Païta Etienne Etudiant L2 informatique DANT

Licence Informatique

Complémentaire métier

Parcours Développeur d'Applications - Nouvelles Technologies

Rapport de projet

Création d'un clone du jeu Pokémon

Juillet – Août 2021

Table des matières

I. Introduction

Ce projet consiste en la réalisation d'un clone du jeu Pokémon. C'est un jeu en deux dimensions dans un monde imaginaire. Des personnages, les dresseurs, doivent attraper et se battre contre des créatures dotées de pouvoirs, les Pokémons. Le but est simple, attraper tous les Pokémons du jeu et faire la meilleure équipe de 6 Pokémons. Ces Pokémons vont servir aux joueurs, d'armes de combat, et sont pourvus de vies ainsi que d'expériences gagnées à chaque combat remporté. Chaque expérience permet aux Pokémons de monter de niveau et ainsi d'apprendre de nouvelles compétences d'attaques.

Dans ce rapport vous allez suivre ma progression dans la conception et la réalisation de mon projet, et vous pourrez voir les choix effectués ainsi que les différentes solutions apportées aux problèmes rencontrés.

II. Conception

1. Choix des outils

Je savais que le langage que j'utiliserai serait orienté objet. Je me suis alors longuement documenté pour choisir un langage performant pour le jeu que je voulais créer, mais également sur les différentes API [1]qui pourraient m'être utiles lors du développement. Mon choix s'est alors tourné vers le langage C++, très utilisé dans la conception de jeux et d'applications, ainsi que vers L'API SFML[2].

L'API SFML est une interface de programmation d'applications qui facilite grandement le développement de jeux ou d'applications multimédias. Elle est composée de cinq modules : système, fenêtrage, graphisme, audio et réseau. La documentation de ces modules est très riche et complète, ce qui facilite sa prise en main. De plus, la SFML est disponible dans un bon nombre de langages, dont le langage C++.

La SFML étant facilement configurable sur Visual Studio, j'ai décidé de créer ce projet sur cet environnement de développement.

2. Récupération des données essentielles pour le jeu

Pour concevoir ce projet, il me fallait d'abord récupérer un maximum de données nécessaires au développement de mon futur jeu. Une fois les données récupérées, je devais maintenant établir les classes que j'allais créer ainsi que les liens et les interactions qu'il y aura entre elles. De ce fait, la réalisation d'un diagramme de classes était donc primordiale pour bien commencer mon projet sans avoir besoin de passer des heures pour implémenter chaque nouvelle étape.

Je suis alors parti sur une base simple. Une classe, nommée Game, gérera tout, les rendus visuels, les combats et les déplacements en appelant ou non certaines classes. (Annexe figure 1).

Je vais donc devoir créer plusieurs classes dont je vais définir l'utilité.

Nous avons la classe Object qui est une classe abstraite grâce à sa méthode virtuelle pure, estTraversant. Cette méthode permet de savoir si un objet peut être traversé par un joueur. Ces objets vont donc composer notre monde virtuel. Pour savoir de quel type d'objet il s'agit, cette classe contient un attribut, un type de ma classe Elements.

Venons-en à cette classe Elements, à quoi sert-elle ? J'ai longuement réfléchi sur, comment faire pour pouvoir importer une carte de jeu à partir d'un fichier texte. Je suis arrivé à une solution simple mais certainement pas la plus efficace, chaque objet de mon monde aura un numéro désigné. Elements est donc une enum class qui permet de créer plusieurs types avec un numéro associé. Ces numéros vont être utiles lors du rendu visuel du jeu.

Nous avons alors une classe pour chaque objet sur la carte. Pour le moment je n'ai créé que 3 objets essentiels, Wall, Grass et TallGrass qui héritent de la classe Object. J'ai également décidé qu'un

[1] API : Application Programming Interface
[2] SFML : Simple and Fast Multimedia Library

Dresseur serait considéré comme un objet afin de faciliter le rendu visuel. Dresseur hérite donc aussi de la classe Object. J'ai également créé une classe Player qui désignera l'utilisateur qui joue au jeu. Cette classe hérite de Dresseur et est, par conséquent, un Object.

