs et de la vitesse du processeur.

Il nous fallait absolument l'implémenter dans notre jeu afin de pouvoir complètement maîtriser le cycle de vie de notre jeu, gérer l'ordre des actions que nous allions réaliser, ainsi que structurer nos classes et notre projet.

Même s'il semble compliqué, il s'agit d'un patron de conception assez simple à mettre en place. Il consiste à réaliser une boucle *à priori* infinie dans laquelle font se succéder 3 grandes étapes :

1. **Gestion des entrées utilisateur**
2. **Mise à jour du jeu**
3. **Affichage**

Il faut noter ici que l'ordre à une importance.

Contrairement à la plupart des logiciels, les jeux continuent de fonctionner, de se mettre à jour, de réaliser des calculs lorsque l'utilisateur ne réalise pas d'action (autrement dit, lorsqu'il n'appuie sur aucune touche).

C'est d'ailleurs cela qui amène un charme à la plupart des jeux : ceux qui sont basés sur un rythme ou un délai ne fonctionne que parce que le jeu continu de se dérouler en arrière-plan, même si l'utilisateur ne l'a pas sollicité.

C'est le premier élément clé d'une véritable boucle de jeu : il traite les éléments clés mais ne les attend pas. Nous allons ici nous attarder sur l'étape de mise à jour du jeu, les étapes d'affichage et de gestion des entrées sont expliquées plus bas dans ce rapport.

4.1.2 Ajout d'un système de gestion du temps

Par la suite, il nous a fallu intégrer la gestion d'un véritable temps dans notre jeu, car sinon le jeu se met à s'exécuter extrêmement rapidement, consommant tout le processeur et les ressources disponibles de la machine. Autrement dit : le jeu va plus vite sur les machines plus performantes, ce qui n'est pas ce que nous voulons.

Nous devons ainsi imposer un temps régulier où le jeu sera mis à jour. Nous avons pris comme objectif d'avoir un jeu mis à jour 60 fois par seconde (ce qui est le plus commun dans les jeux, on parle également de FPS*). Si on se base sur une unité de nanosecondes (1 seconde = 10^9 nanosecondes), alors le jeu devra être rafraîchit une fois toutes les 60 / 10^9 secondes, soit une fois toutes les 16 666 666 nanosecondes environ.

Pour cela, nous nous sommes basés sur le temps d'exécution entre chaque tour de boucle. Si on considère un tour de boucle t, le temps total pris avant l'itération de ce tour (i.e. comprendre la mise à jour du jeu) est x nanoseconde, alors au début du tour t_{+1}, on peut réaliser la différence entre ce temps x et le temps x_{+1} mesuré. Si cette différence est supérieure ou égale au taux de rafraîchissement (nous avons calculé et établit précédemment que c'était 16 666 666 nanosecondes), alors le jeu doit être mis à jour. Dans les autres cas, on ne fait rien, hormis attendre que le délai entre le taux de rafraîchissement et x_{+1} soit écoulé.

Voici un exemple plus parlant :

```
// On récupère le temps total depuis l'éxécution du programme.
dernierTemps = System.nanoTime();
// On entre dans la boucle "infinie".
while (actif) {
    // On récupère le temps associé à cette itération dans la boucle.
    tempsCourant = System.nanoTime();
    tempsEcoule = tempsCourant - dernierTemps;
    // Si la différence entre le temps de cette itération et le temps de celle où le
    // jeu a été mis à jour est supérieur ou égale à la fréquence de mise à jour
    if (tempsEcoule >= TEMPS_AVANT_NOTIFICATION) {
        ticker(tempsCourant); // Mise à jour du jeu.
        dernierTemps = tempsCourant;
    }
    else {
        // Sinon on calcule la différence avant de mettre le jeu à jour.
        tempsAttente = TEMPS_AVANT_NOTIFICATION - tempsEcoule;
        try {
            // On s'endort jusqu'à la prochaine itération.
            sleep(tempsAttente / TEMPS_MILLISECONDE, (int) (tempsAttente %
TEMPS_MILLISECONDE));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Le temps de la dernière itération est récupéré par System.*nanoTime*. On entre ainsi dans la boucle infinie, et à chaque tour, on récupère le temps courant. Si la différence entre le temps courant et le temps de la dernière itération est supérieur ou égal à la fréquence de rafraîchissement (16 666 666 nanosecondes), alors la méthode *ticker* est appelée et il faut imaginer qu'elle va venir mettre à jour l'ensemble du jeu. On réaffecte ensuite le dernier temps.

Dans l'autre cas, on calcule le temps qu'il reste avant de mettre à jour le jeu et plutôt que de réaliser des itérations inutiles, on s'endort du temps nécessaire. De cette manière, au prochain tour, on devrait de nouveau entrer dans la condition et mettre à jour le jeu.

Ce patron nous a ainsi permis d'avoir un système qui met à jour régulièrement notre jeu, tout en tenant compte du temps et du délai que pouvait prendre la mise à jour (qui est aléatoire et non prévisible). Avec un tel système, nous avions un décompte des secondes très précis, et le jeu était mis à jour 60 fois par seconde.

```
Nouvelle seconde : 1
Nouvelle seconde : 2
Nouvelle seconde : 3
Nouvelle seconde : 4
```

Cependant, nous devions maintenant prévenir notre jeu qu'il devait se mettre à jour.

4.1.3 Système de notification et thread

Nous n'allions pas réaliser toute la mise à jour du jeu dans la méthode *ticker*, c'est pour cela que nous avons conçu notre classe jeu comme ayant une boucle de jeu (la classe chargée de *ticker* à intervalles réguliers).

Il nous fallait ainsi un moyen d'avertir notre jeu à intervalle régulier que celui-ci devait se mettre à jour. Pour atteindre cet objectif, nous avons utilisé le patron Observateur.

Il permet de mettre un jour un objet quand un autre objet change d'état, et c'est ainsi ce système d'écoute qui permet à notre jeu d'être réactif et de se mettre à jour.

Voici la manière dont nous l'avons implémenté :

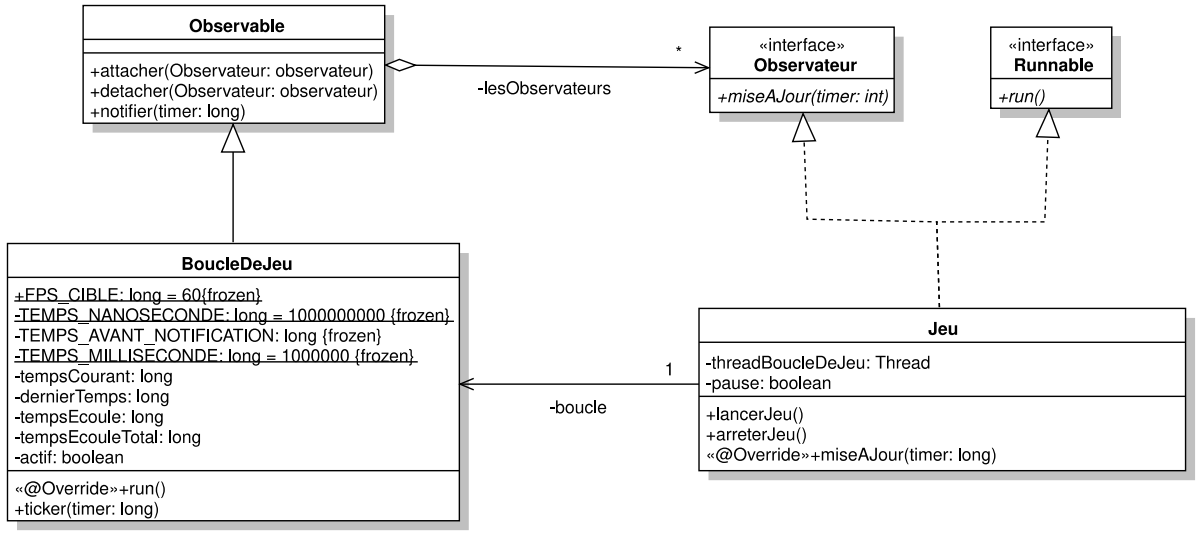


Figure 4-1 Image vectorisée représentant un diagramme de classe mettant en avant le patron Observateur sur notre boucle de jeu.

Ici notre boucle de jeu, une fois lancée, ne s'interrompt jamais (on a tout de même une méthode pour l'arrêter). Notre jeu possède une boucle de jeu qu'il instancie et qu'il lance. Il possède une méthode de mise à jour qui est appelée par la boucle de jeu à intervalle régulier comme expliqué précédemment.

La méthode de la classe Jeu est appelé, pourtant la boucle de jeu ne connaît pas jeu. En réalité, le jeu est un observateur, autrement dit, c'est celui qui va être notifié d'un changement, c'est le témoin. Il implémente donc la méthode de mise à jour dans laquelle il reçoit un temps, et avec laquelle il pourra effectuer la mise à jour du jeu, utiliser le temps ou non qui lui est passé en paramètre, etc.

A l'inverse, la boucle de jeu est un observable. On peut ainsi s'abonner ou se désabonner au jeu, et notifier tous les observateurs qui sont maintenus dans sa collection.

Ainsi, dans la méthode *ticker* de la boucle de jeu, on vient notifier tous les observateurs qui sont abonnés à celle-ci. Cela signifie que lorsqu'on instancie une boucle de jeu dans la classe jeu, on vient s'abonner à celle-ci de manière à être notifié régulièrement, et pouvoir mettre à jour le jeu. Voici la méthode *ticker* en question :

