Projet Logiciel Transversal

Tactical Wars

Moussa DIALLO , Maxime FARNOUD, Matthis HADDOUCHE & Timothé MUSETE LEKAN

Sommaire

1 . Objectif

1.1 . Présentation générale

Archétype : Advance Wars/Wargroove

- Jeu de stratégie militaire au tour par tour
- Gestion de ressources

Figure 1: Image du jeu « Advanced Wars »

Figure 2: Image du jeu « Wargroove »

1.2 . Règles du jeu et Ressources

Règles :

- Condition de Victoire : Prendre le QG ennemi
- Déroulement d'une Partie :
  - Le royaume adverse a des vues sur un territoire stratégique, défendez-le !
  - Chaque joueur débute dans un coin de la map, avec seulement son Quartier Général (QG) et sans aucune unité. Les deux joueurs disposent également d'un peu de mana (ressource fondamentale du jeu) pour commencer la partie.
  - Le QG sert à produire les différentes unités. Chaque unité possède un nombre prédéfinis de membres qui composent l'unité.
  - Une fois une unité produite au QG le joueur peut la déplacer selon la statistique de mouvement de cette unité.
  - Au début de la partie, la carte est couverte d'un brouillard de guerre qui disparaît en fonction de la position de chaque unité et de leur statistique de vision (chaque joueur a donc sa propre « vue » de la carte qui dépend de ses propres unités). Le brouillard cache complètement une case et ce qu'il y a dessus (terrain, unité, bâtiment) mais une fois disparu il ne revient pas.
  - Le joueur commence uniquement avec son QG comme bâtiment, et pourra en capturer d'autres durant la partie en y plaçant des unités. Lorsqu'une unité accomplit l'action de capture, elle ne peut pas attaquer. Un bâtiment se capture en 2 tours : 1er tour 50% de capture, 2ème tour 100% capture)
  - L'objectif est de prendre le QG adverse même si toutes les unités ennemies ne sont pas tombées.
- Il existe différents types de bâtiments :
  - Les mines de mana : permettent une récolte de mana abondante, ressource essentiel dans la partie qui permet de générer des unités.
  - Les villages : régénèrent la santé des unités se trouvant dessus et servent à l'effort de guerre en payant un léger impôt en mana à chaque tour.
  - Les camps d'entraînement : grâce à leurs prouesses, les officiers formateurs peuvent former des soldats performants ; ils permettent de générer des unités autre part qu'au QG et permettent donc de gagner du temps.

- Il y aura également différents types de terrains qui attribueront des malus de déplacements aux unités se trouvant dessus : la plaine est un environnement neutre et n'attribue aucun malus aux unités, la forêt est un environnement légèrement hostile et attribue un simple malus de déplacement aux unités (une unité commençant son tour sur une forêt sera ralentie et ne pourra pas aller aussi loin qu'elle ne le pourrait nirmalement), la montagne est un

environnement très hostile qui attribue un malus de déplacement très sévère aux unités (une unité ne peut pas aller sur une case où se trouve une montagne peut importe les circonstances).

Voici ci-dessous un exemple de tuiles qui seront utilisées pour représenter les différents bâtiments et terrains sur la grille de jeu :

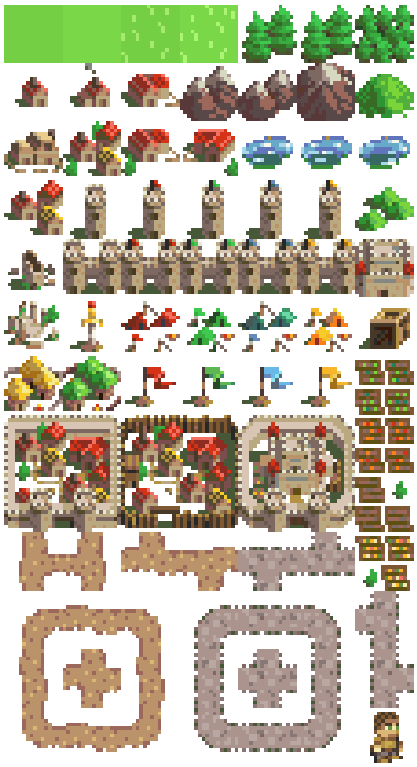

Figure 3: Tuile d'image pour les décors (terrains et bâtiments)

- Les différents types unités :
  - Les gobelins (unités de base): unité de corps à corps, peu résistante mais assez mobile, il s’agit d’une unité de base polyvalente.