Tous ces objets sont stockés dans un tableau dynamique à deux dimensions de la classe Map. A l'aide d'un fichier composé de chiffre et/ou de lettres, nous allons pouvoir créer notre monde. Par exemple, un 0 dans le fichier, correspond à l'Object Grass comme on peut le voir dans la classe Elements. La classe Map va donc, à partir d'un fichier, créer un monde que l'on pourra ensuite afficher.

J'ai créé une classe Textures qui pourra importer des images et modifier les différents paramètres d'affichage. Ensuite, une classe Render viendra afficher tout le contenu d'une Map à l'écran, à l'aide de la classe Texture.

Maintenant, passons à la classe Pokémon. Cette classe permet de créer des Pokémons qui possèdent toute sorte d'attributs et aussi des attaques.

Les attaques sont des instances de la classe Attaque. Cette classe permet de créer une attaque avec un nom et un nombre de dégâts.

Les Pokémons sont visibles et utilisables uniquement à travers les dresseurs et les combats. Une classe Combat est donc utile pour gérer le rendu visuel d'un combat de Pokémons ainsi que les actions et les modifications que cela engendre sur les pokémons.

Bien évidemment la classe Game est appelée au travers de mon *main*.

Une fois les données récupérées et les outils à utiliser choisis, je peux me mettre au développement de mon jeu.

3. Développement

a. Création d'une fenêtre de jeu

Dans un premier temps, je devais créer une fenêtre de jeu. La SFML le permet facilement mais elle parait "gelée", incapable d'être déplacée, redimensionnée, ou bien encore fermée. Il fallait donc au moins pouvoir fermer la fenêtre. Pour cela, la SFML nous met à disposition la possibilité de gérer les différents évènements clavier et souris de l'utilisateur et permet de récupérer l'instruction de fermeture de l'écran.

J'ai ensuite créé une classe Game qui servira de point de contrôle pour toutes les actions et affichages du jeu.

La classe Game créée une fenêtre, grâce à l'outil SFML, dans laquelle nous pouvons dessiner des formes et des images pour ensuite les afficher à l'écran.

b. Déplacements d'un joueur

Une fois la fenêtre utilisable, je me suis mis à la création d'un " joueur " et de ses déplacements (voir annexe figure 2). Pour commencer je me suis concentré sur le déplacement d'un potentiel joueur. Je n'ai utilisé qu'un simple cercle que je déplaçais en fonction des touches Z, Q, S, D, et je l'empêchais de dépasser les bords de mon écran. Une fois le bon fonctionnement des différents mouvements réalisé, j'ai créé la classe Player, contenant les coordonnées du joueur, et j'y ai raccordé les déplacements.

c. Création d'une Map

Il fallait ensuite pouvoir afficher un monde dans lequel je pourrais me déplacer. J'ai alors mis une image de fond représentant une carte connue du jeu Pokémon mais un problème vint s'ajouter. Comment vais-je pouvoir gérer les collisions avec certaines textures de mon monde ? Il aurait fallu pour cela ajouter toutes les coordonnées où le joueur ne peut pas se trouver… un peu compliqué et pas très robuste niveau code. J'ai donc réfléchi quelques jours et c'est alors qu'une solution m'est venue. Cette solution était de créer de A à Z une map avec des instances d'objets que je créerai. Cette idée rajoute beaucoup de codes au jeu et sûrement de nouveaux défis que j'étais impatient de rencontrer.

