RAPPORT FINAL - PROJET LONG

Ystoria - Logiciel de création de jeux vidéo 2D (RPG)

Auteurs :

BEDROSSIAN Paul
AUGUSTO Arthur
BECHU Alex
LAURENT Maxime
MONTALDO Alexandre
BROCHARD Titouan
JAUSSET Nathan

Table des matières

Introduction

Dans le cadre de l'UE Technologie Objet et de l'option Méthodes Agiles du module de CAM, nous avons dû réaliser un travail de groupe nommé Projet Long. Cet exercice a pour but aussi de vérifier que nous maîtrisons les connaissances techniques acquises lors du semestre, mais aussi que nous sommes capables de réaliser un travail de groupe à la manière d'un travail dans le monde professionnel. Nous avons pu par la passé lors des 2 rapports d'itération énoncer et notifier notre avancement sur notre projet, qui nous le rappelons est un logiciel de création de jeux vidéo 2D sans connaissance en programmation. Maintenant, étant arrivé à la fin de notre période de développement de projet, il est temps pour nous de présenter notre rendu final. Ainsi, ce rapport fera l'état des lieux de notre projet. Nous présenterons dans ce document l'architecture de notre logiciel, les principales fonctionnalités, les problèmes rencontrés lors de leurs développements et les solutions apportées.

Rappel du sujet

Nous avons créé un logiciel de création de jeux vidéo 2D, c'est-à-dire qu'un utilisateur sans compétence informatique pourra créer son jeu incluant une carte, un personnage pouvant se déplacer et des intéractions.

Principales fonctionnalités

Grâce à notre application, l'utilisateur pourra créer un jeu vidéo de type *Role Play Game.* Cela nécessite 2 outils à développer : un logiciel d'édition (le *Game Editor*) et un logiciel qui est capable de comprendre le jeu créé par le *Game Editor* et de permettre au joueur de jouer : le moteur (le *Game Engine*). Ces deux logiciels ont été développés en même temps (même si le développement du *Game Editor* à été un peu plus rapide que celui du *Game Engine*). Voici un récapitulatif des principales fonctionnalités et de leur avancement :

Fonctionnalités	*Game Engine*	*Game Editor*	Itération
Cartes (avec tuiles)	✅	✅	1 & 2
Personnages	✅	✅	2
Compétences	❌	✅	2
Attributs des personnages	❌	✅	2
Dialogues	❌	✅	3
Combats	❌		
Point de départ	✅	✅	3
Téléportation entre les cartes	✅	✅	3
Déplacement	✅		3
Inventaire	❌	❌	
Objets	❌	❌	
Monstres	❌	✅	2
Quêtes	❌	❌	
Fin du jeu	❌	❌	
Musiques	❌	❌	
Sauvegarde d'une partie	❌		

Fig. 1 - Tableaux des principales fonctionnalités et leur avancement

Fonctionnement de l'application

Notre application est séparée en trois applications différentes : le *Game Engine* (pour faire tourner le jeu), le *Game Editor* (pour créer le jeu) et enfin le *Launcher* (pour choisir l'endroit où l'on veut stocker le jeu). D'un point de vue conception, nous avons choisi de séparer le *Game Engine* et le *Game Editor* afin que le *Game Engine* n'ait pas besoin du *Game Editor* pour fonctionner. L'inverse n'est pas vrai puisque depuis l'éditeur nous pouvons lancer le jeu afin de le tester en parallèle de sa création. Cette solution reste néanmoins discutable car elle a mené à la création de classes *Java* en double.

