# Projet ZCOM

Simon GRANGIER – Valentin HORTANED

# Table des matières

# 1 Objectif

## 1.1 Présentation générale

L'objectif est de vaincre les unités adverses en faisant progresser ses unités sur une carte quadrillée et en les faisant évoluer à chaque unité adverse détruite.

## 1.2 Règles du jeu

Le joueur commence une partie en arrivant sur une carte quadrillée, avec des *zones d'obstacles*, vue d'en haut avec un nombre d'unité équivalent à son adversaire. A chaque tour, une nouvelle unité apparaît dans la *zone d'apparition* puis chaque unité peut être déplacée et attaquer. Une unité possède un nombre de points de vie, un nombre de cases de déplacement et un type d'arme faisant plus ou moins de dégâts. Le type et la puissance d'une arme change lorsqu'une unité détruit une unité adverse. Le but est de détruire la(les) tour(s) de l'adversaire représentant sa vie.

## 1.3 Conception Logiciel

*Cmake, SFML, DIA, Boost, gcc, Microhttpd, JSON*

# 2 Description et conception des états

## 2.1 Description des états

*Un état du jeu est formée par un ensemble d'éléments fixes (le terrain) et un ensemble d'éléments mobiles (les unités). Tous les éléments possèdent les propriétés suivantes :*

*— Coordonnées (x,y) dans la grille*

*— Identifiant de type d'élément : ce nombre indique la nature de élément (ie classe)*

### 2.1.1 État éléments fixes

*Le terrain est formé par une grille d'éléments nommé « cases ». La taille de cette grille est fixée au démarrage du niveau. Les types de cases sont :*

*Cases « Obstacle ». Les cases « Obstacle » sont des éléments infranchissables pour les éléments mobiles et bloquent leur champ de tir. On considère les types de cases « Obstacle » suivants :*

*— Les espaces « montagne », qui permettent de se mettre à couvert*

*— Les espaces « rivière », par dessus lesquels il est possible de tirer*

*— Les « Objectif », qui ont une barre de vie. La destruction d'une de ces cases entraîne la victoire de l'équipe adverse.*

*Cases « Sol ». Les cases « Sol » sont les éléments franchissables par les éléments mobiles. On considère les types de cases « Sol » suivants :*

*— Les espaces « plaine » ou « pont »*

*— Les espaces « Apparition », sur lesquels apparaissent les nouvelles unités.*

### 2.1.2 État éléments mobiles

*Les éléments mobiles (unités) possède une direction (gauche, droite, haut ou bas), une position, des points de déplacements, des points de vie, une attaque et un pouvoir unique comparé aux autres unités. À chacun de leur tour, les joueurs pourront déplacer une par une leurs unités, dans l'ordre souhaité. Chaque tour, une unité peut être déplacé d'un nombre de cases égal à ses points de déplacements et peut choisir entre attaquer, utiliser son pouvoir ou terminer son action.*

*— Status « immobile » : animation d'attente jusqu'à que le status change.*

*— Status « en mouvement» : après sélection d'une unité et de la case où l'on souhaite et où il est possible de la déplacer, une animation de mouvement et un déplacement vers la case est effectué.*

*— Status « attaque » : après sélection d'une cible, une animation d'attaque se lance selon le type d'arme.*

*— Status « pouvoir » : lorsqu'une unité détruit une unité adverse, elle évolue, changeant alors ses ses propriétés, son attaque et son pouvoir.*

*— Status « apparition » : chaque tour, une nouvelle unité apparaît dans la « zone d'apparition ».*

*— Status « mort » : cas où l'unité n'a plus de vie, elle quitte le terrain.*

### 2.1.3 Etat général

*A l'ensemble des éléments statiques et mobiles, nous rajoutons les propriétés suivantes :*

*— Tour : définie le joueur pouvant effectuer des actions avec ses unités.*

*— Temps écoulé: le nombre d'époque depuis le début du tour d'un joueur.*

## 2.2 Conception logiciel

Le diagramme des classes pour les états est présenté en Figure 5, dont nous pouvons mettre en évidence les groupes de classes suivants :

Classes GameObject Toute la hiérarchie des classes filles de GameObject (en orange) permettent de représenter les différentes catégories et types de game object. C'est un exemple très classique d'utilisation du Polymorphisme. Etant donné qu'il n'y a pas de solution standard efficace[1] en C++ pour faire de l'introspection, nous avons opté pour des méthodes comme isStatic() qui indiquent la classe d'un objet. Ainsi, une fois la classe identifié, il suffit de faire un simple `static_cast<Classe>(gameObject)`.