Figure 4: Gobelin

- Les chauve souris (unités d’éclaireur): très bonne vision mais très fragiles. Leur mobilité leur permet de s’échapper rapidement et de frapper sans être vu.

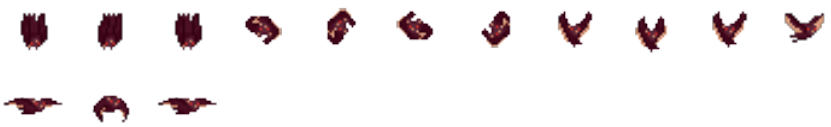

Figure 5: Chauve-souris

- Les magiciens : unité puissante à distance et moyenne au corps à corps. Assez fragile.

Figure 6: Magicien

- Les nains : de féroce combattant au corps à corps qui n’hésite pas à baisser leur garde pour attaquer l’adversaire.

Figure 7: Nain

- Les chevaliers : de véritables murs défensifs, ils sont prudents et se protègent derrière

leurs boucliers et leur armure, cela les rangs résistant mais diminue leur attaque et leur déplacement.

Figure 8: Chevalier

- Les golems, peu mobiles mais aussi puissants à distance qu'au corps à corps. Leur faible mobilité en font des cibles de choix pour les magiciens malgré leur formidable défense.

Figure 9: Golem

Le tableau suivant présente les valeurs des différentes statistiques des unités. Ces valeurs ne sont que de des valeurs de référence selon une échelle arbitraire et servent essentiellement à établir les proportions entre les différentes unités. Évidemment, ces valeurs sont vouées à être modifiées à l'avenir après les premiers tests dans l'idée de mieux équilibrer le jeu :

Characters	PV	Portée d'Attaque	Dégats	Portéée de déplacement	Coût en mana
Magicien	30	100	70	60	70
Goblin	25	20	25	60	10
Golem	100	60	100	20	100
Chauve-souris	10	40	10	100	20
Nain	50	20	50	40	50
Chevalier	70	20	30	40	50

2 . Description et conception des états

2.1 . Description des états

Un état est essentiellement composé d'une carte en 2D quadrillée. Sur cette carte se trouvent des éléments fixes (les terrains et les bâtiments) et des éléments mobiles (les unités). Tous ces éléments ont une propriété commune qui est leurs coordonnées (x,y) qui correspond à leur position sur le quadrillage.

En plus de cela, un état doit prendre en compte les informations spécifiques aux joueurs (noms, ressources, vue de la carte, quantité de mana).

2.1.1 . État des éléments fixes sur la carte

La carte est formée d'une grille d'éléments nommés « cases ». La taille de la grille est définie à l'avance et est directement lié au nombre de case (par exemple une grille 128x128 contient 16 384 cases). Chaque case de la grille correspond à une position sur la carte (avec les coordonnées x et y correspondantes).

Les éléments fixes sur la carte se partagent en deux catégories :

- **Les éléments pouvant être possédés par un joueur :**
    - Les bâtiments et leurs différents types (camp d'entraînement, mine de mana, QG et village) et leur statut de contrôle (« neutre » ou « possédé » par un des deux joueurs). À noter qu'il n'y a que le QG qui n'est jamais neutre et qui appartient toujour au même joueur (jusqu'à sa destruction).
- **Les éléments ne pouvant pas être possédés par un joueur :**
    - Les environnements et leurs différents types (montagne, forêt et plaine). Chaque case de la grille, à part celles où se trouvent des bâtiments, contient un type d'environnement.

Il faudra donc dans un état donné, avoir toutes les informations liées à une coordonnée du quadrillage : le type d'unité qu'il y a dessus (s'il y en a), le type d'environnement qui s'y trouve (un seul par case) et le bâtiment situé à cet emplacement (s'il y en a un). Les informations liées aux bâtiment et aux unités sont également à connaître : qui les contrôles et de quel type il s'agit.

2.1.2 . État des éléments mobiles sur la carte

Les unités sont les seules entités qui ont la capacité de se déplacer sur le quadrillage. Elles ont également la possibilité d'influer sur les bâtiments et les autres unités (ils peuvent prendre le contrôle des bâtiments et peuvent attaquer les unités ennemies). Nous devons donc connaître à chaque état, leur position et la valeur de leurs différents attributs.