```
public void ticker(long timer) {
    Platform.runLater(() -> notifier(timer));
}
```

Note : l'appel à la méthode de classe *runLater* de la classe *Platform* signifie que l'on autorise le thread graphique à réaliser l'action de notification au thread courant, afin de faire la mise à jour de l'affichage.

Cependant un problème subsistait : le fait de s'endormir dans le programme de la boucle de jeu a pour effet d'endormir tout le thread principal de notre programme. C'est

comme si le jeu se figeait ou plantait. Pour éviter cela, nous avons créé un thread (ou fil d'exécution en français) que nous maintenons dans la classe jeu. Le jeu est alors notifié à intervalles réguliers qu'il doit se mettre à jour, depuis un autre thread (le thread de la boucle de jeu).

4.2 Création et affichage des cartes

4.2.1 Mise en place de l'environnement visuel

4.2.1.1 Le tile-mapping

L'objectif était de faire un jeu et donc d'avoir un rendu graphique. Nous devions donc créer et afficher nos cartes. Pour cela nous avons utilisé le système de tile-mapping.

Le *tile-mapping* est une technique de création de cartes, qui consiste à utiliser des *tilesets* (qui sont des images composées de plusieurs tuiles), de manière à pouvoir créer des cartes en plaçant librement les tuiles de ces *tilesets*. On peut ainsi voir une carte comme étant une grosse grille dans la laquelle on vient afficher des petits carrés, qui, bien assemblés, donnent l'illusion d'une carte.

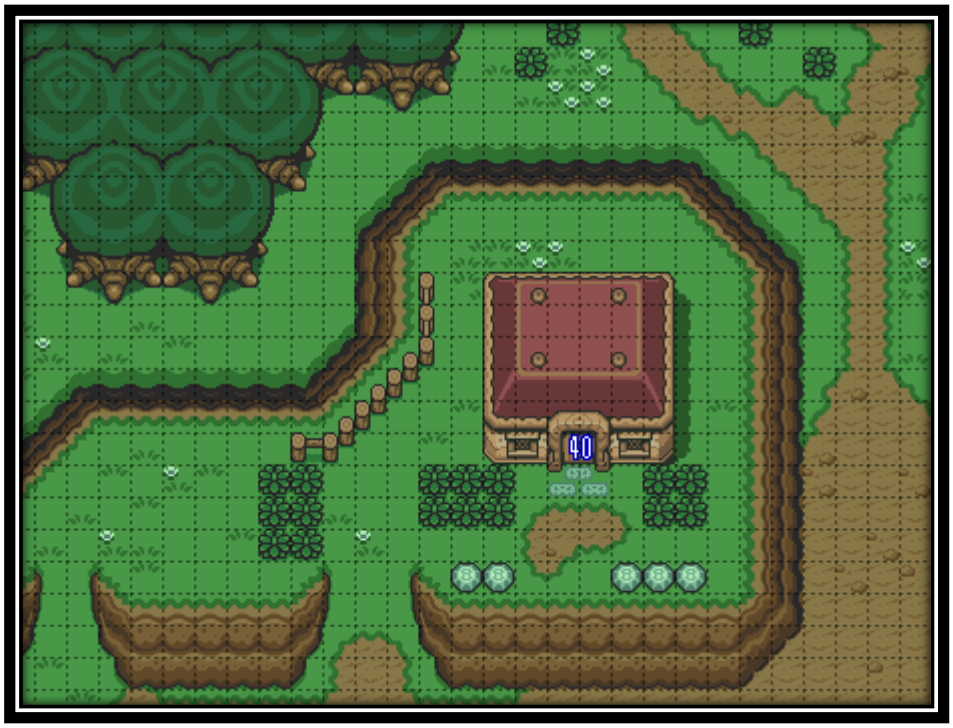


Figure 4-2 Capture d'écran d'une carte du jeu The Legend of Zelda A Link to The Past

A la manière des jeux Zelda ou Mario, de très nombreux jeux se basent sur ce système de tile-mapping.

Nous avons décidé d'implémenter ce système de *tile-mapping* pour créer notre jeu mais à plus haute échelle : nous voulions pouvoir superposer plusieurs calques les uns sur les autres, de manière à pouvoir superposer des tuiles, amener de la profondeur à notre jeu, pouvoir créer une sensation de 3D. En bref, nous voulions que les graphismes soient plus poussés dans les détails.

4.2.1.2 Recherche de ressources

Pour obtenir un tel rendu, il nous fallait créer nos cartes, mais avant cela, nous avions besoin de ressources graphiques. Les *tilesets* sont des collections de tuiles et donc d'images, nous devions donc trouver plusieurs de ces *tilesets* afin de pouvoir les utiliser dans notre jeu.

Figure 4-3 Image d'un tileset.

Ce tileset est composé de 1024 tuiles de 32x32 pixels. Il nous a permis de créer une carte extérieure avec sa diversité de couleurs et d'éléments de décoration.

Ces ressources graphiques ont toutes été cherchées sur des sites libres de droits, comme OpenGameArt ou Itch.io, puis ont été répertoriés sur une page Wiki de notre *GitLab*. Cependant, certains *tilesets* ne correspondaient pas à nos attentes et contenaient des tuiles que nous ne voulions pas ou certaines tuiles étaient manquantes.

4.2.1.3 Création de nos propres ressources et de notre logiciel

Ces différents problèmes on fait naître des besoins que nous avons tout d'abord essayés de régler en y faisant à la main car nous ne trouvions pas d'outils pour le palier ce problème. Depuis *Paint*, nous réalisions des découpages, mais nous avons très vite abandonné car cela prenait beaucoup trop de temps et les découpages manuels étaient source d'erreurs. L'idée nous ait ensuite venu d'automatiser ce traitement via la création d'un logiciel. L'avantage était que nous pouvions répondre parfaitement à nos besoins comme nous étions à la fois les clients et les développeurs.

Nous avons donc conçu ce logiciel de création de *tilesets* en *JavaFX*. Cet outil est simple d'utilisation mais également très puissant. Ce logiciel permet de sélectionner un nombre quelconque de *tilesets* par l'utilisateur, de manière à lui laisser un maximum de possibilités.

L'application se décompose de deux zones distinctes. La partie de gauche est la zone de dessin, où l'utilisateur va pouvoir créer son *tileset*, et la partie de droite contient les différents *tilesets* sélectionnés, dans différents onglets.

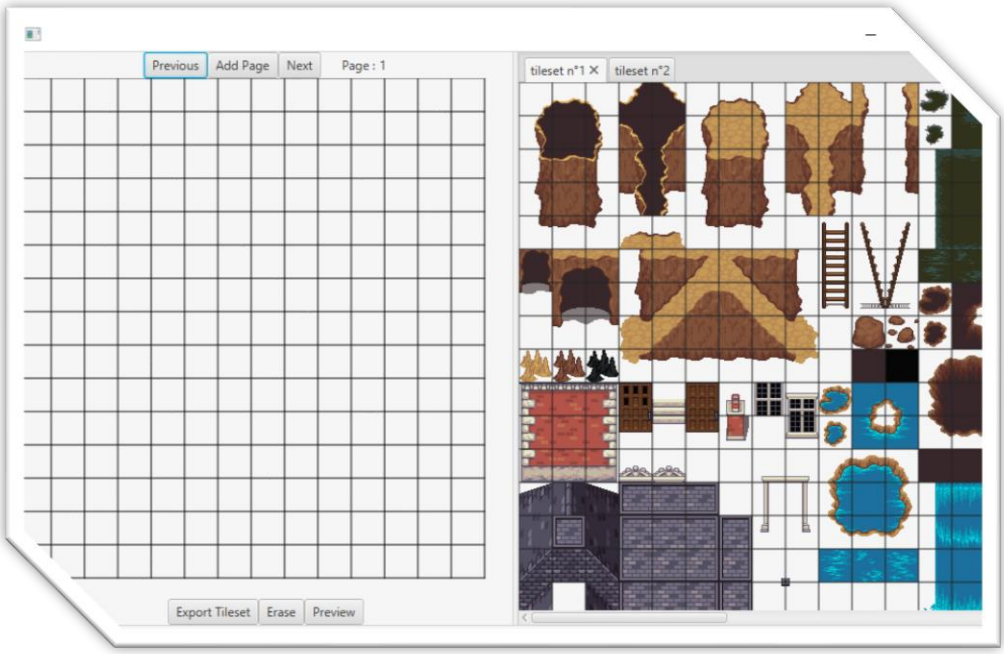


Figure 4-4 Capture d'écran de la page principale du logiciel de création de tilesets.

Le *tileset* est visuellement découpé en carrés et chaque carré correspond à une tuile graphique du *tileset*. La partie de gauche possède quant à elle plusieurs outils pour dessiner, effacer, sélectionner. Un système de navigation a été réalisé permettant de naviguer entre les différentes pages mais également d'en ajouter.

Plus d'informations et de captures d'écran de ce logiciel sont disponibles dans les annexes.

Finalement, cette application nous a permis d'appréhender le fonctionnement de *JavaFX*, mais également d'avoir à disposition un outil multi usage qui nous permettra de gérer facilement nos ressources graphiques et de complètement personnaliser notre système graphique durant l'entièreté du projet.

4.2.1.4 Création de nos cartes avec le logiciel Tiled

Après avoir créé ce logiciel, nous pouvions maintenant créer nos cartes. Pour ce faire, nous avons utilisé le logiciel libre *Tiled*. C'est un logiciel dédié à la création de carte de jeu vidéo qui utilise le système de *tile-mapping*.

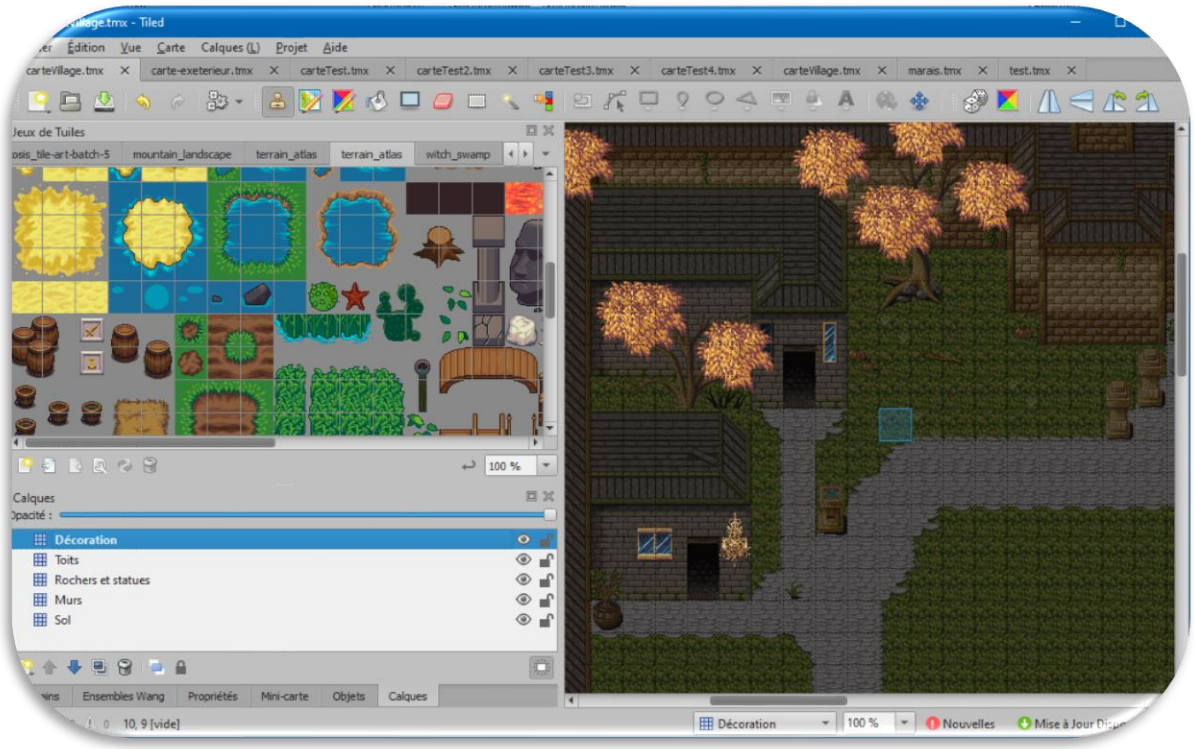


Figure 4-5 Capture d'écran du logiciel d'édition de cartes Tiled. Son interface utilisateur est assez intuitive. On retrouve la notion d'édition de carte (à droite), et la notion de tilesets (à gauche).

Il fut aisé de réaliser des cartes avec ce logiciel. Son utilisation est simple : on retrouve une grande zone de dessin délimitée sous forme de carreaux (ou de tuiles), dans laquelle il est possible de sélectionner des tuiles des tilesets que l'on souhaite, et de les placer dessus librement.

Son fonctionnement est détaillé dans les annexes si vous souhaitez en savoir plus.

4.2.2 Chargement et affichage des cartes

4.2.2.1 De la nécessité d'un système de parsage à la librairie TiledReader

Une fois les cartes créées, il fallait les afficher dans notre jeu. Nous ne pouvions pas simplement afficher l'image de la carte, car chaque tuile va par la suite posséder des collisions qui lui sont propre. Ainsi, afficher simplement une image ne nous permettrait pas d'effectuer les calculs pour le système de collision (nous ne pourrions pas détecter les tuiles sur l'image). Il nous faut ainsi afficher chaque tuile séparément dans notre jeu.

Mais avant cela, nous devions donc exporter nos cartes depuis le logiciel *Tiled* et les lire depuis notre programme Java afin de pouvoir les afficher. *Tiled* propose un système d'export en différents formats, on peut notamment citer le lua, le json, le xml et le tmx. C'est ce dernier format d'export qui nous a intéressé.

Il s'agit en réalité de XML mais avec des balises et attributs propres à l'éditeur *Tiled*, et il exporte les tuiles comme étant des identifiants uniques pour chaque carte.

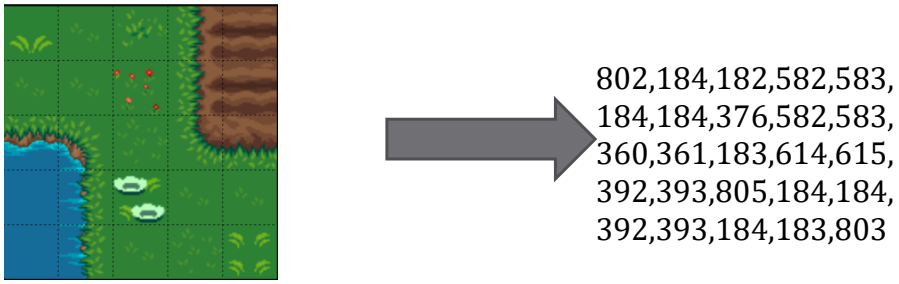


Figure 4-6 Une carte de 5x5 tuiles dessinée sous l'éditeur Tiled (à gauche), et sa correspondance une fois exportée (à droite), sans les balises XML.

Si vous voulez en savoir plus sur la manière dont *Tiled* transforme une carte dessinée en tmx, vous pouvez consulter les annexes.

Nous ne savions pas comment lire ces fichiers tmx, et puis nous avons découverts sur le site officiel de Tiled qu'il existe deux librairies en Java permettant de parser ce XML et de récupérer les informations des cartes. Une seule permettait de réaliser vraiment ce que nous voulions, c'est-à-dire récupérer les ID des tuiles (pour ensuite leur appliquer des transformations, faire des calculs, etc.). Cette librairie se nomme TiledReader.

Cette bibliothèque n'avait pas beaucoup de documentation, n'était plus à jour, et il nous a fallu à plusieurs reprises lire le code qui se trouvait dans les classes pour bien comprendre comment elle fonctionnait. De plus, elle n'intégrait pas toutes les fonctionnalités que l'on espérait. Par exemple, il n'était pas possible de récupérer les images des tuiles ou des *tilesets*, nous avons dû gérer cette partie manuellement.

Elle ne permettait pas non plus de récupérer le *firstgid* de chaque *tileset* alors que pourtant, elle le parsait bien et le stockait bien en mémoire. Le *firstgid* correspond au premier ID de tuile de chaque *tileset*, c'est utilisé lorsqu'on utilise plusieurs jeux de tuiles sur une même carte pour différencier les tuiles entre les jeux de tuiles).

Nous nous sommes ainsi servis de cette bibliothèque pour élaborer une classe qui récupère toutes les données d'une carte créée depuis *Tiled* et qui les stocke dans notre modèle.

Pour organiser nos données, nous avons réalisé une conception particulière au sein de nos classes. Voici un diagramme de classes pour mieux comprendre :

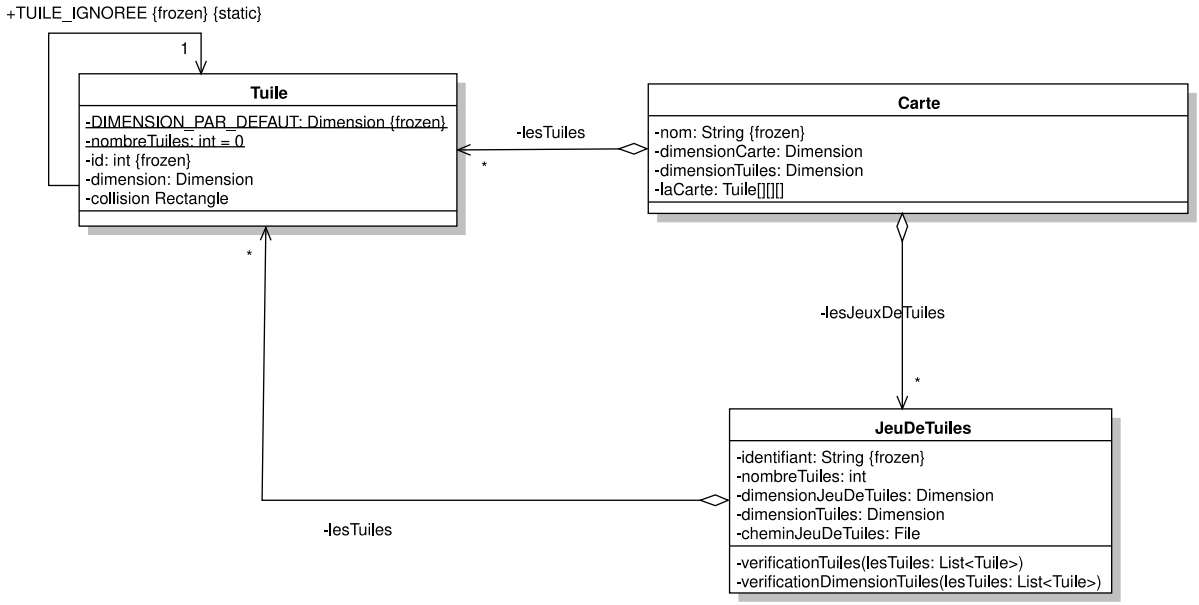


Figure 4-7 Image vectorisée représentant un diagramme de classes sur les données de nos cartes.

La classe *Tuile* représente ainsi une case dans notre monde. Elle possède ses propres dimensions, un rectangle représentant une zone de collision (nous y reviendrons après) et surtout un ID permettant de l'identifier de manière unique.

La classe *JeuDeTuiles* contient un grand nombre de données utiles. Elle est unifiée par son identifiant, elle conserve une liste de références vers chacune des tuiles du jeu de tuiles, ainsi que les dimensions de ces tuiles (elle effectue des vérifications au préalable pour être sûr que toutes les dimensions sont compatibles). Il est possible de connaître les dimensions de ce jeu de tuiles (en nombre de tuiles en largeur et en hauteur), ainsi qu'un fichier qui amène à l'image de ce jeu de tuiles.

Enfin une *Carte* n'est rien d'autre qu'une collection de jeux de tuiles et de tuiles, et est identifiée par une chaîne de caractère (le nom de la carte, qui est forcément unique). On connaît également ses dimensions. Nous avons choisi de maintenir les données relatives à nos cartes dans un tableau de Tuiles à **3 dimensions**, en voici les raisons :

- Les deux dernières dimensions permettent de dessiner notre grille de tuiles sous forme d'un plan à deux dimensions. La dimension en plus sert à permettre l'utilisation de calques, la superposition d'objets et les effets de 3D.
- Une fois instanciée, notre carte ne va pas s'étirer ou se rapetissir, d'où l'intérêt d'utiliser un tableau, il n'y aura pas de réallocation de mémoire car pas de changement de dimensions.
- Nous n'allons pas modifier les tuiles qui se trouvent sur la carte en pleine partie, donc il semble intéressant d'opter pour un tableau, et l'intérêt principal est que nous allons accéder directement aux éléments du tableau (et souvent). Cette structure de données est idéale dans ce cas de figure car la complexité pour récupérer un élément dans un tableau est constant

Note : Nous avons stocké les dimensions des cartes, tuiles et jeux de tuiles en mémoire car nous avions pour objectif de réaliser un programme générique. Nous ne voulions pas devoir changer notre programme si nous souhaitons ajouter de nouvelles cartes, tuiles ou *tilesets* de dimensions différentes dans le futur.

4.2.2.2 Un découpeur d'images pour notre toolkit graphique

Une fois les données stockées en mémoire, il nous fallait afficher notre carte. Mais avant cela, il restait une dernière étape : charger les images des *tilesets*. Ces images contiennent toutes nos tuiles, et ne sont donc pas utilisables en l'état. Pour s'en servir, nous devions les découper en fonction des dimensions des tuiles. Ainsi un *tileset* de 512x512 pixels contenant des tuiles de 32x32 pixels sera découpé en carrés multiples de 32, de manière à donner lieu à 16 x 16 tuiles, soit 256 tuiles au total.

Nous avons réalisé une classe ayant la responsabilité de découper les images de nos *tilesets* afin de pouvoir stocker ces images avec leurs tuiles correspondantes.

Enfin nous avons réalisé l'affichage de notre carte en affichant chaque tuile individuellement dans un canvas. On précise alors la position d'où on veut afficher la tuile, et elle est alors dessinée.

4.3 Création des entités de notre jeu

4.3.1 Un besoin d'organisation et de responsabilisation

La structuration des entités a été faite de telle sorte à centraliser les comportements pour créer des points d'extensibilité. Notre classe la plus abstraite est *Entite*. Elle représente une entité ayant une vitesse de déplacement ainsi qu'une direction. En descendant d'un niveau d'abstraction, la classe « Vivant », représente quant à elle une entité possédant des points de vie ainsi qu'une attaque. Pour ce qui est des classes de plus bas niveau nous y trouvons les classes *PersonnageJouable*, *Ennemi* et « Destructible ». Un *PersonnageJouable* n'est autre qu'un personnage qui est jouable par l'utilisateur. Cette classe ne possède qu'un état d'action qui reflète l'action qu'est en train de faire le personnage. La classe « Ennemi » est propre aux ennemis de notre jeu. Un ennemi possède un comportement mais cette partie vous sera détaillée juste après. Pour ce qui est de la classe « Destructible », son rôle est de représenter un élément (par exemple un projectile) qui sera détruit.