Fonctionnement global

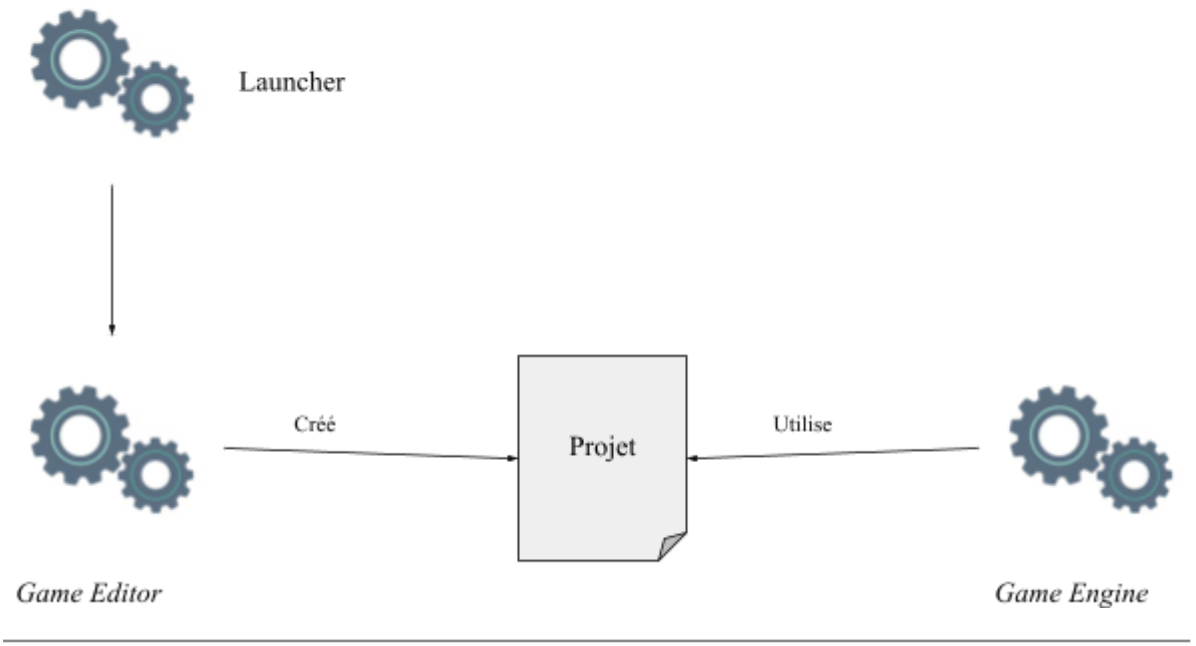


Fig. 2 - Schéma de fonctionnement global

Comme le montre le figure 1, le *Launcher* permet de lancer le *Game Editor*. L'éditeur crée et modifie un dossier contenant l'ensemble des informations du jeu que l'on est en train de créer. Le *Game Engine*, lui, est capable d'interpréter le dossier pour faire fonctionner le jeu que l'on a créé.

Système de fichiers

Dans cette section, nous allons aborder le système de fichiers que nous utilisons pour stocker les données des jeux créés. Le dossier est composé de deux sous dossiers :

- ***assets***: ce dossier permet de stocker l'ensemble des images/musiques qu'utilise le jeu
  - ***tiles***: images des tuiles
  - ***characters***: images des personnages
- ***src***: ce dossier comprends les données du jeu
  - ***database***: la base de données (SQLite) du jeu contenant les tuiles, les personnages, les cartes, les dialogues, etc…
  - ***maps***: le dossier contenant les cartes du jeu (dans chaque fichier est écrit une matrice d'identifiants correspondants aux identifiants des tuiles correspondantes dans la base de données.

Pour communiquer avec ce système de fichiers nous avons créé un *package* utilisé par *Game Editor* et *Game Engine* qui s'appelle *Architecture* et qui contient deux classes. La première permet de communiquer[1] avec la base de données (*Database Adapter*) tandis que la deuxième permet de communiquer avec les fichiers contenant les cartes.

[1] Communiquer = *écrire ou lire*

Game Editor

Principe

L'objectif du Game Editor est de permettre à un utilisateur n'ayant aucune compétence technique de créer lui-même son jeu vidéo. Dans l'éditeur que nous proposons, l'utilisateur peut créer ses propres cartes de jeu : il peut en renseigner les dimensions et y placer des tuiles qu'il aura préalablement créées et qui ont des caractéristiques spécifiques (mûr, portail). L'utilisateur peut ainsi créer plusieurs niveaux différents.

Il peut également créer ses propres personnages en important leur design, et en renseignant leurs caractéristiques comme leur nom, leur type (jouable, non jouable ou monstre) et leur affecter les attributs qu'il souhaite.