Exemple: Une unité de nains possède 3 combattants, la santé globale de l'unité est de 150 HP, sa portée de déplacement est de 30 et sa portée d'attaque est de 8 et la valeur de son attaque est de 40 (les valeurs ici sont purement arbitraire et servent juste d'exemple). La vision de l'unité doit également être prise en compte pour le joueur qui la contôle afin de savoir quelle partie du brouillard de guerre il peut voir.

2.1.3 . État des éléments propres au joueur

Les éléments propres au joueur permettent de distinguer les deux adversaires et de leur associer des attributs liés au gameplay :

- La perception que le joueur aura sur la carte, c'est à dire si sa vision est masquée ou non par le brouillard.
- Les éléments identifiant les joueurs : les noms des joueurs ainsi que leur couleur. Ces deux éléments servent à distinguer les unités et les bâtiments contrôlés par les joueurs. Typiquement, pour un état donné nous pourrions avoir : « Joueur 1 » - Équipe Rouge contre « Joueur 2 » - Équipe Bleu.
- Le quantité de mana que chacun des joueurs possèdent à un état donné. Cette quantité sera amenée à fortement évoluer lors d'une partie. Elle peut être nulle, mais cependant elle n'a pas de limite supérieure et ne peut être négative.
- L'état de l'armée du joueur doit être connu. Un joueur a des unités dans son armée, on veut avoir accès au nombre de ces unités, les états de chacune des unités et toutes les infos correspondantes.

2.1.4 . L'état général de la partie

L'état général de la partie doit comporter :

- Le numéro du tour actuel de jeu. (identique pour les deux joueurs)
- Un tour est composé de deux phases : la phase de jeu du Joueur 1 et la phase de jeu du Joueur 2. L'état du jeu doit connaître la phase pour savoir qui doit jouer.
- Le statut de défaite ou victoire des joueurs une fois la partie terminée.

2.2 . Conception Logiciel

Le state est fondé sur une classe state qui récapitule l'état générale du jeu :

- Qui est le joueur 1 ?
- Qui est le joueur 2 ?
- Quel est le nombre de tours ?

- Le jeu a t-il commencé et si oui qui joue ?

On donne ensuite une description de **Player** avec un statut (joue/ a gagné / a perdu), une liste de ressources (un id qui indique le type de ressource), une collection de bâtiments (liste de Building) qui associe chaque id à un Building, un objet UnitFactory qui sert à créer des Units, une couleur, une collection de Unit qui associe chaque id à une Unit. L'objet Player possède:

- une méthode *attack* pour attaquer
- de getters et setters pour l'ensemble de ses attributs.
- une méthode *init* servant à initialiser le joueur.
- une méthode *move* appelant *move* Unit.

La classe **Building** est composée d'une position, d'un identifiant de contrôle permettant de savoir si le bâtiment est neutre, ou s' il appartient au joueur 1 ou au joueur 2, amountMana qui permet de mesurer la quantité de mana du bâtiment, amountHp qui permet de mesurer la santé du bâtiment avant sa destruction, un bâtiment BuildingID qui permet d'identifier le bâtiment de manière unique, et typeID qui permet de distinguer le type de bâtiment. Building possède une unique méthode *setBuildingId* qui permet de modifier l'ID.

De cette classe hérite quatres classes :

- **Headquarter**
- **ManaMine**
- **Town**
- **TrainingCamp**

On décrit ensuite la classe **UnitFactory** qui a pour objectif d'appliquer le pattern Factory à notre projet pour la création d'objet.

La classe **Unit** contient plusieurs attributs : size qui contient le nombre de combattants dans l'unité, positions qui continent la position de l'unité, singleHpUnit qui contient les points de santé d'un unique membre de l'unité, globalHp qui contient les points de santés de l'unité complète, attackrange qui permet d'obtenir la portée de l'attaque d'une unité, sightRange qui permet d'obtenir la puissance de la vision d'un membre de l'unité, globaleDamage qui permet de quantifier les dégâts totaux infligés par une unité, moveRange qui permet de quantifier les déplacements d'une unité, unitID pour identifier l'unité de façon unique et typeId pour identifier la race de l'unité. Nous avons également ajouté un attribut globalID afin de générer des id unique pour chaque unité.