Conteneurs d'élément. Viennent ensuite les classes Player, World et GameState qui permettent de contenir des ensembles de game object. GameState contiendra tous les objets permettant de récupérer une partie en cours. La classe World est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état initial d'une partie, permettant ainsi de relancer une partie. Et enfin, la classe Player les données et objets des joueurs qui s'affrontent.

Note 1 : Actuellement, la classe World est génère la carte dans le constructeur à partir d'un string inscrit dans le code. A terme, celui-ci appellera une fonction loadMap qui chargera les informations depuis un fichier.

Note 2 : Les classes GameState et Character ont des méthodes non implémentés. Dû au prototype généré par dia2code, nous n'avons pas réussi à attribuer une valeur aux attributs de type [Classe]* depuis les arguments de type const [Classe]&.

1 Il existe « Run-Time Type Information (RTTI)", cependant cela est peu portable et souffre de sérieux problèmes de performance.

## 2.3 Conception logiciel : extension pour le rendu

## 2.4 Conception logiciel : extension pour le moteur de jeu

## 2.5 Ressources

*Illustration 1: Les différentes unités*

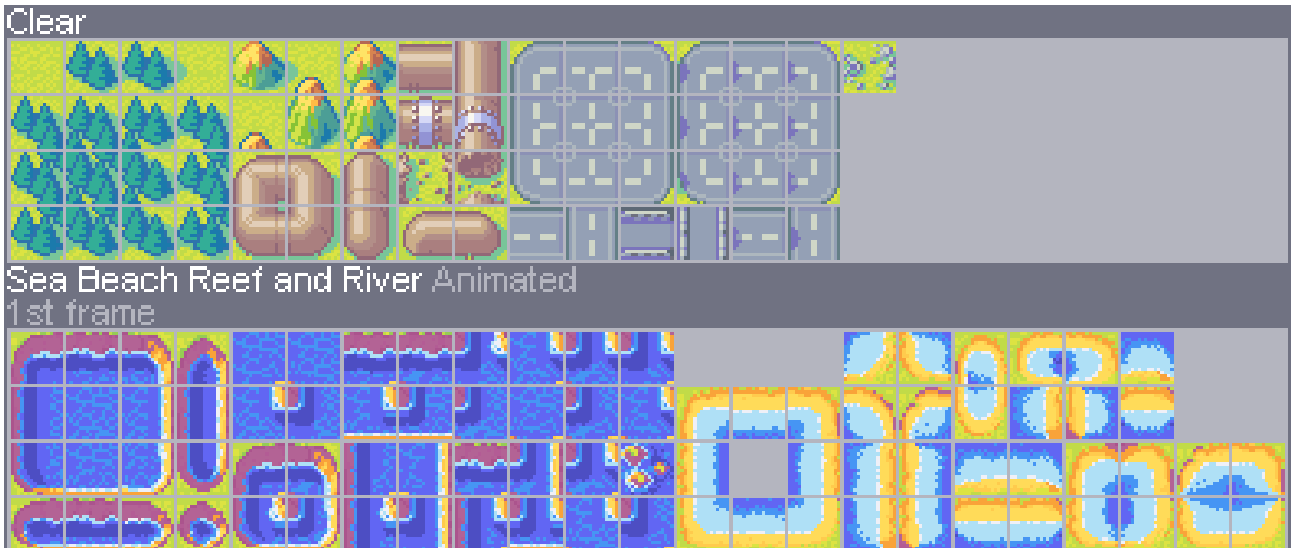


*Illustration 2: Éléments du décor*

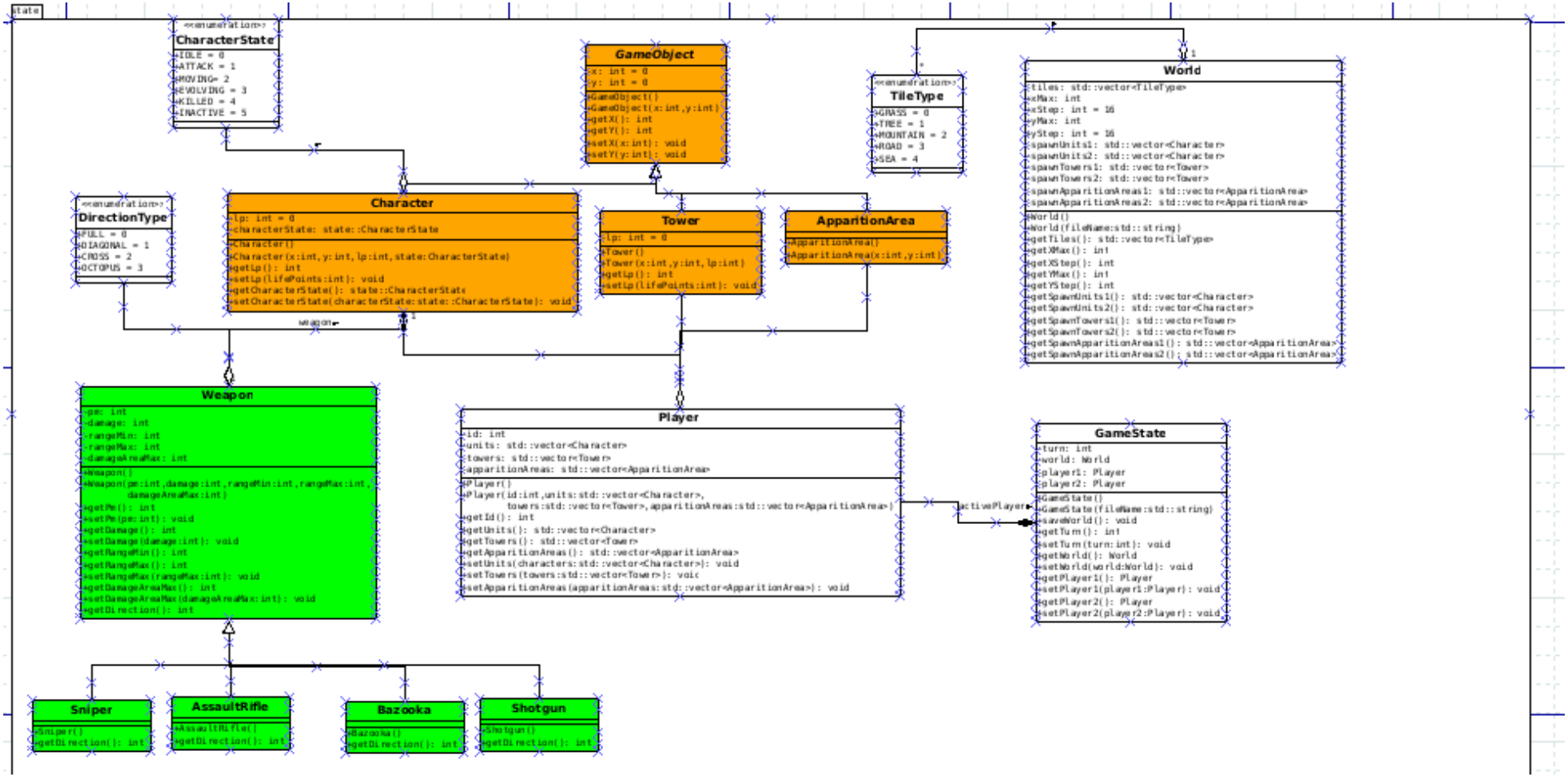


*Illustration 3: Diagramme des classes d'état*

# 3 Rendu : Stratégie et Conception

*Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.*