Afin de ne pas me perdre dans toutes mes idées, j'ai effectué un diagramme de classe (voir annexe figure 1). La meilleure construction que j'ai pu avoir, était de créer une classe abstraite Object qui serait la base de tout objet de mon monde. Je devais être capable de savoir si un objet peut être traversé ou non pour un joueur. Il me suffisait alors de créer une méthode virtuelle pure qui me renverrait true ou false en fonction de l'objet voulu. La seule chose qui ne me convenait pas dans mon diagramme, était le fait de devoir créer autant de classes qu'il y a d'objets potentiels. Malheureusement je n'ai trouvé aucune autre solution et j'ai donc continué sur cette piste. Je pouvais désormais créer ma classe Map dans laquelle je retrouverais un tableau dynamique d'Object à deux dimensions. Un autre souci m'a tracassé, comment savoir de quel objet il s'agit dans mon tableau ? Je devais pouvoir, à partir de la classe Object, connaitre la nature de l'objet que je veux utiliser. J'ai donc implémenté une classe supplémentaire, l'enum class Elements, qui énumère les différents éléments possibles de la map. Encore une fois, pour chaque nouvel objet que je voudrais ajouter à mon jeu, je devrais également l'ajouter à la classe Elements.

J'ai ensuite décidé que le Player serait considéré comme un Objet, pour cela, Player hérite d'une nouvelle classe, Dresseur, qui elle-même hérite d'Object. Ainsi, si j'implémente des Dresseurs contre lesquels le joueur pourra se battre ou interagir, il ne pourra pas passer au travers. Un autre avantage d'être un objet, c'est de simplifier le rendu visuel.

Je peux enfin essayer de créer ma propre carte ! Pour cela la classe Map construira une carte de jeu à partir d'un fichier texte placé en paramètre (voir annexe figure 3). Ce fichier texte sera composé de chiffres, ceux correspondant à chaque élément de la classe Elements ainsi que d'un 'P', lieu d'apparition du Joueur.

d. Création d'un tableau de MAP dans GAME

J'ajoute un tableau dynamique de pointeurs sur Map à ma classe Game afin de potentiellement pouvoir générer plusieurs Map. Pour se faire, la classe Game initialisera son tableau de Map à partir d'un fichier contenant le chemin vers les différentes Map utiles. Pour le moment ce fichier (AllMaps) contient un seul nom de fichier.

e. Affichage de la MAP

Maintenant, nouvelle étape importante consiste à afficher ma Map !

Ce fût une vraie réflexion de savoir comment gérer l'affichage de ce monde. Il me fallait avoir facilement accès aux différentes textures à afficher. Premièrement, la SFML permet d'importer des images pour en faire des textures affichables. J'ai alors créé une classe Textures qui pourra ainsi importer des images et modifier les différents paramètres d'affichage. Ensuite, une classe Render initialise toutes les textures que l'on devra utiliser et les stocks dans un tableau dynamique. Ce stockage doit se faire dans le même ordre que les attributs de Elements afin de faciliter l'accès aux textures grâce aux numéros associés aux types Elements.

Un attribut render de type Render est ajouté à la classe Game. Il sera initialisé et initialisera les différentes textures utilisées.

Chaque texture est créée une fois et une fois seulement. Grâce aux fonctionnalités de la SFML, je n'ai pas besoin de créer de nouvelles textures pour chaque case de ma map. Je peux utiliser une texture, la dessiner, et changer ses coordonnées. Je peux ainsi, à l'aide d'une seule texture, afficher le nombre d'objets que je veux correspondant à cette Texture. Pour faire simple, si j'ai un mur en coordonnées (0,0) mais aussi aux coordonnées (5,2), je fixe les coordonnées de la texture de mon mur en (0,0), je l'affiche, puis change ses coordonnées en (5,2) et affiche à nouveau ! Sur une grande Map, avec des objets complexes, cette méthode peut s'avérer très efficace.