4.3.2 Des ennemis avec des comportements interchangeables

Les ennemis de notre jeu peuvent posséder plusieurs comportements. Comme premier comportement nous avons *ComportementTireur*, qui est le comportement d'un ennemi tirant des projectiles. *ComportementPoursuite* est quant à lui celui d'un ennemi qui va continuellement chercher à se rapprocher du personnage pour lui faire des dégâts au corps à corps. Pour finir, le *ComportementNull* est un comportement qui ne fait rien, l'ennemi n'aura aucune réaction.

4.3.3 Un joueur aux multiples actions

Le joueur possède différentes actions « SANS_ACTION, ATTAQUE, SE_PROTEGE, INVINCIBLE ». Lorsque le joueur n'appuie pas sur la touche d'attaque ou de protection il reste dans l'état « SANS_ACTION ». Respectivement, la touche d'attaque fait passer le joueur en état « ATTAQUE » et « SE_PROTEGE » pour la touche de protection. L'état « INVINCIBLE » n'est pas utilisé actuellement mais pourra être utilisé, par exemple, lors d'une prise de bonus en jeu.

4.4 Gestion d'événements

4.4.1 Principe général

Les événements ont pour but de signaler l'intention de faire une action. Dans notre cas ils offrent un point d'extensibilité permettant de rajouter des événements sans avoir à modifier une classe existante. Les événements encapsulent leur traitement nous permettant par la suite d'utiliser le polymorphisme sans nous soucier réellement de l'instance de l'événement.

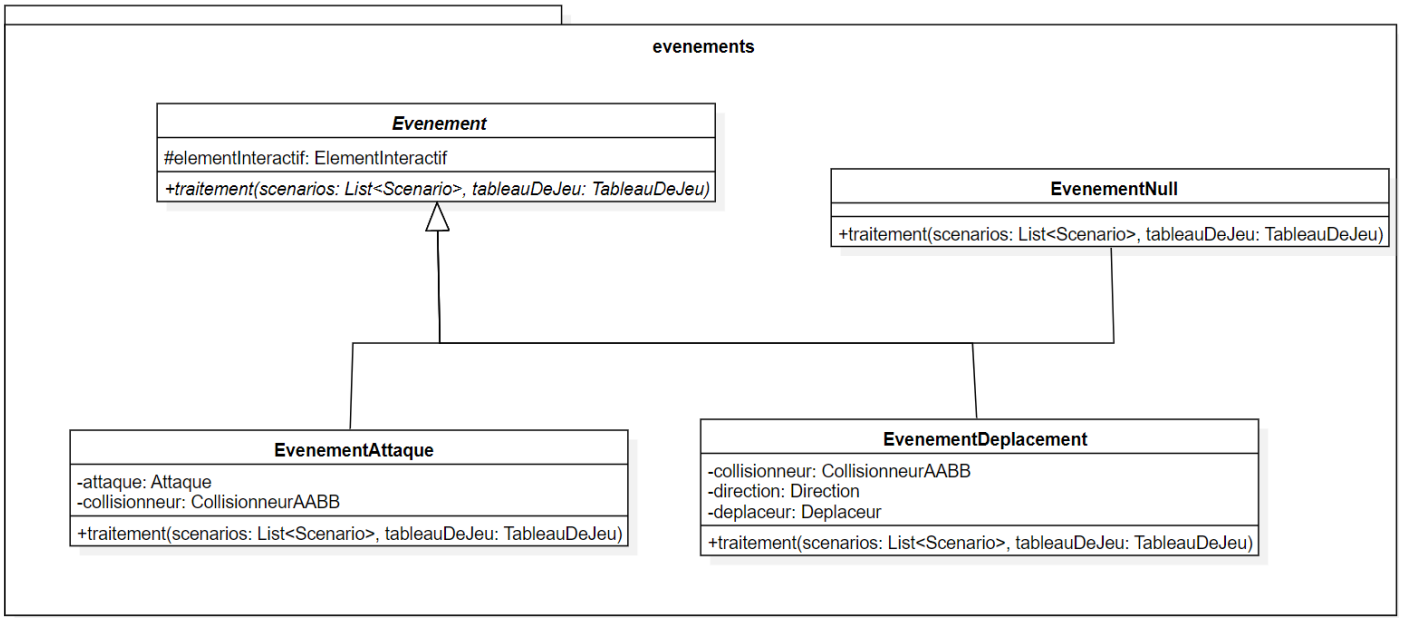


Figure 4.8 Diagramme de classes sur les événements

4.4.2 La nécessité d'un moteur qui centralise ces évènements

Nous avons créé un gestionnaire d'interactions pour centraliser la gestion des événements.

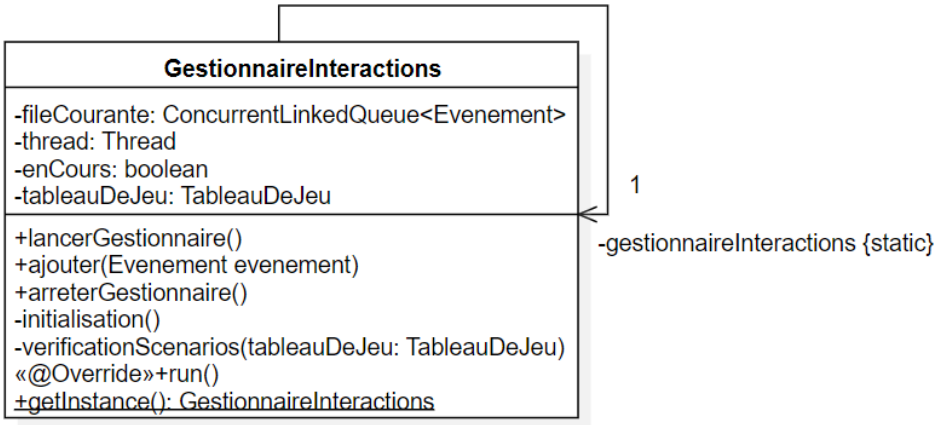


Figure 4.9 Diagramme de classe du gestionnaire d'interactions

Ce gestionnaire d'interactions tourne en parallèle dans un autre *Thread*. Cela signifie qu'on a le *thread* de la boucle de jeu qui tourne, ainsi que le *thread* qui vient traiter les évènements. Le gestionnaire implémente le patron de conception *Singleton* qui nous permet d'avoir une instance unique de la classe *GestionnaireInteractions*.

Les événements sont ajoutés dans une structure de données de type file (donc FIFO). Cela permet d'avoir un ordre de traitement des événements du « premier arrivé, premier traité », comme dans une file d'attente. Nous avons choisi cet ordre de traitement car l'évènement représente l'intention de réaliser quelque chose. Or, pour les collisions par exemple, cet ordre est important.

En effet, si on imagine que deux joueurs veulent aller à la même position, dans une situation comme cela :

Figure 4-8 Schéma d'une situation initiale avant déplacement.

Si P1 et P2 sont deux entités quelconques. Alors si P1 manifeste son intention de se déplacer en premier comme ci-dessous, et que P2 manifeste ensuite son envie de se déplacer :

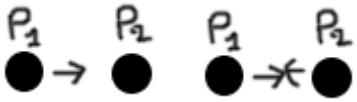


Figure 4-9 P1 veut se déplacer en premier, puis P2 veut se déplacer ensuite.

On voit qu'il va y avoir collision. On aura alors dans notre file l'évènement de déplacement de P1 avant celui de P2 :

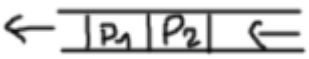


Figure 4-10 Etat de la file après ajout des évènements liés à P1 et P2.

Ainsi, l'évènement P1 sera traité en premier ce qui est normal, car P1 a manifesté en premier son intention de se déplacé et sera déplacé. P2 lui ne sera pas déplacé car il y aura une collision avec P1 qui sera déjà à son emplacement, et P2 ne pourra pas bouger.

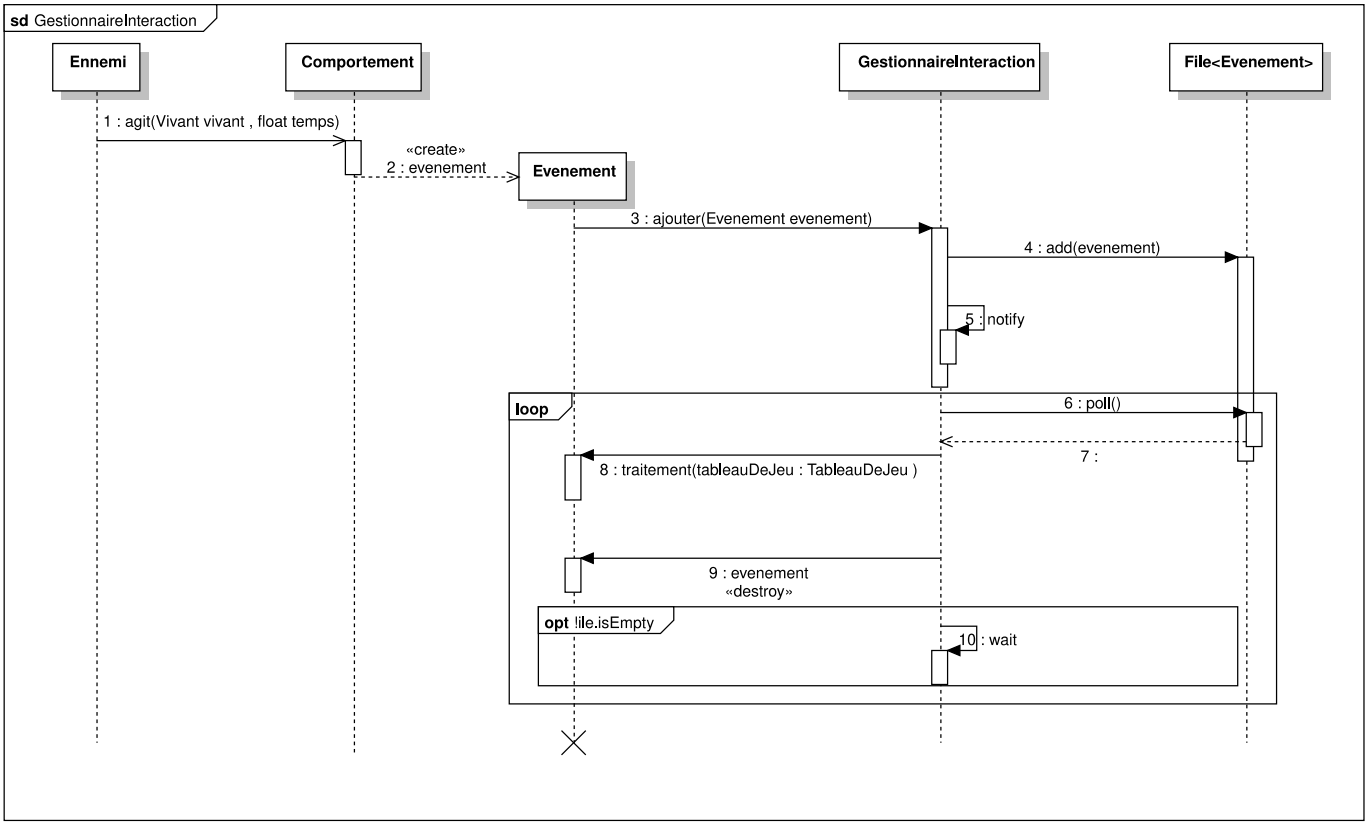
4.4.3 Les collisions

Les collisions sont un mécanisme apparaissant lorsque deux éléments se percutent. Nous avons implémenté des Collisionneurs pour encapsuler la vérification de collisions entre deux zones rectangulaires.

4.4.4 Gestion des événements

Ainsi dès qu'une entité veut bouger, on génère un évènement de déplacement qui vient indiquer l'intention qu'à l'entité de se déplacer. Cet évènement est alors inséré dans le gestionnaire et celui-ci va le traiter.

Cependant le gestionnaire tourne certes, en continu (dans une boucle *infinie*), mais il s'endort lorsqu'il n'a rien à traiter. Il est actif que lorsque sa file d'évènement n'est pas vide. Voici un diagramme de séquence qui résume son fonctionnement :



Lorsque des évènements sont ajoutés dans le gestionnaire d'interaction (que ce soit par un ennemi ou un joueur), on ajoute l'évènement dans la file et on vient notifier le *thread* de traitement qu'il a un évènement à traiter. Si celui-ci dort, cela le réveillera et permettra le traitement de l'évènement.

En interne, le gestionnaire tourne dans une boucle et traite chaque évènement de la file. Une fois traité, on retire l'évènement de la file et on le supprime. Si celle-ci est vide car il a fini de tout traiter, alors il s'endort jusqu'à ce qu'on le notifie pour lui dire qu'on a de nouveaux évènements à traiter.

La gestion des attaques se fait en utilisant le *CollisionneurAABB** qui va vérifier la collision entre la zone d'attaque de l'entité et un élément interactif. Cette vérification est faite pour tous les éléments interactifs provenant de la carte.

4.4.5 Les solveurs

Les solveurs de collisions et d'attaques ont été créés pour résoudre toutes les collisions de déplacements et d'attaques. Lorsqu'une collision est détectée, pour un déplacement ou une attaque, le solveur de plus haut niveau va récupérer le nom des deux classes envoyées en paramètre et va ensuite appeler le solveur de bas niveau correspondant. Il existe des solveurs pour chaque combinaison d'éléments interactifs, même entre deux éléments interactifs de même type. Chaque solveur de bas niveau gère ses cas indépendamment des autres solveurs. Il est ainsi possible d'ajouter des solveurs pour créer de nouveaux comportements aux classes qui sont ajoutées.

4.5 Gestion des entrées utilisateur

La gestion des entrées utilisateur a pour but de détecter quand le joueur appuie sur des touches du clavier. On vérifie si la touche appuyée correspond à quelque chose dans un dictionnaire. Si oui, la touche est ajoutée à une liste contenant les touches appuyées qui est ensuite traitée. Les touches trouvées sont interprétées par le jeu pour mener ou non à des résultats car le jeu possède un nombre de touches limité.

La classe *GestionnaireDeTouches* est abstraite et ne possède que la liste des touches appuyées. Pour ce jeu, nous utilisons un gestionnaire de touches spécialisé pour JavaFX. Cette détection de touche se fait via l'observation d'événement en utilisant le Framework JavaFX.

4.6 Elaboration d'une caméra

4.6.1 Problématique :

Nous voulons nous déplacer sur des cartes de grande taille et les afficher. Cependant, les afficher en entier rendrait chaque tuile trop « petite ». C'est pour cela que nous avions besoin d'une caméra, un objet qui permet de définir et de restreindre l'espace de jeu du joueur à une zone prédéfinie. La caméra définit ce que le joueur voit et ne voit pas pour pouvoir se concentrer précisément sur l'endroit où se déroule l'action.

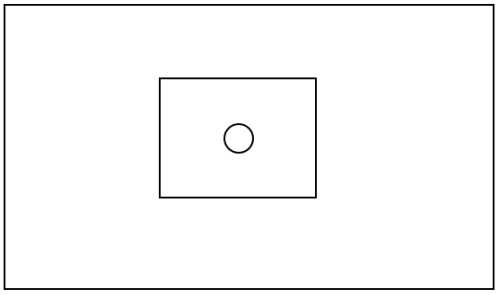
4.6.2 Fonctionnement

Cette caméra se centre sur le personnage jouable, de manière que l'utilisateur soit toujours au milieu de l'écran et à distance égale des bords de la fenêtre. La caméra se sert d'une carte et une zone observable.

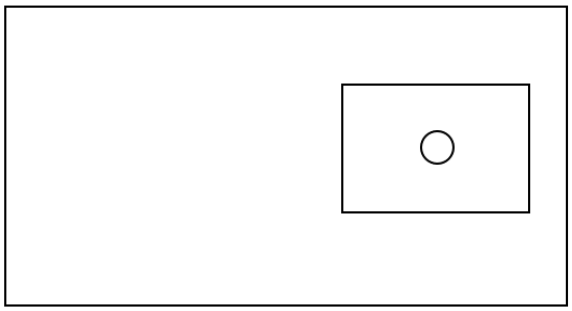
La carte correspond à la carte sur laquelle le personnage évolue. La zone observable correspondant à la taille de la caméra, la partie de la carte qui est affichée à l'écran. Les tuiles comprises dans la zone observable sont contenues dans un tableau à 3 dimensions. On a une dimension du tableau pour les calques, une pour la hauteur de la zone observable, et une pour sa largeur. Nous avons fait le choix du tableau, car la taille de la zone de la caméra est fixe et cela peut permettre de l'évolutivité. En effet, nous pourrions par exemple ajouter une dimension supplémentaire pour adapter la caméra à de la 3d. La

caméra suit une entité lorsqu'elle se déplace, le tableau de tuile visible est donc actualisé à chaque tour de la boucle de jeu.

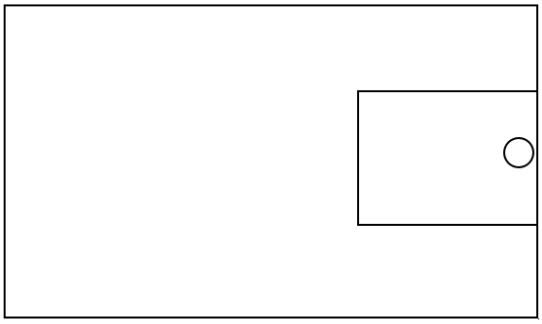
Quand l'entité est proche des bords de la carte, la caméra arrête de suivre ses mouvements, il est laissé « libre ». Quand l'entité suivie se rapproche des bords de la caméra, elle commence à le resuivre.



Caméra quand le personnage est au centre de la carte



Caméra quand le personnage est à droite de la carte



Caméra immobile et personnage « libre » quand le personnage est près des bords de la carte

L'affichage des tuiles et des autres éléments graphiques se fait au moment du centrage de la caméra, à chaque tour de boucle et cet affichage se fait en fonction de la position de la caméra pour qu'ils soient affichés au bon endroit.

Un enjeu important de la caméra est la gestion de la mémoire, en effet redessiner la totalité des éléments graphiques sur une carte de grande taille peut rapidement provoquer des problèmes de mémoire et donc des ralentissements en jeu. Cette caméra règle ce problème en n'affichant que les tuiles que le joueur a besoin de voir.

Encore une fois dans un souci de limitation d'utilisation de la mémoire par l'affichage, on n'affiche pas la première tuile du jeu de tuile qui est invisible.

4.6.3 Qualité de l'affichage

Pour finir, nous avions observé des baisses de la qualité des images lors des déplacements de la caméra. Ceci était dû à l'utilisation de double, les chiffres à virgules dans l'affichage, leur utilisation provoquait des erreurs d'arrondis ce qui provoquait un effet de flou. Pour résoudre ce problème, nous avons dû caster en entier les positions de dessins ce qui a permis de régler ce problème et d'obtenir un affichage net des éléments graphiques de la fenêtre de jeu.

Nous avons également dû faire en sorte que la taille du canvas corresponde avec la taille de la zone affiché par la caméra pour éviter d'afficher à l'écran l'extérieur des cartes.

Il gère différents types de tuiles et permet d'ajouter des zones de collision qui ne seront pas affichées à l'écran. Il permet également de gérer plusieurs calques de tuiles et de placer les tuiles au pixel près, plutôt que de la dimension d'une tuile, et également de gérer les transitions de terrain automatiques

4.7 Scriptage du jeu

4.7.1 L'intention du système de scriptage

Après avoir géré globalement le moteur de notre jeu avec les déplacements, collisions…, nous voulions ajouter des interactions, des éléments de *gameplay* concrets. Autrement dit, nous voulions pouvoir préciser par exemple qu'une attaque du personnage pointée vers un levier avait pour effet de faire apparaître deux ennemis (si on imagine que c'est un piège).

De la même manière, nous voulions intégrer bien d'autres interactions à notre jeu, qui seraient plus ou moins spécifiques. C'est ce qui permettrait d'égailler nos cartes, de fournir plusieurs façons de jouer.

Subséquemment, nous avons songé qu'il serait intéressant d'intégrer un système permettant à d'autres utilisateurs d'ajouter ou supprimer des *éléments* du jeu.

Nous ne voulions pas écrire en dur dans le programme tous les évènements qui seraient intégrés à notre jeu. Il nous semblait peu raisonnable de réaliser un jeu qui ne peut pas être modifié. Nous avons pensé à tous les utilisateurs qui aiment *modder* leurs propres jeux, afin de leur ajouter des comportements et des évènements. On peut par exemple évoquer Minecraft qui laisse la possibilité d'ajouter ses propres cartes, ajouter des objets etc.

L'idéal aurait été de laisser la possibilité à tous les utilisateurs de *modifier* le jeu, qu'ils puissent créer leurs propres cartes, leurs propres transitions entre ces cartes, qu'ils puissent les incorporer à leur jeu tout en configurant leurs touches comme ils le souhaitent.

Nous avons donc réfléchi à un système de *scriptage*, qui permettrait à l'utilisateur de créer son propre jeu de A à Z. Nous avons donc décidé de compléter notre projet afin de lui donner une couche d'abstraction supplémentaire : pour en faire un moteur de jeu.

4.7.2 Création d'un dossier dans le home de l'utilisateur

Pour atteindre cet objectif, il fallait pouvoir laisser la possibilité à l'utilisateur d'intégrer ses cartes, *tilesets* et images, ainsi que ses fichiers de configuration.

L'idée nous est donc venue de créer un dossier central, facilement accessible par l'utilisateur, et dans lequel il n'aurait qu'à déposer ses ressources pour que le prochain chargement du jeu génère tout ce qu'il aura placé à l'intérieur.

Nous avons créé ce dossier dans le *home* de l'utilisateur. Au lancement, un dossier **.mauja** est créé dans son dossier personnel. De cette manière l'utilisateur a facilement accès à ce dossier et peut déposer ses ressources à l'intérieur. On lui a fait prendre cette arborescence :