La classe contient plusieurs méthodes :

- une méthode *init* pour initialiser l'unité
- une méthode *move* pour se déplacer
- une méthode *place* pour déplacer arbitrairement l'unité à une Position donnée
- une méthode *attack* pour attaquer

De cette classe hérite les différents types d'unités :

- **Bat**
- **Gobelin**
- **Dwarf**
- **Knight**
- **Wizard**
- **Golem**

Une classe **Environnement** avec un attribut typeID pour identifier le type de biome, et allPositions pour accéder à la liste de Positions que couvre le biome. Cette classe contient deux méthodes : *getTypeID et setTypeID*.

De cette classe hérite :

- **Mountain**
- **Forest**
- **Plain**

Une classe **Position** qui contient deux attributs x et y et plusieurs méthodes :

- une méthode *changePlace* pour se déplacer de façon arbitraire
- une méthode *move* pour se déplacer
- deux méthodes *getX* et *getY*

deux méthodes *setX* et *setY*

Voici le rendu final du state.dia (une version plus visible se trouve dans le dossier src du projet) :

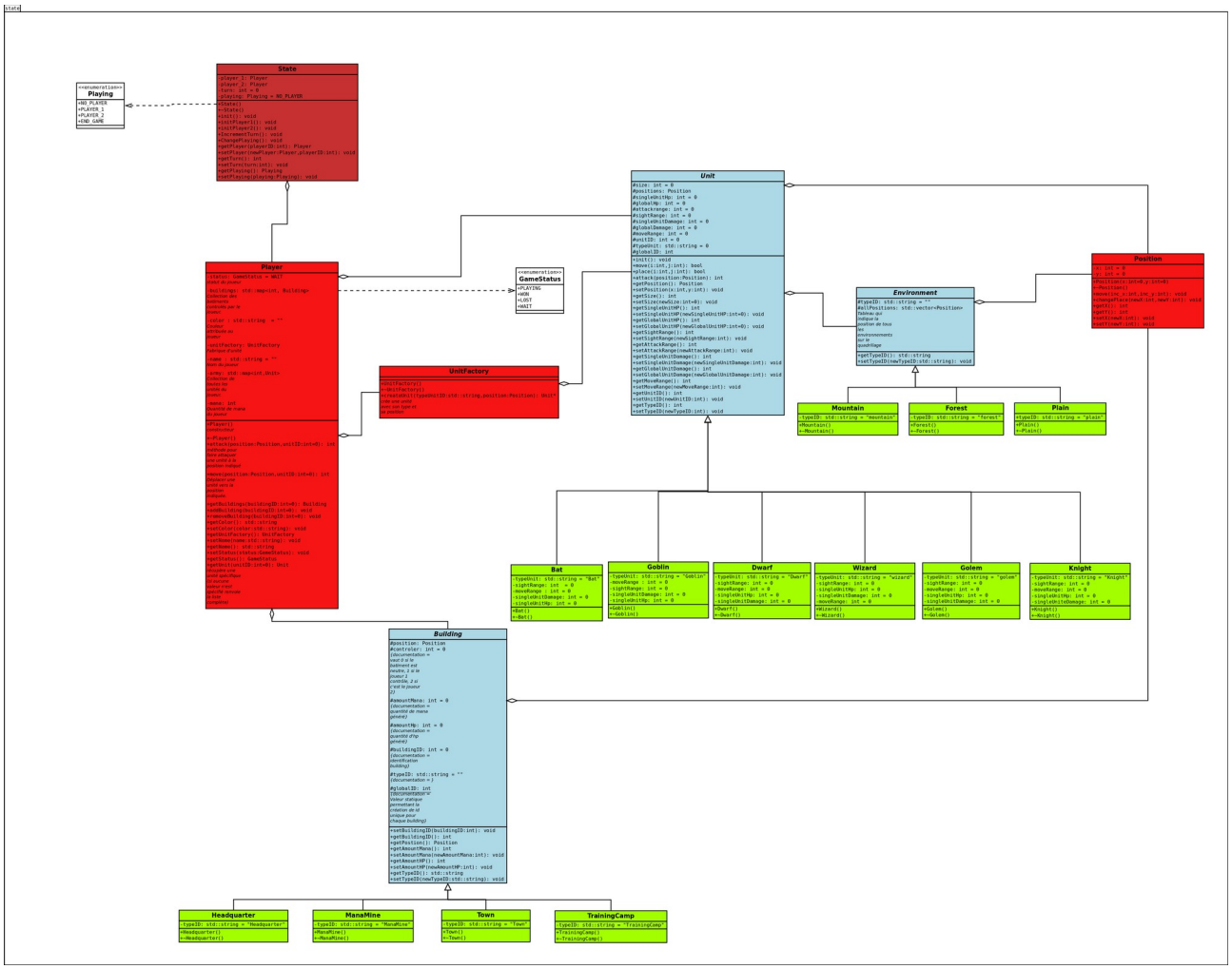

Figure 10: state.dia

À noter qu'ici un certain code couleur à été utilisé :