La classe Render est dotée d'une méthode de rendu qui va d'abord faire le rendu de la Map avec la méthode vue ci-dessus, et ensuite le rendu du Player via ses coordonnées. Le joueur peut alors se déplacer sur la Map, sans avoir besoin de changer le moindre élément de la Map ! Mais avant ça, je dois légèrement modifier le code, pour le déplacement de mon joueur, de la classe Game en vérifiant, pour chaque mouvement, si l'Object d'arrivée est traversant. S'il ne l'est pas, le joueur ne se déplace pas.

f. Centrer la caméra sur le joueur

C'est bon ! Ma Map s'affiche et mon personnage peut se déplacer. Mais dans tous les jeux Pokémons, le personnage est au centre et c'est l'écran qui se 'déplace' sur la Map. J'ai alors créé de nouveaux attributs dans la classe Game, centeredMap et nbCase. La variable nbCase désigne le nombre de cases horizontales et verticales de mon écran. CenteredMap sera exactement de cette dimension avec comme point de repère central, le player. Après chaque mouvement, la fonction

updateCenteredMap (annexe figure 4) récupère les coordonnées du player et commence à remplir centeredMap en récupérant les objets de la map courante. Cette fonction s'assure que les coordonnées envoyées à la Map courante ne soient pas OutOfRange, et si c'est le cas, j'ajoute un Wall à centeredMap à la place. Cela permet de pouvoir se déplacer jusqu'au bord de la map, tout en restant au centre de l'écran, et en affichant quelque chose d'existant.

g. Combat Pokémon

Je devais maintenant pouvoir trouver un Pokémon, l'attraper et me battre contre lui.

Avant tout il me fallait créer la classe Pokémon.

Un Pokémon est créé de la même façon que les Object. Ils sont initialisés une fois dans Game et on utilise leur texture pour chaque même Pokémon. La classe Dresseur a été modifiée pour accueillir une équipe de 6 Pokémons et des méthodes relatives à l'ajout et la recherche de ces derniers ont été ajoutées.

Dans la classe Game, une autre modification est apportée. La découverte d'un Pokémon se fait via l'Object TallGrass. Lorsqu'un joueur se trouve sur cet objet de la Map, il y a une probabilité d'en trouver un.

Lorsque la probabilité est correcte, un combat se lance. La classe Combat est appelée au travers d'une méthode de la classe Game.

Combat gère le rendu visuel d'un combat ainsi que toutes modifications sur les Pokémons combattants. Pour avoir le coté visuel du tour par tour, j'ai dû ajouter au Player une variable booléenne turn. Cette variable indique par true ou false si c'est le tour du joueur. Si c'est le cas, le joueur peut choisir son action, le résultat de cette action est affiché, et ensuite le Pokémon adversaire fait une action aléatoire qui est ensuite affichée à son tour.

Les Pokémons sont dotés d'un tableau de quatre attaques. Ces attaques sont créées par la classe Attaque qui est initialisée avec son nom et ses dégâts.

Lors du combat, le choix d'une attaque se fait avec les mêmes touches de déplacement. Le programme sait sur quelle attaque il se trouve en fonction de la couleur de fond des cases. La case 0 correspond alors à l'attaque 0 et ainsi de suite. Il suffit ensuite d'appuyer sur la barre d'espace pour valider son choix.

Le joueur peut également fuir et attraper un Pokémon, mais a une chance que ces actions échouent. Si c'est le cas, son tour est passé, et c'est à l'adversaire d'attaquer.

Il a fallu cloner chaque Pokémon que le joueur attrapait car les données rentraient en conflit entre elles. Si le joueur combattait contre le même Pokémon en sa possession, chaque dégât subi par un, était aussi subi par l'autre. Désormais les Pokémons du joueur ne sont plus liés aux Pokémons sauvages.

Si l'adversaire n'a plus de vie, le Pokémon du joueur actuellement en combat, gagne des points d'expériences. Il peut alors monter de niveau et gagner de la vie ainsi que de la puissance d'attaque.