```
C:\USERS\JTREM\.MAUJA
├── cartes
├── configurationTouches.txt
├── images
│   └── tilesets
├── scripts
├── tilesets
└── transitions.txt
```

Arborescence du contenu du dossier .mauja créé dans le dossier personnel de l'utilisateur.

On y distingue un dossier relatif aux cartes que va pouvoir incorporer l'utilisateur. On voit également qu'il peut incorporer ses jeux de tuiles et images dans les dossiers tilesets et images/tilesets. Enfin, un fichier de scripts est présent dans lequel il va pouvoir spécifier les règles du jeu.

Nous avons donc fait en sorte que notre jeu vienne toujours se servir en ressources dans ce dossier. L'utilisateur n'a qu'à déplacer ses fichiers à l'intérieur. S'il supprime des fichiers de configurations ou des dossiers, ils seront alors automatiquement recréés. Par défaut si le dossier est vide, le programme vient alors copier des fichiers de manière à ce que l'utilisateur puisse avoir une petite version de démonstration.

Nous avons ensuite pu réaliser le système de transitions entre les cartes.

4.7.3 Transition entre les cartes

Nous avons également pris la décision de rendre les transitions entre les cartes facilement modifiable et ajoutable au jeu avec toujours la même logique : utiliser un fichier pour renseigner ses transitions.

1. carteTest4.tmx 15 5
2. carteTest3.tmx 3 8 => carteTest4.tmx 16 17
3. carteTest4.tmx 4 4 => carteTest3.tmx 16 15
4. test.tmx 14 14 => marais.tmx 4 8
5. marais.tmx 26 20 => carte-exeterieur.tmx 77 106
6. carte-exeterieur.tmx 98 74 => carteVillage.tmx 21 11
7. carteVillage.tmx 21 22 => carteTest.tmx 11 14

8. carteTest4.tmx 28 19 => test.tmx 12 14

Fichier contenant les transitions du jeu.

Vous pouvez voir ici le contenu du fichier de transitions, à la ligne 1 on a renseigné le nom de la première carte du jeu suivi de la position d'apparition du joueur sur cette carte. Les lignes suivantes représentent de véritables transitions, dans le format suivant : nom de la carte de départ suivi des coordonnées de départ. De l'autre côté de la flèche => on retrouve le nom de la carte et les coordonnées d'arrivée de la transition.

4.7.4 Règles de jeu

Nous avons décidé de créer des règles pour mettre en place l'histoire du jeu. Ces règles vont ensuite être interprétées par le gestionnaire d'interactions.

4.7.4.1 Création d'un fichier pour décrire les règles

L'idée que nous avons eu est de créer un fichier xml, pour stocker les règles, composé de balises également imaginées par nos soins. Les balises dont nous avons eu besoin sont principalement : *ElementInteractif*, *Action*, *Condition*, *Scénario.*

