POLYTECH SORBONNE

Projet C++ : Moteur Jeu 2D et Implémentation mini-jeu

Noah Veytizoux - Paula Burbano

27 janvier 2023

1 Description de l'application

Initialement, notre but était de développer un jeu de type RPG, donc avec des ennemis à combattre, des objets à récupérer et des quêtes à compléter. Cependant, face à la complexité de développer un tel jeu, nous nous sommes plutôt orientés vers le développement d'un moteur de jeu 2D, efficace et facilement adaptable, qui contient toutes les classes et méthodes nécessaires à l'ajout d'entités et de fonctionnalités. Donc même si le mini-jeu implémenté en surface peut paraître très simple, la complexité et la propreté du code derrière donne toute sa valeur à notre projet.

Afin de montrer le potentiel de ce moteur, nous avons implémenté un mini-jeu de football à 2 joueurs, qui jouent sur le même ordinateur. Le but du mini-jeu est simple, faire rentrer le ballon dans la cage de l'adversaire. Malheuresement, la complexité de ce jeu ne permet pas de rendre compte de toutes les possibilités de ce moteur.

Ce mini-jeu se décompose en une surface de jeu, les cages et les murs, qui empêchent les entités de sortir de la surface de jeu. À chaque fois que la balle rentre dans les cages, elle est remise en jeu. Un système de collision permet de gérer les interactions joueurs/murs, joueurs/balle et balle/mur.

2 Fonctionnement du Moteur

Le programme se divise en deux parties : l'**ECS** ("*Entity Component System*") et l'ensemble des fichiers faisant appel à l'**ECS** pour donner forme au jeu. Ces fichiers sont ceux qui permettent de faire tourner le jeu, de créer la map et de gérer les collisions. On a également implémenté un fichier *TextureManager.hpp* qui simplifie l'utilisation de la *SDL2* ainsi qu'un fichier *Vector.hpp* qui nous fournit une classe de vecteurs 2D qui répond à nos besoins. Le diagramme UML donne une meilleure vision du programme : 4.2.

2.1 ECS

Nous allons détailler rapidement le fonctionnement de l'**ECS**. Ce système nous permet de créer des objets, et de leur attribuer différentes components, dont la principale est la position (ie. *TransformComponent*), puisqu'elle est utilisée par la plupart des autres components.

La gestion individuelle de chaque component de n'importe quelle entité permet de décider quelles caractéritiques possèdera tel ou tel entité. Par exemple, la zone de jeu dans le mini-jeu est délimitée via des murs (ie. *Colliders*) qui n'ont pas de visuel (ie. *SpriteComponent*) puisqu'on ne veut pas les voir s'afficher. Aussi, on a donné aux deux joueurs un accès au clavier via le component *Keyboard*. Un constructeur avec des arguments permet de donner des touches différentes à chaque joueur.

C'est aussi cette grande adaptabilité qui nous a permis de rajouter une nouvelle classe fille *MultiSpriteComponent* de *SpriteComponent*, qui nous permet de rajouter des animations aux objets, et notamment aux personnages dans notre cas.

La principale complexité de l'**ECS** se situe au niveau du fait que les fonctions ses fonctions de base (ie. *"hasComponent<>()", "getComponent<>()" ou "addComponent<>()"*) sont des **templates**, ce qui complexifie le code.

POLYTECH SORBONNE

2.2 Implémentation mini-jeu

Toutes la déclaration des entités et le choix de ce que l'on veut leur faire se fait dans le fichier *"Game.cpp"*. C'est notamment dans la fonction *"update()"* que l'on va décider de modifier les variables d'un component via des fonctions accesseurs.

Afin d'accéder facilement aux entités, on les divise en plusieurs groupes en fonction de leurs types (joueurs, murs, map, etc...). De cette façon on peut itérer sur certains types d'entités et vérifier des interactions entre un groupe et un autre. C'est notamment de cette façon que l'on vérifie les collisions.

3 Contraintes techniques

Quelques chiffres : notre programme répond aux exigences et se compose de **14 classes**, avec **3 niveaux de hiérarchie** (*Component* → *SpriteComponent* → *MultiSpriteComponent*). C'est d'ailleurs notre classe *Component* qui contient nos **fonctions virtuelles**. Il n'est pas nécessaire de repréciser que ces fonctions sont virtuelles dans les classes filles car elles sont automatiquement considérées comme tel. Nos **surchages d'opérateur** sont utilisées pour faciliter le calcul avec les vecteurs dans le fichier *Vector.hpp*. Enfin, nous nous servons des **conteneurs** *vector*, *array* et *bitset* dans le fichier *Entity.hpp*, qui est le principal fichier de l'**ECS**.

La complexité de notre moteur nous a permis d'exploiter pleinement les contraintes techniques, nous avons pu remplir ces conditions en se servant correctement de chaque feature du C++. Par exemple, les conteneurs comme *vector* et *bitset* sont parfaitement adaptés à notre besoin, puisqu'ils nous permettent de gérer efficacement les components de chaque entité.

Aussi, l'héritage de la classe *Component* nous permet de coder efficacement et de rajouter de nouvelles fonctionnalités. L'étape suivante, si nous étions restés sur notre idée de base, aurait été de rajouter des classes *ComportementComponent* et *InventoryComponent*, qui auraient géré respectivement, le comportement d'entités non joueur (ie. **PNJ**) et les objets transportés par des entités.

4 Mode d'emploi

L'application a été développée pour que deux joueurs puissent jouer de façon simultanée. Le mode d'utilisation du jeu est décrit ci-dessous.