Enfin, l'utilisateur peut créer des monstres qui peuvent affecter les attributs des personnages via des compétences. Comme les attributs, ses compétences sont également paramétrées par l'utilisateur via divers coefficients.

Détail de l'architecture

Le Game Editor est composé d'un ***Main*** qui permet de lancer le programme principal. Ce programme utilise **EditorView** qui gère l'affichage des différentes sections d'édition (Cartes, Personnages, Monstres, Attributs et Compétences). Chaque section a son propre affichage qui se découpe en (au moins) 3 parties graphiques communes :

- **Assets View** : interface pour la partie latérale gauche de la section qui sert à la création d'un nouvel élément, et à l'affichage des éléments existants.

- **View Model** : interface pour la partie centrale de la fenêtre, qui permet l'affichage et l'édition de l'élément sélectionné (map, personnage).

- **Setting Model** : interface pour la partie latérale droite de la fenêtre, qui permet l'affichage et la sélection des paramètres/attributs de l'élément sélectionné.

Ces interfaces sont chacune réalisées par les différentes classes de chaque section.

Les classes implémentant **Assets View** (Asset Tiles View, Assets Character View, …) réalisent également deux interfaces **ActionListener** et **Validate Listener** (qui agit comme un écouteur d'événement) qui permettent respectivement de rendre actif le bouton d'ajout d'élément, et de mettre à jour l'affichage des éléments.

Les classes implémentant **View Model** (View Tile, ViewCharacter, …) réalisent aussi ces 2 mêmes interfaces pour rendre actif le bouton supprimer et pour la mise à jour du menu déroulant des éléments.