## 3.1 Stratégie de rendu d'un état

Notre stratégie de rendu d'un état se base sur l'utilisation de l'interface SFML qui s'appuie sur OpenGL afin de générer un rendu en 2D. Nous avons utiliser les fonctions de SFML afin de charger dans le processeur, la liste des éléments à afficher par le processeur graphique.
Nous avons découpé notre affichage graphique sur deux niveaux, la carte avec les éléments décoratifs (mur, sol, escalier) et les éléments venant à changer à savoir les gameObject (unités, tours et zones d'apparitions) qui se superposent sur la couche précédente. On transmet l'état du jeu ainsi que les textures de toute la carte avec leurs coordonnées et la texture des unités avec leurs coordonnées et leur orientation à afficher.
La carte étant très grande, nous l'avons entièrement chargé dans la mémoire pour l'affichage.
Lorsque qu'un changement d'état se produit, la vue est modifiée en fonction du changement appliqué à l'état. Si le changement modifie uniquement la position des unités, seul le rendu des unités est mis à jour, si ce changement s'applique à l'environnement l'ensemble du rendu de la carte est mis à jour.
Lorsque l'ensemble de l'état est modifié le rendu de l'état est entièrement mis à jour. Dans le cas où le jeu est fini la fenêtre est automatiquement fermée.
Nous avons rajouté sur la scène finale, certains attributs des unités (point de vie actuelle et id des unités) en bas de la fenêtre.