```
<Carte id="1" end="oui">
   <Scenario end="oui">
      <ElementInteractif type="com.mauja.maujaadventures.interactions.elements.Levier"
active="false"
                        nombreEtats="3" id="1" end="non">
         <collision type="com.mauja.maujaadventures.logique.Rectangle" x="0" y="0"
largeur="40" hauteur="40" end="non"/>
         <position type="com.mauja.maujaadventures.logique.Position" x="32" y="32"
end="oui"/>
         <Condition
type="com.mauja.maujaadventures.interactions.conditions.ConditionCollision" end="oui"/>
         <Action
type="com.mauja.maujaadventures.interactions.actions.ActionActivationLevier" idEffet="1"
end="oui">
            <ElementInteractif type="com.mauja.maujaadventures.entites.Ennemi"
xEn="300" yEn="500" largEn="10" hautEn="10"
                              xCol="10" yCol="10" largCol="10" hautCol="10" id="2"
                              pointsDeVie="3" end="oui">
            </ElementInteractif>

            <ElementInteractif type="com.mauja.maujaadventures.entites.Ennemi"
yEn="400" xEn="500" largEn="10" hautEn="10"
                              xCol="10" yCol="10" largCol="10" hautCol="10" id="2"
                              pointsDeVie="3" end="oui">
```