Lorsqu'un Pokémon du joueur n'a plus de vie, il est remplacé par le prochain Pokémon en vie de l'équipe (s'il existe). Le combat se termine si l'adversaire n'a plus de vie, ou s'il n'y plus aucun Pokémon en vie dans l'équipe du joueur.

De plus, un booléen 'utilisé' a dû être ajouté à la classe TallGrass car à la fin d'un combat, le joueur étant sur une zone d'apparition de Pokémon, il était presque impossible de bouger sans entrer dans un Combat. Cet attribut permet alors de faire une seule fois le test de probabilité, et qu'il soit positif ou négatif, ne pourra être retesté qu'une fois sorti de l'objet.

h. Musiques de jeu associées à certaines classes

La SFML permet également d'ajouter des fichiers audios dans un projet. La classe Game ainsi que la classe combat sont donc dotées d'un attribut audio que le programme lance en boucle lorsque c'est nécessaire.

III. Apports du projet

Ce projet m'a été extrêmement bénéfique. Il m'a permis d'apprendre un nouveau langage, le C++, que je voulais pratiquer depuis longtemps. J'ai également pu me familiariser avec l'API SFML qui facilite et rend plus accessible la création de jeux et d'applications multimédia.

Ce projet m'a appris la rigueur et la difficulté de créer un projet seul en partant de rien. J'ai dû passer beaucoup de temps à me documenter sur les différentes API existante, faire un choix en fonction des performances que je recherchais, mais aussi créer des UML et diagrammes pour structurer mon projet et le rendre plus abordable. Toutes ces connaissances que j'ai acquises durant ce projet me donnent envie d'apprendre davantage et surtout d'apprendre aux côtés de professionnels.

IV. Conclusion

Ce projet est l'aboutissement de deux mois de travail, sur le développement d'un jeu basé sur l'univers Pokémon. Nous pouvons nous déplacer dans un monde en deux dimensions, généré par une carte importée à partir d'un fichier. Le combat et l'entrainement des Pokémons ainsi que des actions tels que, fuir le combat, ou bien attraper un Pokémon pour l'ajouter à son équipe, sont possibles.

Ce projet m'a permis d'apprendre un nouveau langage et de me familiariser avec l'utilisation d'une API, mais également à m'ouvrir sur d'autres aspects de la programmation en élaborant des diagrammes et UML.