## 3.2 Conception logiciel

Pour afficher un état on crée une scène qui génère un ensemble de pointeurs sur des objets graphiques de types LayerRender pour la carte. Pour les unités on instancie un Sprite par unité. Ce Sprite contient à la fois les coordonnées de l'unité sur la fenêtre et sa position sur le Tileset utilisé. Puis, on utilise la méthode Scene : :update() pour déclencher l'ouverture de la fenêtre et l'affichage de l'état. Il est à noter que l'état contient d'ores et déjà toutes les informations nécessaires à l'affichage de la carte. En effet, à la création de l'état celui-ci parse un fichier texte (ici un string dans main.cpp)contenant toutes les informations nécessaires à l'affichage de la carte.
classe Scene est un des Observer de GameState. Ainsi, lorsque l'état subit un changement, Scene est notifiée et met à jour le rendu en fonction du type d'évènement qui lui est transmis :
— Si un PLAYER_EVENT (correspondant à changement d'état pour l'un des joueurs) la méthode update-Players() responsable de la mise à jour du rendu des unités est appelée.
— Si un WORLD_EVENT (correspondant changement d'état lié à l'environnement ou à la partie) la méthode updateMap().
Par ailleurs, les points de vie de chacunes des unités sont, pour le moment, affichés et mis à jour en temps réel à l'intérieur de la méthode draw() de la classe Scene.
Chaque objet LayerRender parcourt l'ensemble des tuiles d'un étage de la carte, détecte la position de la tuile sur la ressource graphique (i.e : l'image du tileset) et calcule sa position sur la fenêtre. Pour optimiser les performances seule l'image du tileset est chargée dans une Texture, l'objet

LayerRender conserve seule ment la position de chaque tuile de la carte sur cette texture. Les unités sont chacun rendus à l'intérieur d'un Sprite. Lorsqu'un mouvement de l'un des unités est détecté le la classe scène met à jour l'affichage de l'ensemble des unités.