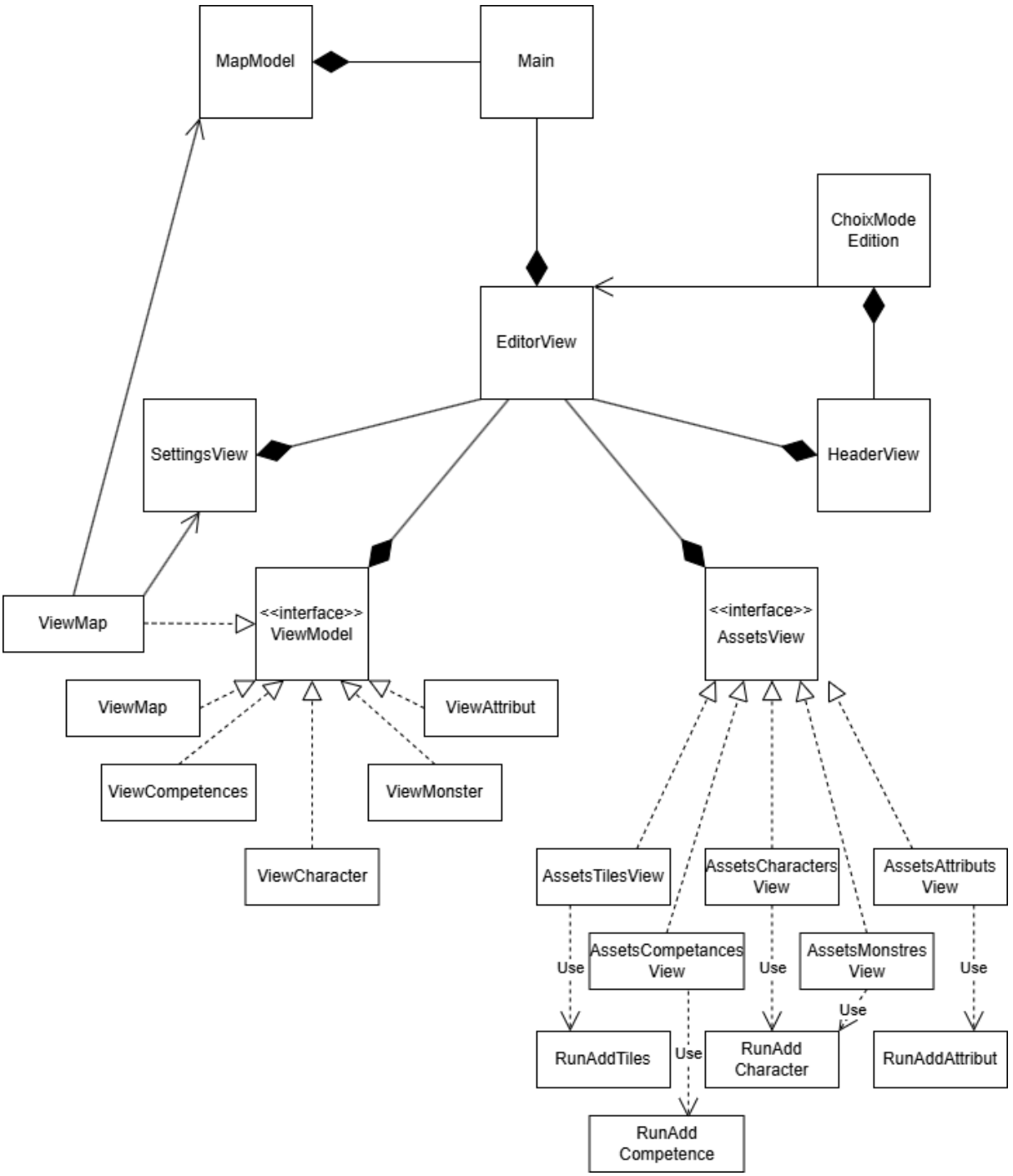


Fig. 3 - Diagramme UML du *Game Engine*

Développement des fonctionnalités et problématiques rencontrées

- **Communication des différentes parties graphiques de la fenêtre**

Objectif : pouvoir mettre à jour l'affichage des éléments dans le menu déroulant et dans la partie latérale gauche après ajout/suppression d'un élément dans la base de données.

Problématiques rencontrées :

1. comment une classe peut signaler à une autre qu'il faut actualiser son affichage ?
2. comment minimiser le nombre d'éléments à mettre à jour (ne pas recharger tout la page si un seul panneau a changé) ?

Résolution des problématiques:

1. Transmettre à la classe qui ajoute/supprime un élément un écouteur d'événement (classe qui implémente l'interface ValidateListener par exemple), qui appelle la méthode action() de cette classe après changement dans la base de données.
   (Exemple rencontré : Dans EditorView, on transmet à AssetCharacterView un écouteur d'événement ViewCharacter pour faire actualiser le menu déroulant de ce dernier après ajout d'un personnage).

2. La méthode action() des classes qui doivent actualiser l'affichage appelle une méthode update spécifique à l'élément qui change. La méthode update supprime tous les éléments du panneau concerné et recharge les éléments à partir de la base de données.

- **Enregistrement et lecture des données :**

Objectif : pouvoir enregistrer son travail à la fois pour le ré-éditer plus tard, mais également pour ensuite pouvoir le transmettre au Game Engine.

En effet, pour réaliser le stockage de nos données, nous utilisons à la fois une base de données SQL et pour tout ce qui est assets, et nous stockons la map des Id ainsi que les coordonnées des portails et des spawns dans un fichier lié au projet.

Game Engine

Les 2 nuages du schéma représentent des dossiers dont les détails auraient alourdi le schéma. ***Architecture*** contient les classes nécessaires à la lecture de la base de données.
Tile contient 2 classes, ***Tile Model*** qui permet de modéliser une tuile de la carte et Coordinate qui contient un système de coordonnées.