4.1 Lancer le jeu

Toutes les instructions à suivre pour l'installation et l'utilisation de l'application sont présentes dans le *Readme.md*.

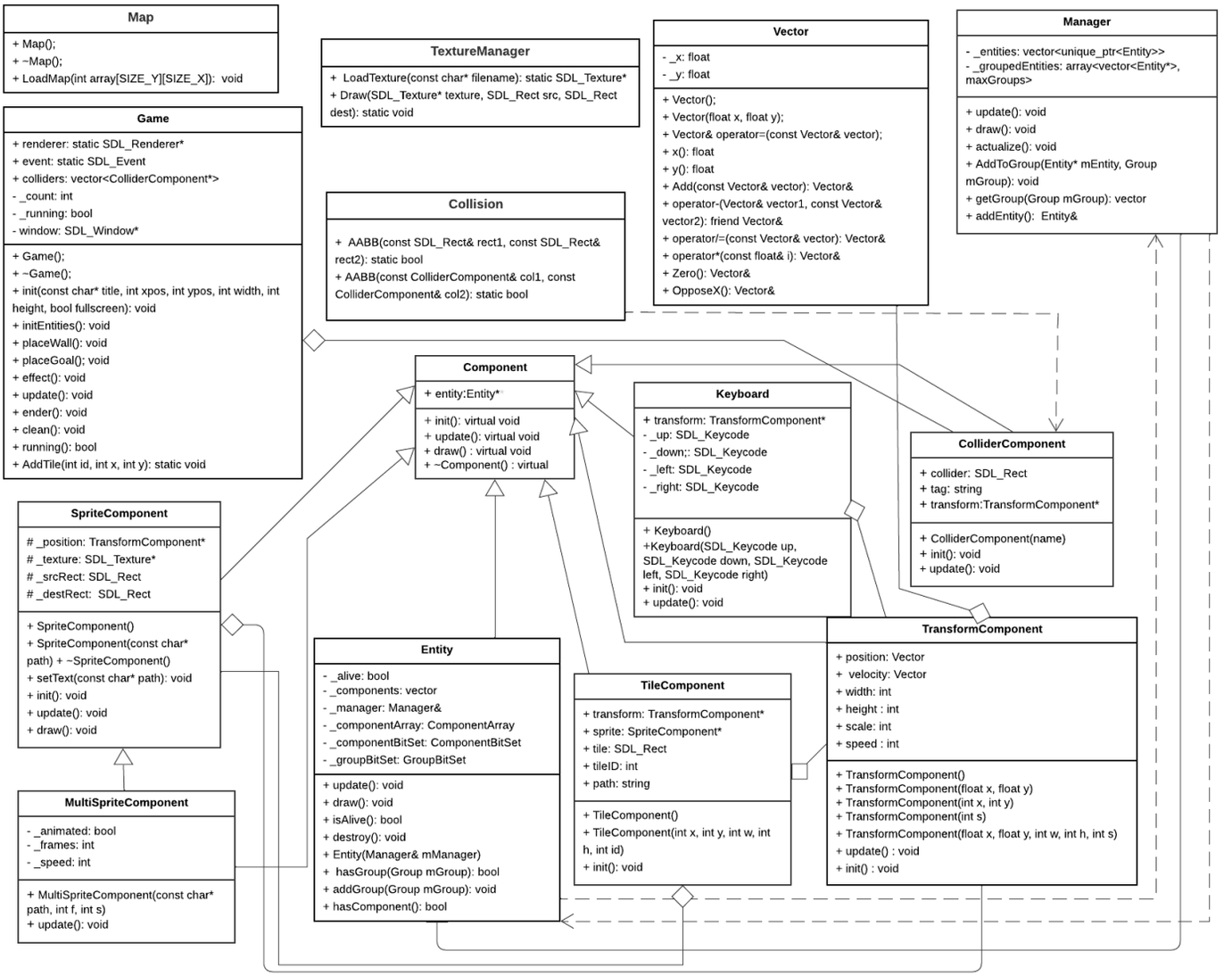
4.2 Commandes à utiliser

Chaque utilisateur possède les mêmes mouvements pour se déplacer en tant que joueur. Les commandes sont les suivantes sur un clavier **AZERTY** :

Commandes		
Se déplacer	Joueur 1	Joueur 2
Droite	'D'	'L'
Gauche	'Q'	'J'
Haut	'Z'	'I'
Bas	'S'	'K'

1 Table – Commandes pour le déplacement des joueurs

5 Diagramme UML



1 FIGURE – Diagramme UML de l'application

Remarques : Toutes les méthodes des classes implémentées ne sont pas présentes dans le diagramme pour que les différents éléments restent visibles. En effet, uniquement les méthodes les plus importantes et celles qui montrent bien les relations entre les différentes classes on été conservées.

6 Nos fiertés

D'une part, nous avons décidé d'utiliser un outil simple à utiliser et adaptable à la complexité de l'application : **CMake**. De cette manière, peu importe la taille du projet, les librairies utilisées, les dépendances entre les fichiers et les extensions nécessaires, nous implémentons une méthode propre et efficace de compilation et de production d'exécutables, qui est portable à plusieurs OS et configurations.

D'autre part, même si nous avons également appris à nous servir d'une librairie graphique, dans notre cas la **SDL2**, ce dont nous sommes le plus fier est la clarté et l'adaptabilité de notre moteur. Même si nous n'avons pas créé un jeu de type RPG, l'implémentation de notre moteur permettra de poursuivre ou commencer un nouveau projet à partir de ce que nous avons fait. Mais nous avons déjà bien décrit plus haut les avantages de l'**ECS**.