- les classes en bleue clair sont les classes abstraites (purement virtuelles) qui ne sont donc jamais instanciée. De gauche à droite il y a *Ressource*, *Building*, *Unit* et *Environment*
- les classes vertes sont les classes qui héritent d'une classes abstraite. Il s'agit des classes telles que *Mana* (pour *Ressource), Town* (pour *Building*), *Bat* (pour *Unit*) et *Forest* (pour *Environment*)
- les classes blanches sont les énumérations. Elles correspondent aux deux classes *Playing* et *GameStatus* (de gauche à droite)
- les classes rouges correspondent aux autres classes, celles dites que nous considérons comme « normales ». Il s'agit des classes telles que *State, Player, UnitFactory* et *Position* qui ont un rôle clé dans l'état du jeu. Il faut noter que la classe *State* a une couleur légèrement différente car elle a, comme son nom le laisse entendre, une place centrale dans la représentation d'un état.

3 . Description et conception du rendu

3.1 . Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons décidé de manipuler des tableaux de Vertex. Ces objets fournis par la bibliothèque SFML, nous permettent de faire appel à des mécanisme bas niveau pour dessiner des choses. En effet notre jeu nécessite une mise à jour constante de l'affichage, ainsi nous devons beaucoup faire appel à la fonction *draw*, cette fonction demande au GPU de changer tout un état OpenGL, de mettre en place des matrices, de changer de texture courante etc. Tout cela pour ne dessiner que quelques formes. Les tableaux de Vertex permettent une meilleure flexibilité que si on devait dessiner des sprites une par une.

Pour la stratégie de rendu, nous avons choisi décomposons les états à afficher en couche, la première couche à afficher étant le background, ensuite sur une couche supérieur nous mettrons les unités et enfin nous mettrons les informations utiles pour les joueurs, par exemple leur nom, leur quantité de mana, les unités en production etc. Chaque couche sera texturée par son propre tileset.

Pour l'instant nous nous concentrons sur les changements permanent dans le jeu, c'est à dire des changements qui modifient l'état du jeu de façon permanente. Si l'un de ces changement survient, nous mettrons à jour le state afin que le rendu soit lui même actualisé.

Les animations ne sont pas encore clairement prises en compte dans notre démarche mais nous utilisons des tilesets permettant de les incorporer dans le rendu final. Nous devons également veiller à ce que la sélection de oute entité du jeu sélectionnable soit prise en compte dans le rendu afin de donner un aspect visuel à la sélection et ainsi de rendre le jeu plus agréable à jouer.

Pour la map « de base » que nous allons passer au rendu au commencement d'une partie, nous l'avons créé à la main grâce au logiciel Tiled. Cet outil nous permet de récupérer nos maps sous forme de fichier .tmx. Une fois parsé comme il faut nous pouvons récupérer les données nécessaires au rendu de la map avec notre propre code.

Nous réfléchissons également à rendre toutes les données du state sous forme d'un fichier csv ou json afin qu'il soit plus facile au render de gérer les mise à jours sur le state.

Une fois la map importée, nous la passons à la stratégie de test. Nous allons pour cela créer plusieurs états et évaluer le rendu pour chacun d'entre eux. Idéalement le scénario d'une mini partie pourra être implémenter afin de faire apparaître les différents concepts de notre jeu.

3.2 . Conception logiciel

Nous organisons notre rendu autour de trois classes :

- **MapData** : il s'agit de la classe qui stock toutes les informations qui lui sont passé par le state. Ses attributs sont nécessaires pour tracer une Map car ils contiennent l'emplacement du fichier tileset à ouvrir en fonction de quel *MapDataType* nous voulons générer. La taille *height, width* de la map sont également passé en argument lors de l'instanciation d'une MapData ainsi que *tileSize* la taille d'une tile du tileset. Grâce à tout cela et aux informations données par le state nous pouvons générer un array *tiles* qui représente par leur id et leur position, toutes les entités à afficher.