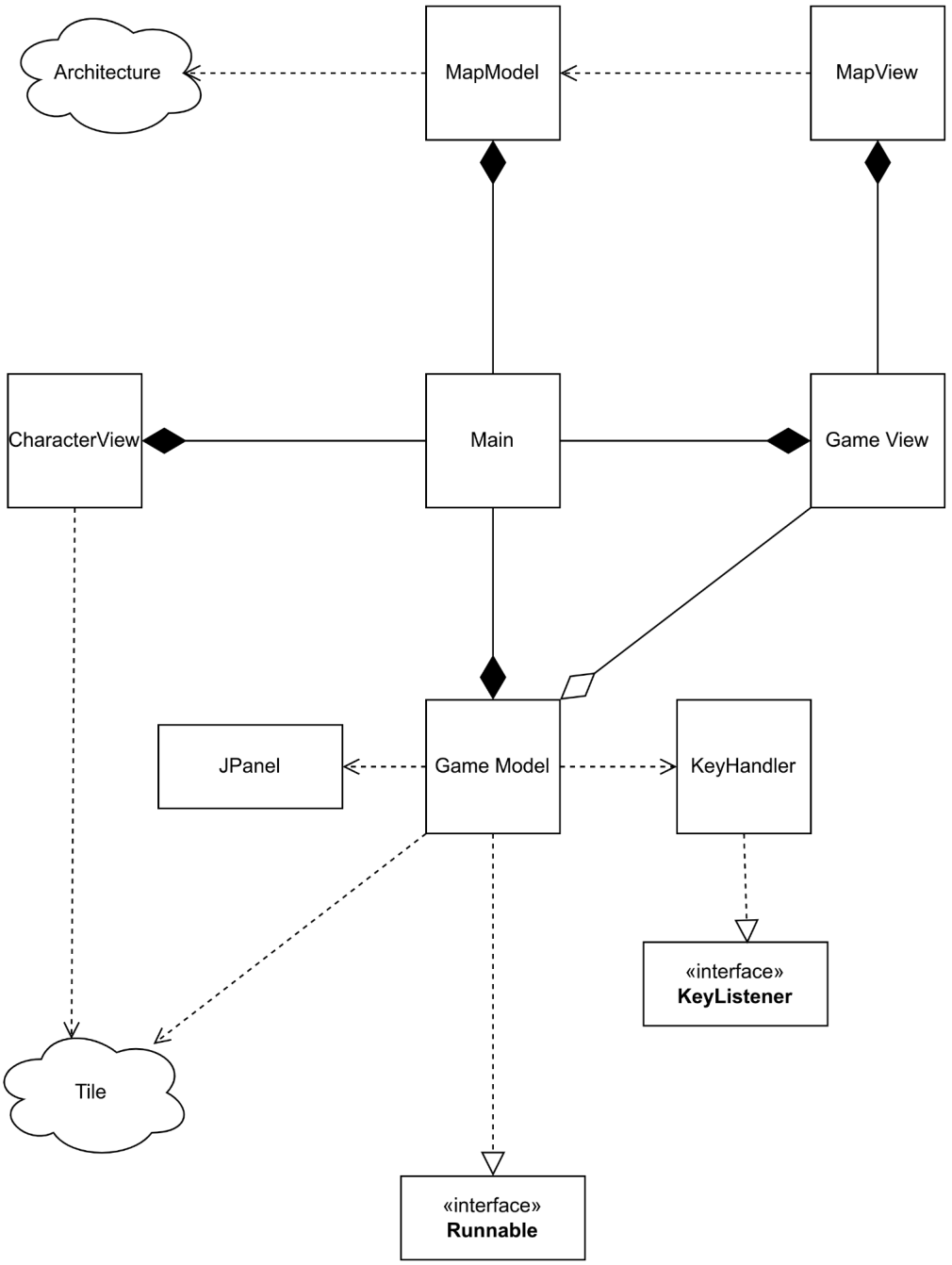


Fig. 4 - Diagramme UML du *Game Engine*

Principe de fonctionnement

L'objectif du Game Engine est de faire tourner le jeu vidéo en lui-même du point de vue joueur (la partie de création et d'édition du jeu se trouve dans le Game Editor) .

Détail de l'architecture

Le Game Engine est composé d'un ***Main*** qui permet de lancer le programme principal, ce programme utilise le ***Game Model*** qui comporte la boucle de jeu qui s'occupe de traiter les touches du clavier utilisées pour déplacer le personnage sur la carte.
Le ***Main*** fait aussi appel à ***GameView, CharacterView*** et ***MapModel.***

Pour fonctionner, le ***GameModel*** implémente et réalise de plusieurs autres classes et interfaces :

- ***KeyHandler*** pour reconnaître les touches du claviers pressées
- L'interface ***Runnable*** qui permet de créer la boucle active du jeu
- Le dossier ***Tile*** permet de changer les coordonnées du personnage sur la carte
- ***Jpanel*** permet de créer le panel du jeu

La classe ***CharacterView*** permet de créer la tuile du personnage et d'animer ses déplacements en changeant son apparence selon la direction du déplacement.