V. Annexes

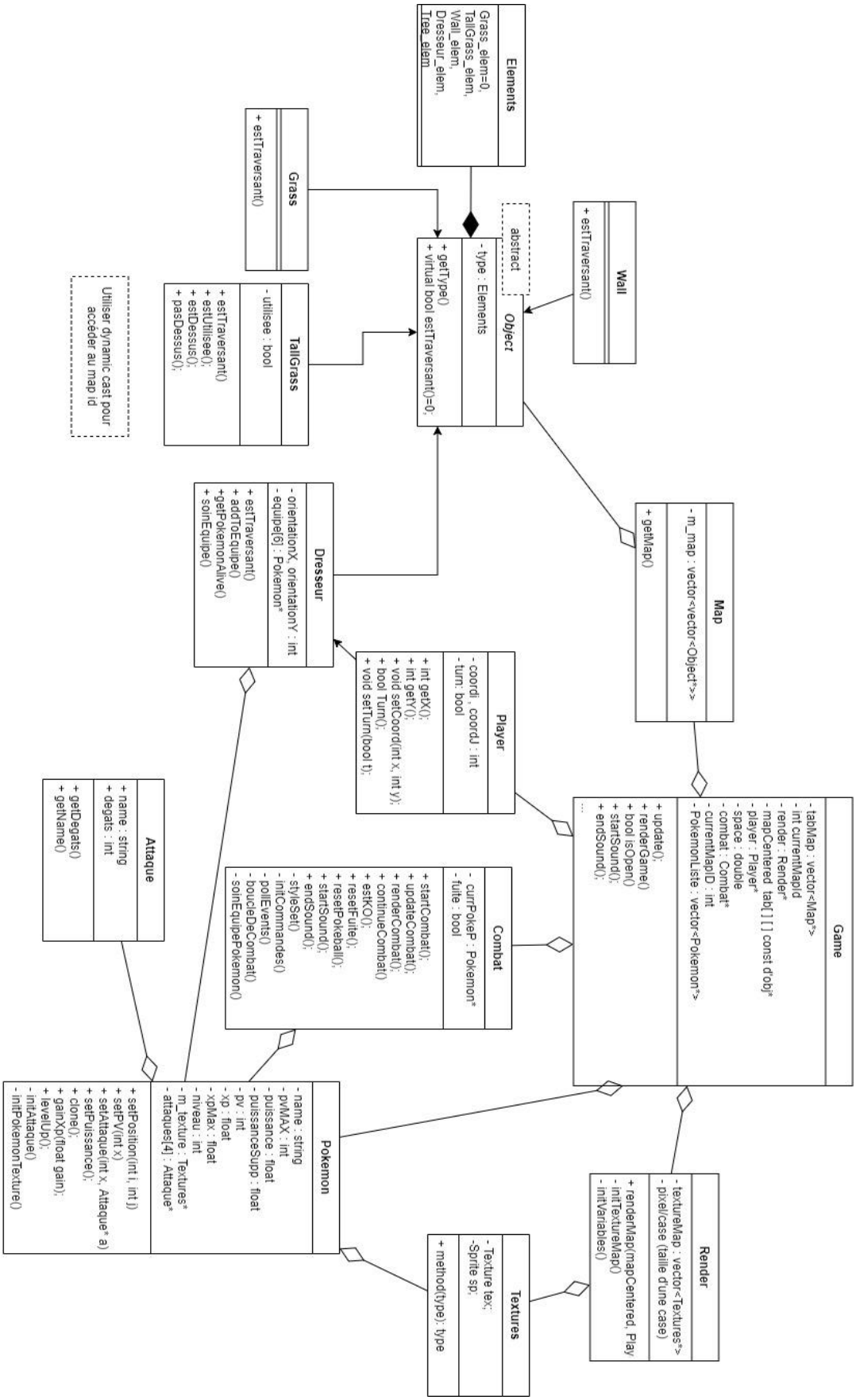
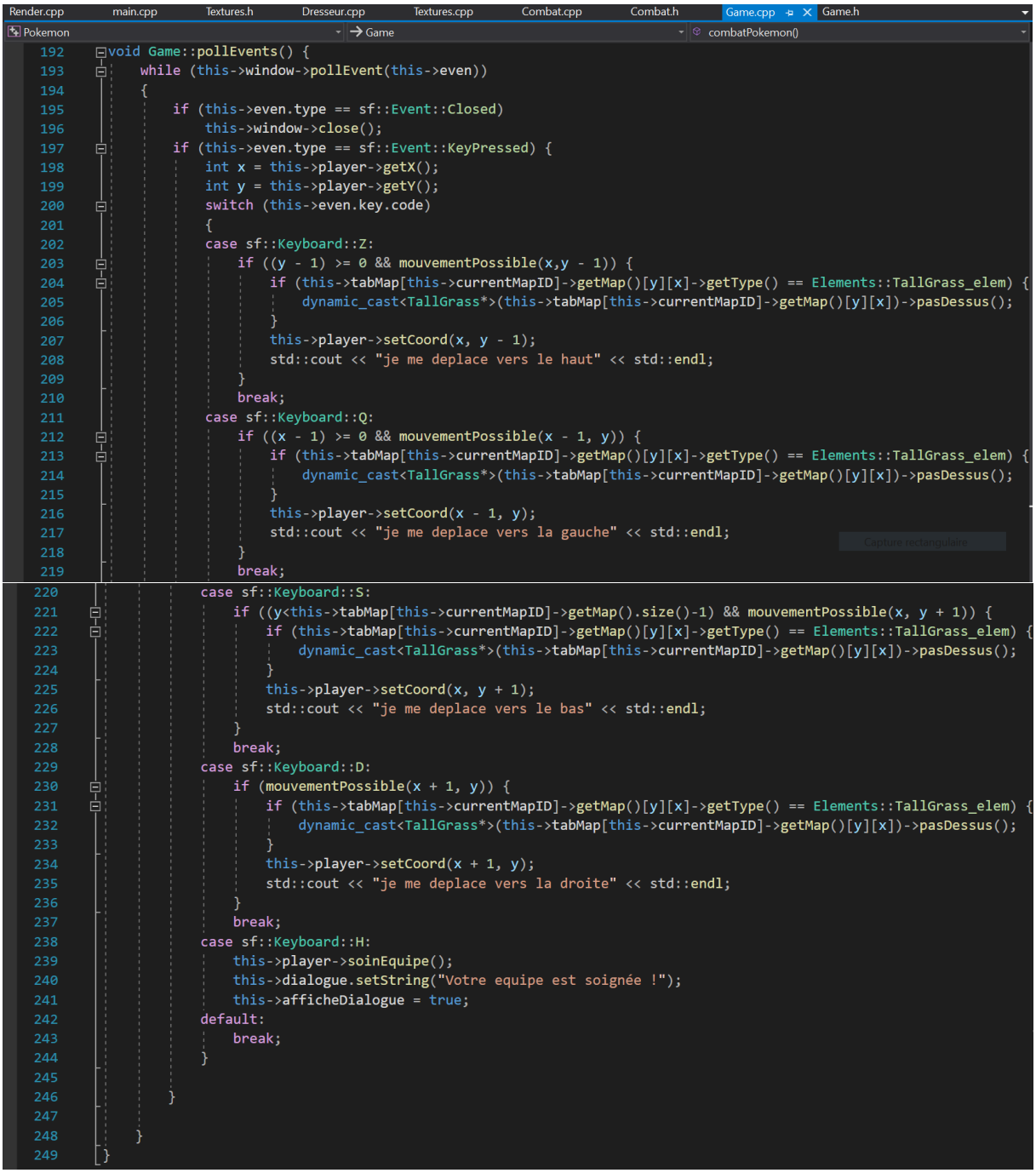