- **TileMap** : Il s'agit d'une structure SFML-like, nous lui avons fait hériter des méthodes des classes *sf::Drawable* et *sf::Transformable.* Cette classe se charge de créer les tableaux de vertex, les texturer en fonction des informations contenues dans MapData.

- **Scene** : Scene est la classe « fenêtre » de notre render, elle se charge de cible où afficher toutes les TileMap que nous créerons et qui représenterons les différentes couches à afficher dans la fenêtre. Cette classe dépend de sf::RenderWindow qui nous permet de draw nos résultats.

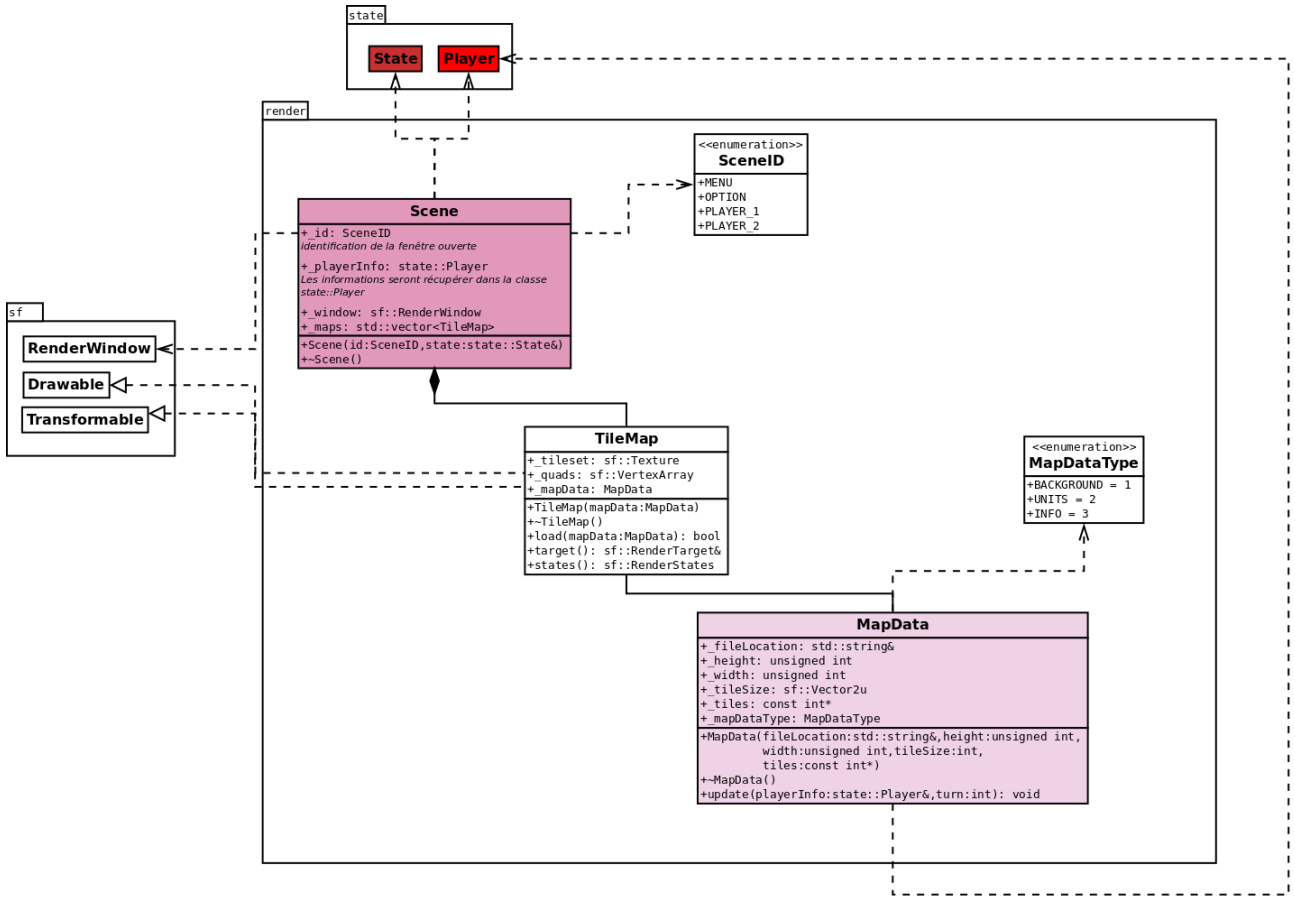

Figure 11: render.dia