La classe ***GameView*** contient la partie graphique du jeu (fenêtre principale qui contient la carte et le personnage). Elle implémente donc la classe ***MapView*** qui gère la partie graphique de la carte.

La classe ***MapView*** implémente donc la classe ***MapModel*** qui contient la carte (sous forme d'un dictionnaire < clé : Coordonnées; valeur : Tuile >) et permet différentes opérations sur les tuiles de la carte.

Développement des fonctionnalités et problématiques rencontrées

➢ **Déplacements du personnage**

Objectif : pouvoir contrôler les déplacements du personnage sur la carte à l'aide des touches *ZQSD* du clavier.

Problématiques rencontrées :

3. comment reconnaître qu'une touche du clavier à été pressée.
4. comment faire se déplacer le personnage sur la carte

Résolution des problématiques :

1. Pour reconnaître les touches du clavier utilisées on utilise un key handler qui grâce à l'interface *KeyListener* repère les touches pressées.

2. Pour ce qui est du déplacement en lui-même du personnage, on utilise un système de coordonnées que l'on fait évoluer en fonction des touches pressées.

➢ **Animation des déplacements**

Objectif : animer les déplacements du personnage

Problématiques rencontrées :

1. Comment réaliser une animation
2. Comment implémenter les animations

Résolution des problématiques :

1. L'animation va consister en l'alternance de 2 images par type de déplacement (un type de déplacement étant un déplacement dans une direction NORD, SUD, EST, OUEST).
2. Lors des déplacements on va créer un compteur d'images permettant de savoir quelle icône du personnage on doit afficher pour créer l'animation.

➢ **Gestion des collisions**

Objectif : Certaines tuiles possèdent la caractéristique d'être un mur, on ne veut donc pas que le personnage puisse se déplacer dessus.

Problématiques rencontrées :

1. Comment tester que la case d'arrivée d'un déplacement est un mur pour ne pas réaliser le déplacement si c'est le cas.

Résolution des problématiques :

1. Mise en place du fonction qui simule le déplacement sans le réaliser et test si la case d'arrivée est un mur.

➢ **Gestion des portails**

Objectif : On veut pouvoir changer de carte grâce à des tuiles spécifiées réalisées par le créateur du jeu dans le Game Editor.

Problématiques rencontrées :

1. Obligation de relancer le Game Engine sur une map différente, générant ainsi une animation désagréable au joueur.

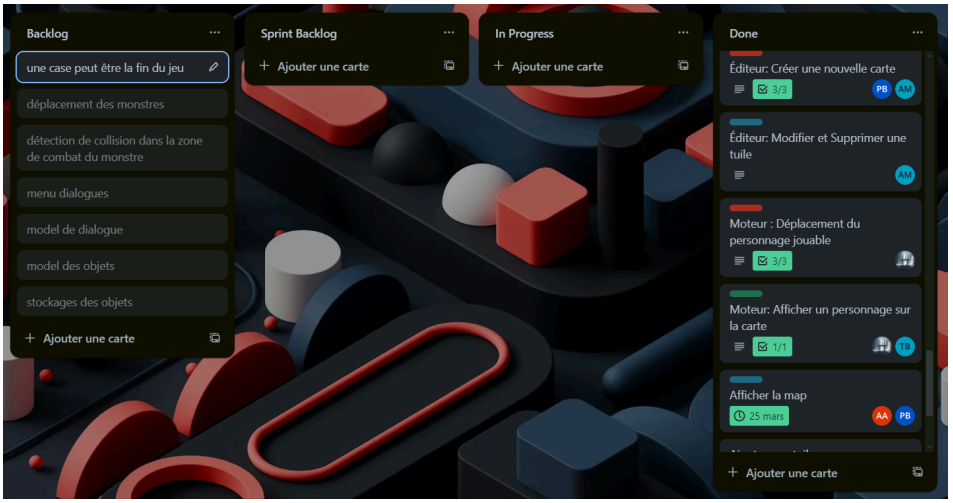
Résolution des problématiques :

1. Actualisation du main panel uniquement en ayant préalablement changer la carte chargée dans le programme.

Organisation (méthodes agiles)

Nous allons maintenant aborder la partie gestion de projet, mobilisant les principes des Méthodes Agiles. Il est clair que ce projet était un projet ambitieux, ainsi il était nécessaire d'avoir une bonne organisation dès le début du projet afin d'être le plus efficace possible.