Figure 1: Diagramme UML du projet



```cpp
void Game::pollEvents() {
    while (this->window->pollEvent(this->even))
    {
        if (this->even.type == sf::Event::Closed)
            this->window->close();
        if (this->even.type == sf::Event::KeyPressed) {
            int x = this->player->getX();
            int y = this->player->getY();
            switch (this->even.key.code)
            {
            case sf::Keyboard::Z:
                if ((y - 1) >= 0 && mouvementPossible(x,y - 1)) {
                    if (this->tabMap[this->currentMapID]->getMap()[y][x]->getType() == Elements::TallGrass_elem) {
                        dynamic_cast<TallGrass*>(this->tabMap[this->currentMapID]->getMap()[y][x])->pasDessus();
                    }
                    this->player->setCoord(x, y - 1);
                    std::cout << "je me deplace vers le haut" << std::endl;
                }
                break;
            case sf::Keyboard::Q:
                if ((x - 1) >= 0 && mouvementPossible(x - 1, y)) {
                    if (this->tabMap[this->currentMapID]->getMap()[y][x]->getType() == Elements::TallGrass_elem) {
                        dynamic_cast<TallGrass*>(this->tabMap[this->currentMapID]->getMap()[y][x])->pasDessus();
                    }
                    this->player->setCoord(x - 1, y);
                    std::cout << "je me deplace vers la gauche" << std::endl;
                }
                break;
            case sf::Keyboard::S:
                if ((y<this->tabMap[this->currentMapID]->getMap().size()-1) && mouvementPossible(x, y + 1)) {
                    if (this->tabMap[this->currentMapID]->getMap()[y][x]->getType() == Elements::TallGrass_elem) {
                        dynamic_cast<TallGrass*>(this->tabMap[this->currentMapID]->getMap()[y][x])->pasDessus();
                    }
                    this->player->setCoord(x, y + 1);
                    std::cout << "je me deplace vers le bas" << std::endl;
                }
                break;
            case sf::Keyboard::D:
                if (mouvementPossible(x + 1, y)) {
                    if (this->tabMap[this->currentMapID]->getMap()[y][x]->getType() == Elements::TallGrass_elem) {
                        dynamic_cast<TallGrass*>(this->tabMap[this->currentMapID]->getMap()[y][x])->pasDessus();
                    }
                    this->player->setCoord(x + 1, y);
                    std::cout << "je me deplace vers la droite" << std::endl;
                }
                break;
            case sf::Keyboard::H:
                this->player->soinEquipe();
                this->dialogue.setString("Votre equipe est soignée !");
                this->afficheDialogue = true;
            default:
                break;
            }

        }

    }
}
```