```
            </ElementInteractif>
        </Action>
        <Action
type="com.mauja.maujaadventures.interactions.actions.ActionTeleportationJoueur" idEffet="1"
end="oui"/>
    </ElementInteractif>
  </Scenario>
  <Scenario end="oui">
  </Scenario>
</Carte>
```

Fichier contenant les règles du jeu.

Comme vous pouvez le voir dans l'exemple de fichier si dessus, nous avons une sorte d'inclusion l'hiérarchique de nos balises. Il nous a donc fallu créer une hiérarchie similaire au niveau de notre conception.

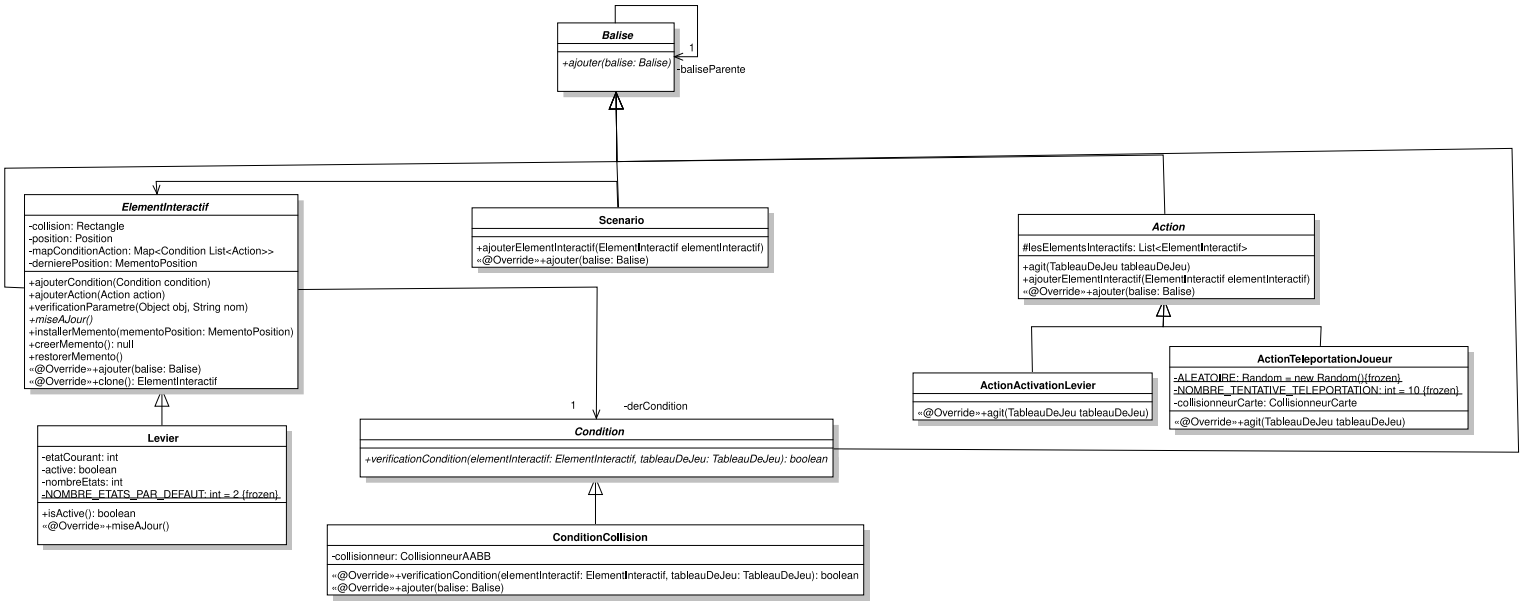


Figure 4.10 Diagramme de classes pour la logique des interactions

Ici nous avons créé les éléments interactifs, ces derniers représentent tous les éléments qui peuvent être activés par le personnage (par exemple un levier), et également tous ce qui peuvent subir l'action d'une activation (par exemple une entité). Les conditions, décrivent ce qui doit être effectué pour qu'une action soit déclenchée. Les actions réalisent grâce à la méthode agit le traitement qui doit être appliqué aux éléments interactifs contenus dans l'action.

4.7.4.2 Parsage du fichier de règles

Pour intégrer au jeu les règles écrites dans le fichier il nous a fallu créer un parseur. Nous avons utilisé la parseur SAX qui était très utile pour parser la hiérarchie présentée précédemment car nous effectuons un parsage en continu.

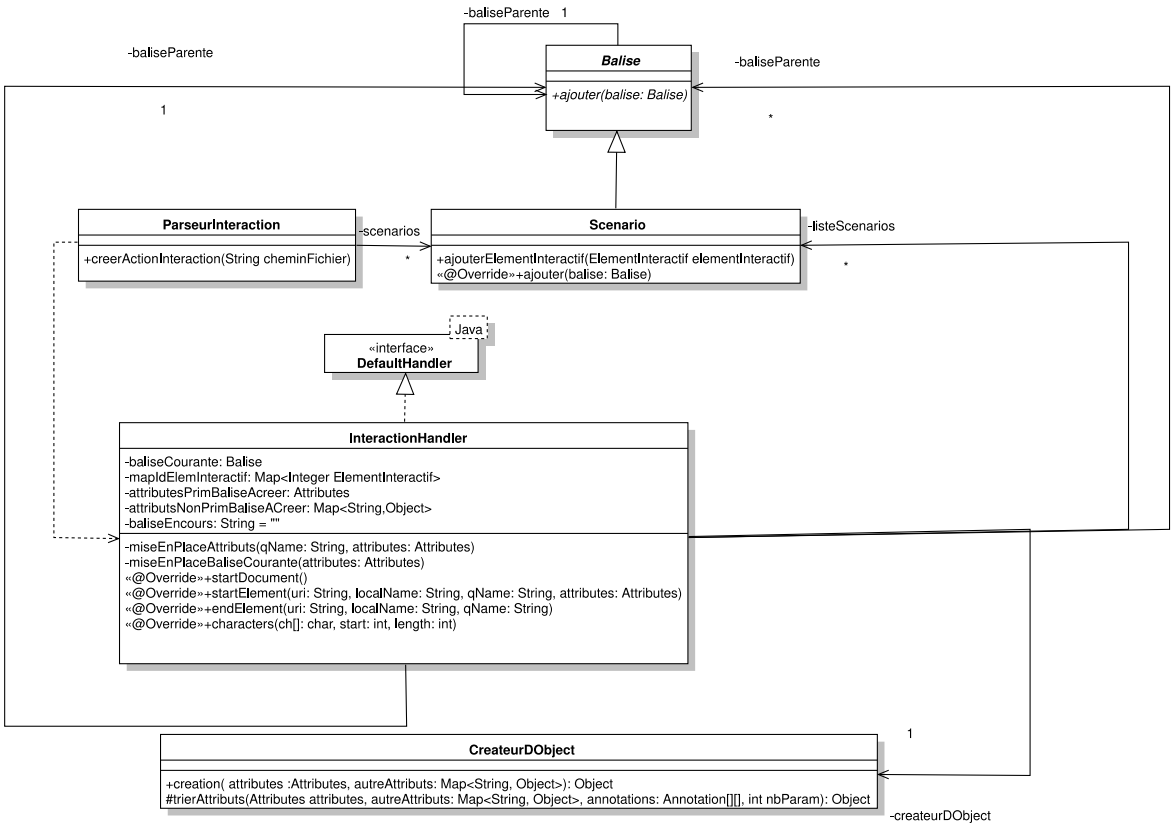


Figure 4.11 Diagramme des classes responsable du parsage

L'IinteractionHandler qui implémente le *DefautHandler* est le cœur du parseur, c'est grâce à lui que l'on va pouvoir imbriquer les balises les unes dans les autres. A chaque balise ouvrante on instancie une balise, à chaque balise fermante on l'ajoute dans sa balise parente.

```
@Target(ElementType.CONSTRUCTOR)
@Retention(RetentionPolicy.RUNTIME)
public @interface ConstructeurXml { }
```

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface Param {
    String nom();
    Class<?> classe() default Double.class;
    boolean estPrimitif() default true;
}
```

Nous utilisons l'introspection pour créer les différents Objets, la création est faite grâce au *CreateurDObjet*. Nous avons une annotation pour savoir quel constructeur utiliser lors de l'instanciation et une autre pour connaitre le type, le nom, et le fait que l'attribut soit primitif ou non.

4.7.4.3 Ajout des scénarios dans le gestionnaire d'interactions

Les scénarios créés grâce au parseur sont ensuite ajoutés au gestionnaire d'interactions. Le gestionnaire d'interactions va venir vérifier chaque scénario à chaque fois qu'un événement est traité. Si un événement vient valider une condition alors l'action associée à cette condition va alors être traitée.

Le jeu peut grâce aux scénarios se dérouler, tout en respectant les règles décrites dans le fichier *xml*. On peut imaginer que ce fichier de règles sera généré en utilisant un logiciel que nous allons concevoir dans le futur.

4.7.5 Configuration des touches

Nous avons également mis en place un fichier qui permet de configurer les touches avec lesquelles l'utilisateur va jouer. Ainsi le joueur peut modifier son contenu pour définir ses propres touches.

```
# Déplacements.
KP_LEFT : FLECHE_GAUCHE
KP_RIGHT : FLECHE_DROITE
KP_UP : FLECHE_HAUT
KP_DOWN : FLECHE_BAS

LEFT : FLECHE_GAUCHE
RIGHT : FLECHE_DROITE
UP : FLECHE_HAUT
DOWN : FLECHE_BAS

# Pour faire demi-tour dans les menus.
ESCAPE : ECHAP

# Attaque.
SPACE : ESPACE

# Défense.
B : B
```

Fichier de configuration des touches

Vous pouvez voir ici le contenu du fichier de configuration. L'utilisateur doit simplement respecter le format KEYCODE_JAVA_FX : TOUCHE_MAUJA. Ce fichier est ensuite lu et interprété pour permettre au joueur de se déplacer, de se défendre et d'attaquer en utilisant les touches renseignées dans le fichier.

5 Bilan Technique

Pour réaliser notre jeu vidéo en 2d, nous avons donc réalisés une boucle de jeu à l'aide d'un thread, nous avons créé des cartes avec le logiciel *Tiled* et nous les avons affichés dans une interface graphique.

En parallèle, nous avons réalisé une application de sélections d'images, également avec *JavaFX* pour pouvoir créer nos propres jeux de tuiles.

Nous avons ensuite créé de multiples entités, parmi lesquels des vivants, comme le personnage jouable et des ennemis et des destructibles comme les projectiles. Nous avons ensuite créé des comportements aux ennemis pour leur donner un but dans le jeu et des actions pour le joueur pour lui permettre d'interagir avec son environnement.

Nous avons ensuite réalisé un gestionnaire d'évènement tournant dans son propre thread pour pouvoir centraliser la gestion de ces évènements. Parmi ces évènements, on retrouve les collisions, les déplacements ou encore les attaques. Nous avons également créé des solveurs pour gérer les collisions. Nous avons ensuite dû récupérer et gérer les entrées utilisateurs à l'aide de pour permettre l'interaction entre le joueur devant sa machine et le jeu.

Pour pouvoir afficher la carte en fonction des déplacements du personnage et pour ne pas avoir à afficher la carte en entier, nous avons réalisés une caméra qui ce centre sur le personnage jouable. Cette caméra affiche le personnage au centre de la carte à chaque tour de boucle.

Nous avons également réalisé un système de scriptage pour ne pas avoir à conserver des données du jeu dans le programme. Nous externalisons ces données dans des fichiers texte ce qui peut permettre à n'importe qui de créer et de personnaliser ses propres cartes et niveaux.

Pour finir, nous avons rendu possible la transition entre les cartes à partir des fichiers de scriptage et entre les différentes fenêtres de vues à partir du menu.

6 Conclusion

S'il fallait synthétiser rapidement nos ressentis après 4 mois, ce projet a été sans aucun doute très enrichissant, et ce, sur divers points. Tout d'abord, il nous a permis de participer à un projet d'une dimension inédite pour nous, que ce soit dans sa longueur dans le temps (4 mois) ou dans le nombre de personnes y participant (5 avec une personne en plus jouant un rôle de client). En effet, avant cela, les projets étudiants précédents se faisait par plus petit groupes et cette expérience était sans aucun doute différente.

Nous avons pu constater quels pouvait être les avantages et les inconvénients de travailler dans un groupe de travail étendu dans un projet assez long. Ce projet a également été l'occasion de découvrir et d'expérimenter la méthode de travail agile Scrum. Cette méthode nous a permis de bien aborder la manière de réaliser un tel projet où il y aurait des risques que certaines personnes se retrouve à l'écart ou encore que des ralentissements sur certaines tâches provoquent un retard général.

Nous n'avons pas toujours été parfait au niveau de notre organisation, mais au fil du temps nous avons réussi à mieux nous approprier la méthode pour pouvoir avancer plus efficacement et nous en tirons tous une expérience positive. Durant toute la durée du projet, nous nous sommes parfois un peu écartés du projet originel pour résoudre des imprévus et nous avons eu de nouvelles idées en cours de route, ce qui nous a permis d'apprendre sur des sujets inattendus comme lorsque nous avons réalisé le système de scriptage ou encore quand nous avons réalisé une application de sélections d'images.

Nous avons bien entendu beaucoup appris également sur le sujet que nous souhaitions concevoir à l'origine. Nous avons vu plus en profondeur comment un jeu vidéo fonctionnait et nous avons pu mettre en pratique des concepts de programmation vu en cours avant et pendant ce projet. Nous avons même pu découvrir et mettre en œuvres des spécificités de Java comme l'introspection, que nous n'avions jamais réellement abordé avant. Nous avons même pu entrevoir à la fin du projet la manière de réaliser un moteur de jeu complet.

Pour conclure, nous avons tous beaucoup appris lors de ce projet et de manière générale nous sommes assez fiers de ce que nous avons réalisé et nous en garderons un bon souvenir.