Dès que nous avons choisi notre sujet de projet, nous avons pu définir les différentes tâches importantes, ce qui nous a permis de nous répartir le travail efficacement avant même que la première itération ne commence. Nous avons donc naturellement mis en place un groupe WhatsApp avec des sous groupes dédiés à des sujets bien précis (par exemple un sous groupe était consacré au développement du Game Editor tant qu'un autre était sur le Game Engine) pour pouvoir échanger facilement sur d'éventuels problèmes rencontré ou notifier notre avancement personnel sur nos tâches respectives. En parallèle, nous avons mis en place un système de répartition des tâches grâce à l'application Trello, celui-ci nous a permis de répertorier facilement les tâches terminées, celles à venir, ou celles étant en développement.



De plus, nous avons mis en place des sessions de travail qui consistaient à passer toute la journée dans un même appartement afin d'avoir des sessions en présentiel et ce qui permettait de progresser plus rapidement. Nous avons pu effectuer cela deux fois, lors des itérations 2 et 3.

C'est grâce aux deux outils qui nous permettaient de communiquer que nous avons pu résoudre un problème qui s'est rapidement développé : nous avons très peu réussis à organiser des sessions de travail où nous pouvions réellement travailler tous les sept en même temps, et celà était dû à nos disponibilités qui ne s'accordaient que rarement. Il peut paraître handicapant, sur un projet de groupe, de ne pas pouvoir travailler ensemble directement, mais les groupes de discussions et le Trello nous ont permis de passer outre ce problème. De plus, au moins une fois par semaine, nous avons fait des réunions en distanciels pour se tenir au courant des avancées et des projets à venir, et dans le cadre où certaines personnes ne pouvaient pas être présentes lors de la réunion, des récapitulatifs étaient envoyés sur le groupe de discussion.

Pour finir, il faut bien évidemment que nous parlions de notre Github qui nous a permis de regrouper les codes réalisés au fur et à mesure du projet. Nous avons décidé pour ne pas se perdre dans les commits que chacun aurait sa branche de développement ou il pourrait ajouter son code en

attente de validation, et une fois validé, le code était mis dans la branche principale afin que tout le monde puisse avoir les nouvelles fonctionnalités ajoutées.

Conclusion

Notre objectif était de créer un logiciel de création de jeux vidéo 2D accessible à des utilisateurs sans compétence en programmation. Pour ce faire, nous avons développé deux outils principaux : le Game Editor, qui permet la création et l'édition des jeux, et le Game Engine, qui fait tourner les jeux créés.

Le développement de ces outils nous a confronté à diverses problématiques techniques, notamment la communication entre différentes parties graphiques, l'enregistrement et la lecture des données, ainsi que la gestion des déplacements et des animations des personnages. Nous avons également dû concevoir un système de fichiers robuste pour stocker et organiser les données des jeux.

L'application finale permet la création de cartes, de personnages, et de monstres, ainsi que l'implémentation de dialogues et de portails pour le changement de cartes. Cependant, certaines fonctionnalités comme les combats, les compétences des personnages, et l'inventaire restent à développer.

Ce projet a été également une opportunité pour nous de travailler en équipe en utilisant les méthodes agiles. Les sprints, les réunions régulières et l'utilisation de Trello pour la gestion de projet nous ont permis de maintenir une bonne communication et d'assurer un suivi rigoureux de l'avancement du projet.

En conclusion, Ystoria est une base solide pour un logiciel de création de jeux vidéo 2D. Il reste encore des fonctionnalités à ajouter pour qu'il soit entièrement complet, mais les fondations sont en place. Nous sommes fiers du chemin parcouru et des compétences acquises tout au long de ce projet.