Figure 2 : Méthode de gestion des entrées clavier à l'aide de l'outil SFML

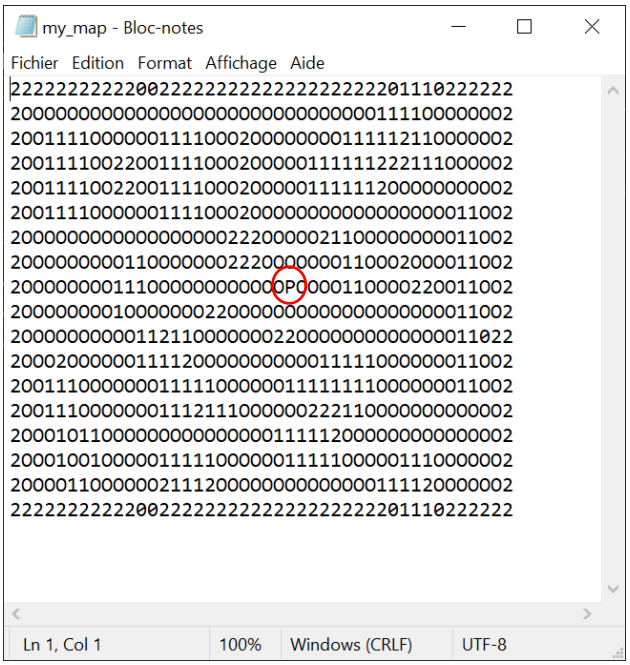


Figure 3 : Exemple de fichier contenant une carte de jeu

```
251  void Game::updateCenteredMap() {
252      //Pareil que initCenteredMap() sauf qu'on creer et remplace centeredMap
253      std::vector<std::vector<Object*>> c;
254      int xPlayer = this->player->getX();
255      int yPlayer = this->player->getY();
256      //std::cout << "player (" << xPlayer << "," << yPlayer << ")" << std::endl;
257
258      int i = yPlayer - (this->nbCase / 2);
259      while ((i < yPlayer + (this->nbCase / 2))) {
260          std::vector<Object*> tmp;
261
262          int j = xPlayer - (this->nbCase / 2);
263          while ((j < xPlayer + (this->nbCase / 2))) {
264              if (i < 0 || j < 0) {
265                  tmp.push_back(new Wall());
266              }
267              else if ((i >= this->tabMap[this->currentMapID]->getMap().size()) || (j >= this->tabMap[this->currentMapID]->getMap()[i].size(
268                  tmp.push_back(new Wall());
269              }
270              else {
271                  tmp.push_back(this->tabMap[this->currentMapID]->getMap()[i][j]);
272              }
273
274              j++;
275          }
276          c.push_back(tmp);
277          i++;
278      }
279      this->centeredMap = c;
280  }
281
```

Figure 4 : Méthode mettant à jour la variable centeredMap