7 Summary of the project in English

This project is part of our two-year university diploma in Computer Science at University Institute of Technology, UCA in Clermont-Ferrand. As part of our studies, we are currently doing a four-month project on a 2D video game using the java language in a group of five. We used the agile method Scrum.

The goal of the project was to create a 2D video game, but during the project, our objective evolved, we now wanted to develop a video game engine so anybody could create his own game based on what we did. We developed this engine entirely from scratch.

First, we had to decide how the game was going to work. We chose to use a game loop to control the life cycle of our game and so we learned how we could implement it. Secondly, we thought about how we would display images. We decided to use JavaFX (a library for creating desktop applications) and a technology called the tile mapping. As the name suggests, it allows you to display game maps by using small images called tiles. Then, we had to manage the movements of the main character as well as the behaviors of the enemies. Finally, we worked on a way to script the game (to write in an external file the story, what's possible in the game and how you do it. This way, we were able to set up which keystrokes we were going to use, which creatures and which interactive elements (levers, doors…) are present, where they are on the map and when they appear.

Our supervisor, Mister Provot went through the project brief with us so that we could get to work on the project. During the brief we decided various things, such as the working method. As stated previously we used the Scrum method. Then, we split the work in different tasks, and we shared those tasks between the members of the group. On average, we had 2 weeks long working periods called 'sprints', where we had an objective, something we had to present. At the end of the sprint we showed the result to our client with a little demo then we talked about what we were going to do during the next sprint. Mister Provot played the role of a customer. Our game engine has been programmed using the Java language. We used IntelliJ, an Integrated Development Environments (IDE) to write code.

To conclude we created a little 2D video game engine, with it, the user can create his own video game with his own map created with tiled and with his own story written in the script files. We created some maps and some stories to show how the game engine could be used. If we had some more time, we would have added the possibility of playing with several players and upgraded the layout of the game view, we would especially have liked to make it responsive. Overall, the project was a good learning experience and a big challenge for us, because we did brand new things like creating a game loop or an event manager system.

8 Bibliographie / Webographie

OpenGameArt (site de ressources graphiques) https://opengameart.org/

Itch.io (site de ressources graphiques) https://itch.io/game-assets

Javadoc docs.oracle.com https://docs.oracle.com/en/java/javase/17/docs/api/

javaFx https://openjfx.io/

Tiled site https://thorbjorn.itch.io/tiled?download

GitHub de versions Tiled https://github.com/mapeditor/tiled/tags

TiledReader http://www.alexheyman.org/tiledreader/

Scrum depuis les tranchées http://agile-lean-et-compagnie.com/wp-content/uploads/2016/01/Scrum-et-XP-depuis-les-tranchees-v2.pdf

9 Lexique

Tile-mapping : Technique d'assemblage de tuile pour créer la dimension graphique d'un jeu viédo.

Tilemap : ensemble d'image graphique disposées sur une grille

Tileset : Jeu de tuile en français, ensemble de tuiles.

Tile : Tuile en français, petite image carrée composant d'environnements graphiques de jeu-vidéo.

Introspection : L'introspection consiste en la découverte, de façon dynamique, des informations propres à une classe Java.

JavaFX : Framework Java permettant de créer une interface graphique.

Git : Logiciel de versioning de fichiers. Le versioning fait référence à l'utilisation et à la gestion de plusieurs versions d'un même document.

Repository : Dépôt en français, stockage centralisé des données d'un projet Git.

Thread : Unité d'exécution d'un programme fonctionnant de manière autonome et de manière parallèle avec d'autres unités d'exécutions.

Toolkit Graphique : Interface graphique permettant la manipulation de ces objets

Collision AABB : Méthode permettant de savoir si 2 objets sont en collisions en se basant sur leurs aires respectives.

Scriptage : C'est le fait de garder des données en dehors d'un programme dans un fichier d'initialisation.

FPS : frame per second ou image par seconde en Français. Nombre de rafraîchissement de l'image chaque seconde.

Scénario : histoire d'un jeu vidéo.

Règle du jeu : règle consistant à contrôler un jeu.

Bibliothèque : Ensemble de fonctions utilitaires, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.

Parsage : lecture d'un fichier et mise en place d'éléments dans le jeu suit à la lecture

Script : histoire du jeu.

Moteur de jeu : base qui regroupe et gère en temps réel les fonctionnalités principales d'un jeu vidéo.

Patron de conception : un patron de conception est la meilleure solution connue à un problème de conception récurrent.

Patron Observateur : patron de conception permettant d'observer l'état d'un objet en faisant de l'attente passive.

Patron Singleton : patron de conception permettant de garantir une instance unique d'une classe.

Patron Stratégie : patron de conception permettant de modifier l'algorithme d'une méthode à l'exécution.

CollisionAABB : 2 rectangles de collisions orthogonales qui s'entrechoquent.

Calque : couche de tuiles.

XML : Langage de structuration de données.

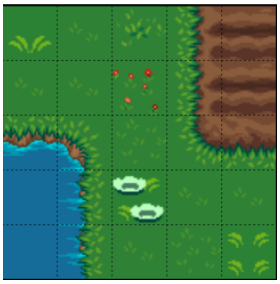
Modder : possibilité de modification par une personne tierce d'un jeu vidéo.

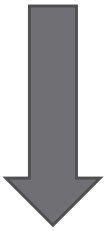
Mod : modification par une personne tierce d'un jeu vidéo.

10 Annexes

Le logiciel Tiled

Vous pouvez voir ci-dessous comment le logiciel converti une petite carte dessinée depuis le logiciel en XML :



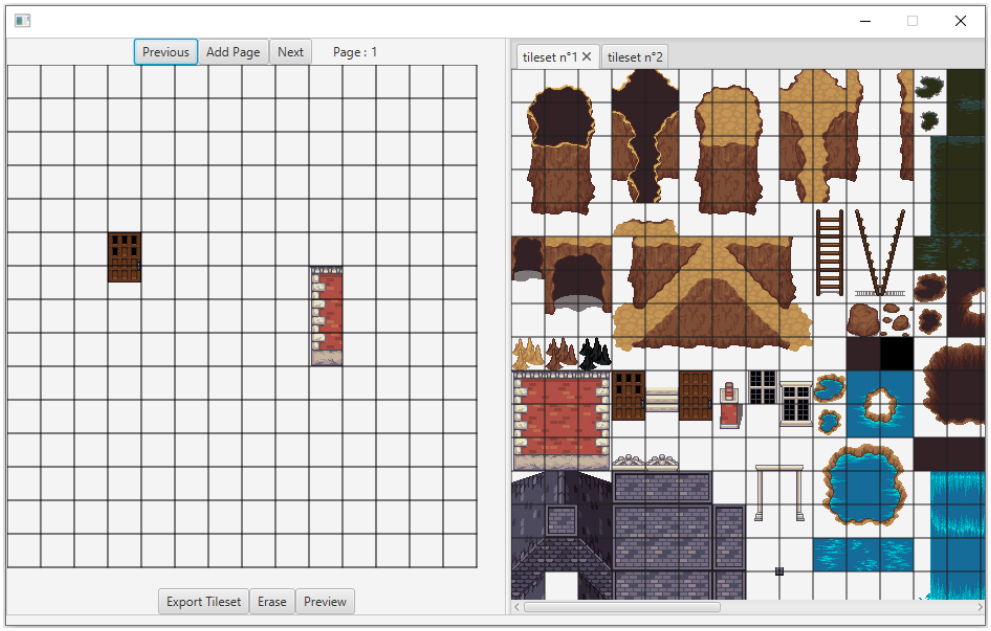


```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.4" tiledversion="1.4.3" orientation="orthogonal" renderorder="right-down"
width="5" height="5" tilewidth="32" tileheight="32" infinite="0" nextlayerid="2"
nextobjectid="1">
    <tileset firstgid="1" source="../.mauja/tilesets/terrain_atlas.tsx"/>
        <layer id="1" name="Sol" width="5" height="5">
            <data encoding="csv">
                802,184,182,582,583,
                184,184,376,582,583,
                360,361,183,614,615,
                392,393,805,184,184,
                392,393,184,183,803
        </data>
    </layer>
</map>
```

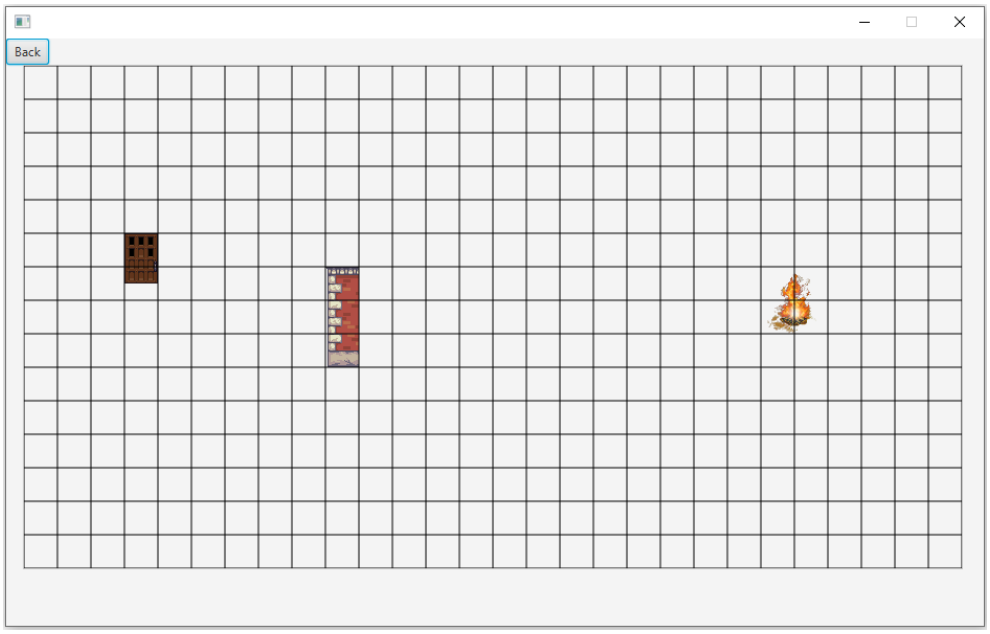
Comme expliqué précédemment, nous avons utilisé la bibliothèque *TiledReader* pour lire ce fichier et récupérer toutes ces données, à savoir, les tuiles affichées sur la carte, ainsi que les différentes dimensions, chemins et calques.

Logiciel de sélection d'images :

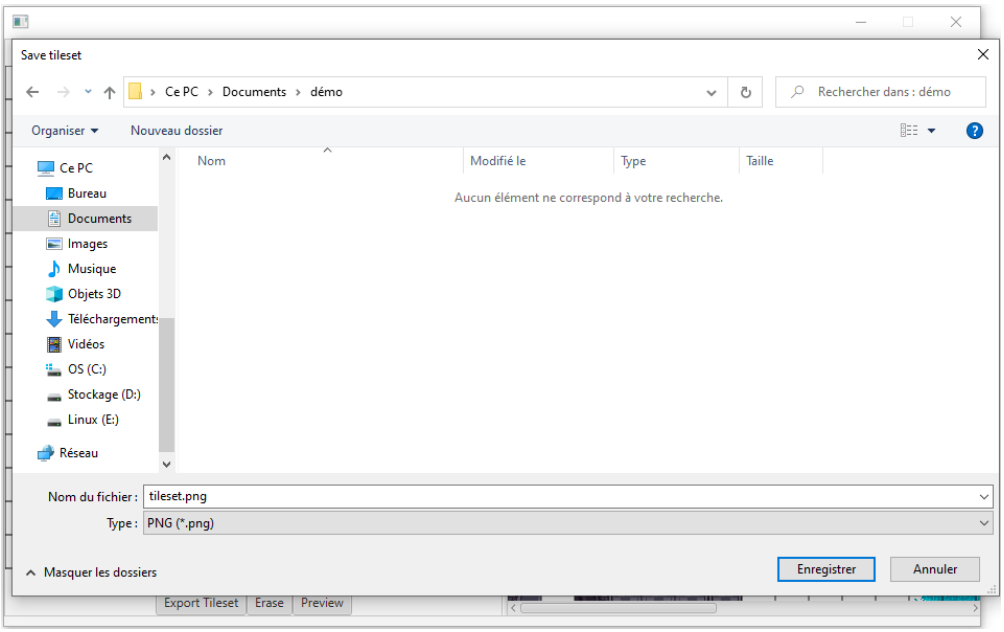
Pour pouvoir dessiner sur la grille vierge il suffit de sélectionner une tuile graphique provenant d'un tileset et ensuite de sélectionner l'endroit où l'on souhaite la placer.