## 3.3 Conception logiciel : extension pour les animations

## 3.4 Ressources

## 3.5 Exemple de rendu

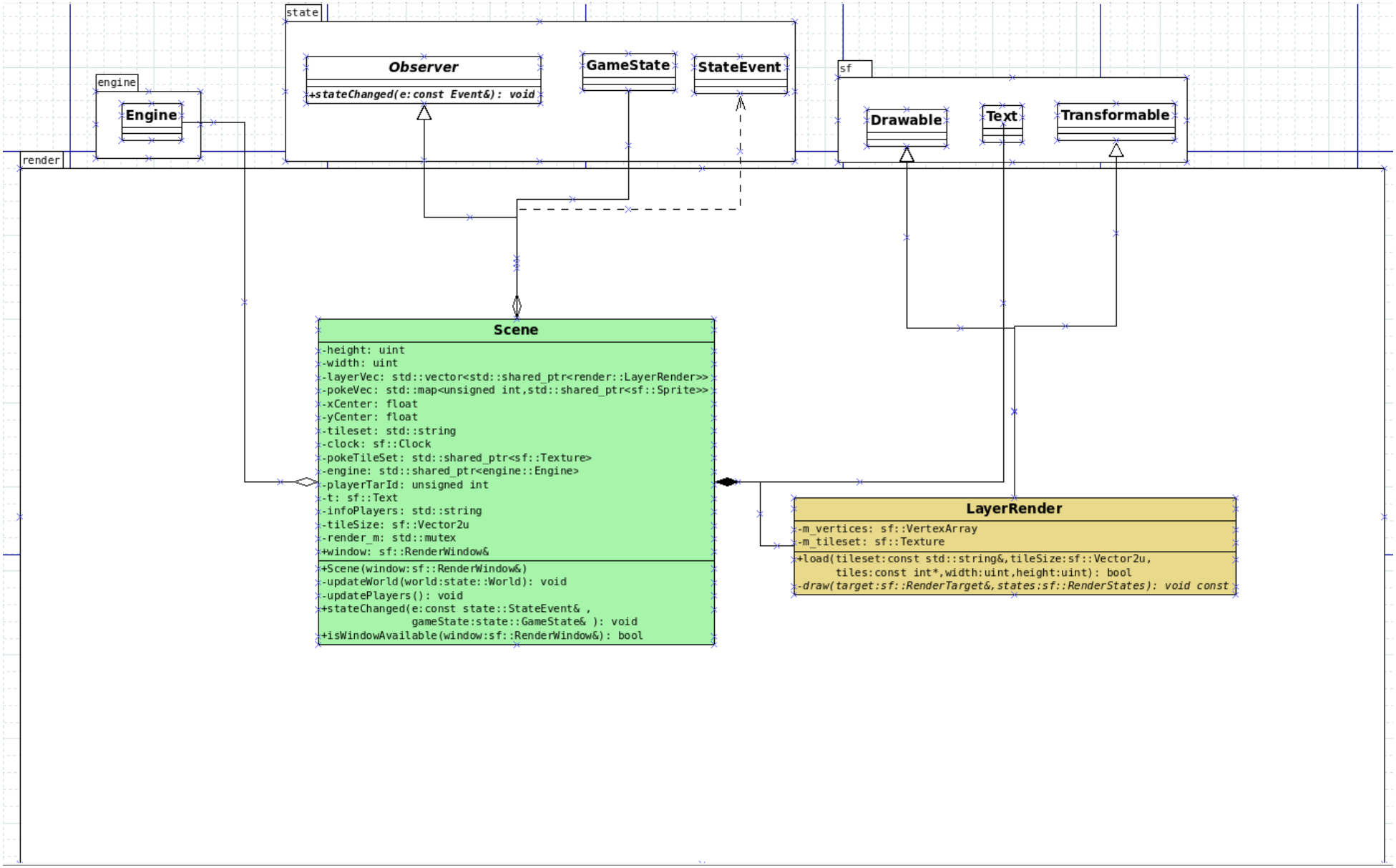


*Illustration : Diagramme de classes pour le rendu*

# 4 Règles de changement d'états et moteur de jeu

*Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.*

## 4.1 Horloge globale

## 4.2 Changements extérieurs

## 4.3 Changements autonomes

## 4.4 Conception logiciel

## 4.5 Conception logiciel : extension pour l'IA

## 4.6 Conception logiciel : extension pour la parallélisation

*Illustration 4: Diagrammes des classes pour le moteur de jeu*

# 5 Intelligence Artificielle

*Cette section est dédiée aux stratégies et outils développés pour créer un joueur artificiel. Ce robot doit utiliser les mêmes commandes qu'un joueur humain, ie utiliser les mêmes actions/ordres que ceux produit par le clavier ou la souris. Le robot ne doit pas avoir accès à plus information qu'un joueur humain. Comme pour les autres sections, commencez par présenter la stratégie, puis la conception logicielle.*

## 5.1 Stratégies

### 5.1.1 Intelligence minimale

### 5.1.2 Intelligence basée sur des heuristiques

### 5.1.3 Intelligence basée sur les arbres de recherche

## 5.2 Conception logiciel

## 5.3 Conception logiciel : extension pour l'IA composée

## 5.4 Conception logiciel : extension pour IA avancée

## 5.5 Conception logiciel : extension pour la parallélisation

# 6 Modularisation

*Cette section se concentre sur la répartition des différents modules du jeu dans différents processus. Deux niveaux doivent être considérés. Le premier est la répartition des modules sur différents threads. Notons bien que ce qui est attendu est un parallélisation maximale des traitements: il faut bien démontrer que l'intersection des processus communs ou bloquant est minimale. Le deuxième niveau est la répartition des modules sur différentes machines, via une interface réseau. Dans tous les cas, motivez vos choix, et indiquez également les latences qui en résulte.*

## 6.1 Organisation des modules

### 6.1.1 Répartition sur différents threads

### 6.1.2 Répartition sur différentes machines

## 6.2 Conception logiciel

## 6.3 Conception logiciel : extension réseau

## 6.4 Conception logiciel : client Android

*Illustration 5: Diagramme de classes pour la modularisation*