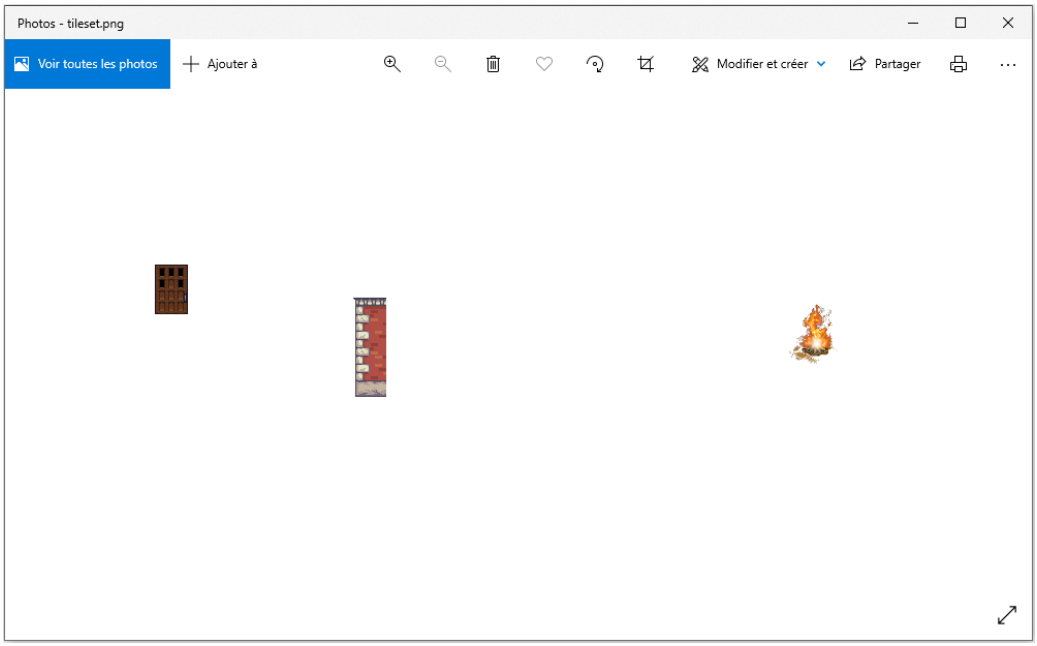
La prévisualisation colle toutes les pages du catalogue et permet de visualiser le rendu avant exportation.



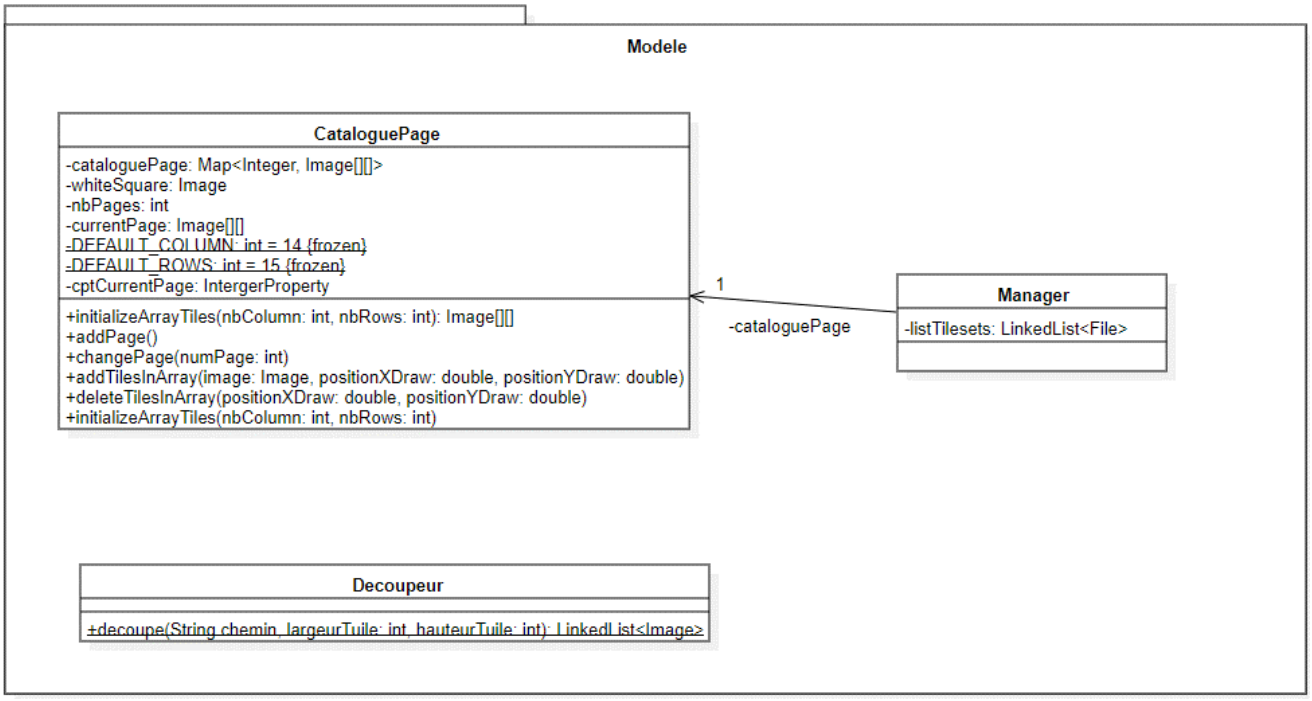
Si le résultat est satisfaisant, il suffit de revenir sur la vue d'édition pour ensuite exporter le tileset. Cela permet d'enregistrer le tileset au format png ou jpg à l'endroit souhaité.



Le nouveau tileset comprend donc des tuiles graphiques provenant de différents tilesets et peut être maintenant utilisé pour créer nos cartes de jeu.



Pour ce qui est du modèle de conception, il est quant à lui simple. La majeure classe de l'application est « CataloguePage » représentant notre catalogue de dessin. La classe « Decoupeur » nous sert à découper les tilesets sélectionnés en plusieurs images de dimensions 32par32 pixels.



Prévisionnels individuels P3 :

Viton Antoine :

Num	Tâches et activités	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11
3-3	Création des cartes										
431-2	Création de l'objet 2										
416-1	Création des ennemis										
416-2	Création des projectiles										
416-4	Création des IA Et attaques										
416-5	Création des boss										
422	Menu principal										
418	Persistance										
417	Jeu en coopération en réseau local										
423	Animation du pers et des ennemis										
424	Ecran GAME OVER										
431	Objets d'intéractions										
432	Intéractions du personnage sur l'environnment										
433	Pièges										
434	Création de trésors et bonus										
435	Gestion des objets clés										
6	Déploiement										

Coudour Adrien :

Num	Tâches et activités	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11
3	Création des cartes										
431	Création de l'objet										
422	Menu principal										
418	Persistance										
417	Jeu en coopération en réseau local										
423	Animation du pers et des ennemis										
431	Objets d'intéractions										
433	Pièges										
435	Gestion des objets clés										
436	Personnage non-joueur										
	Total	2h	2h	3h	3h	3h	7h	4h	1h	3h	2h
Adrien COUDOUR G7											

WISSOCQ Maxime :

Num	Tâches et activités	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11
3 – 3	Création des cartes										
431	Objets d'interaction										
416-4	Création des IA Et attaques										
416-1	Création des ennemis										
416-5	Création des boss										
417	Jeu en coopération en réseau local										
423	Animation du personnage et des ennemis										
435	Gestion des objets clés										
436	Personnages non-joueurs										
6	Déploiement										

Vignon Ugo :

Num	Tâches et activités	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
3	Création des cartes										
416	Ennemis, Boss et attaques										
422	Menu principal										
417	Jeu en coopération en réseau local										
432	Intéractions du personnage sur l'environnment										
431	Objets d'intéractions										
418	Persistance										
423	Animation du personnage et des ennemis										
424	Ecran GAME OVER										
434	Création de trésors et bonus										
433	Pièges										
435	Gestion des objets clés										
436	Personnage non joueur										
6	Déploiement										
421	Adaptation de la fenêtre										
425	Animation des tuiles										

Prévisionnel commun P3 :

Num	Tâches et activités	S1 (10/01 - 16/01)	S2 (17/01 - 23/01)	S3 (24/01 - 30/01)	S4 (31/01 - 6/02)	S5 (7/02 - 13/02)	S6 (14/02 - 20/02)	S7 (21/02 - 27/02)	S8 (28/02 - 06/03)	S9 (07/03 - 13/03)	S10 (14/03 - 20/03)
416	Ennemis, Boss attaques										
3	Création des cartes										
431	Objets d'intéractions										
421	Adaptation de la fenêtre										
422	Menu principal										
418	Persistance										
417	Jeu en coopération en réseau local										
424	Ecran GAME OVER										
432	Intéractions du personnage sur l'environnment										
425	Animation des tuiles										
423	Animation du personnage et des ennemis										
433	Pièges										
435	Gestion des objets clés										
436	Personnage non-joueur										
434	Création de trésors et bonus										
6	Déploiement										

Réalisés individuels :

Vignon Ugo :

Numéro de projet : 20	S1 (8/11 – 14/11)	S2 (15/11 – 21/11)	S3 (22/11 - 28/11)	S4 (29/11 - 5/12)	S5 (6/12 - 12/12)	S6 (13/12 - 19/12)	S Vacances (20/12 - 02/01)
Réalisation d'un modèle du domaine.							
Déplacement du personnage							
Se documenter							
Déplacement du personnage sur la carte							
Chargement de la carte sous forme de tuiles							
NP** - Organisation pour le sprint							
NP** - Réunions et discussions							
414-2 Limitation des mouvements du personnage							
Chargement des jeux de tuiles							
NP** - Présentation de la version de							
Regroupement des images sélectionnées							
3-2 Création des collisions							
3-3 Création des cartes							
découpage d'images							

Tremblay Jérémy :

	S1 (8/11 - 14/11)	S2 (15/11 - 21/11)	S3 (22/11 - 28/11)	S4 (29/11 - 5/12)	S5 (6/12 - 12/12)	S6 (13/12 - 19/12)	S Vacances (20/12 - 02/01)
Réalisation d'un modèle du domaine.							
Réalisation du diagramme de classes							
Rédaction du Backlog de produits							
Affichage de la carte							
Se documenter							
Recherche de ressources graphiques							
Chargement de la carte sous forme de tuiles							
NP** - Organisation pour le sprint							
Chargement des jeux de tuiles							
Modification de l'affichage de la carte							
NP** - Présentation de la version de démonstration							
NP** - Réunions et discussions							
Regroupement des images sélectionnées							
414-2 Limitation des mouvements du personnage							
53 Optimisation*							
3-1 Création d'un logiciel de découpage d'images							
416-2 Création des projectiles							
416-3 Gestion des collisions							
416-4 Création des IA et attaques							
415 Système de combat							
412- Adaptation de la fenêtre							
431-1 Création de l'objet 1							
3-3 Création des cartes							

Maxime Wissocq :

	S1	S2	S3	S4	S5	S6
Se documenter						
Gestion des déplacements du personnage						
Réalisation de sketchs						
413 -1 Déplacement de la caméra						
Découpage de la carte						
NP Réunions et discussions						
413-2 Centrer la caméra sur le personnage						

Coudour Adrien :

		COUDOUR Adrien Numéro de Projet n°20					
			Réalisé individuel P2				
Taches et activités	S1	S2	S3	S4	S5	S6	
Se documenter							
Affichage du personnage							
Déplacement de la caméra							
Réunions et discussions							
Création de méthode manquante + documentation du code							
Gestion de la caméra							
Présentation de la version de démonstration							
Optimisation							
3-1 Création d'un logiciel de sélection d'image							
Total	6h	7h	7h	6h	4h	0h	30