ENSEA
Ecole Nationale
Supérieure
de l'Electronique
et de ses Applications

Projet Zorglub

CHARLET Justine, DESCATOIRE Théau
MONTMAUR Antoine, SATHIAKUMAR Kugappiriyan

6 Octobre 2020

Table des matières

Chapitre 1

Objectif

1.1 Présentation générale

Le Projet Logiciel Transverse (PLT) est un projet de 120h destiné à la conception d'un jeu vidéo de type tour par tour, avec un réseau multi joueur en ligne. Il fait intervenir la plupart des cours dispensés en Informatique et Systèmes à l'ENSEA : génie logiciel, algorithmique, programmation parallèle et web services.
Au cours de ce projet, nous allons faire de la conception UML, réaliser un moteur d'affichage, créer une IA basique, une heuristique puis l'améliorer pour qu'elle soit performante, et enfin développer ce jeu en multijoueur, de façon à pouvoir jouer en ligne.

1.2 Outils pour la réalisation du jeu

Ce projet est développé en langage C++, très adapté pour les jeux vidéos. Nous avons décidé de travailler soit sous VM, soit sur un PC avec un environnement Linux. Le projet est développé sous Ubuntu 18.04. La bibliothèque SFML version 2.4.2 a été utilisé pour le rendu graphique .

Nous avons créé un projet GIT en clonant le projet du GIT de M. Barès et nous avons créé une branche dev qui sera notre environnement avant l'implémentation finale qui sera merge dans la branche Master. De plus, nous avons crée un projet Trello afin de mieux visualiser les tâches restantes et définir les rôles de chacun au sein du projet.

1.3 Archétype du jeu

L'objectif de notre projet est la réalisation d'un jeu vidéo de type tactical RPG (T-RPG), en tour par tour, basé sur le jeu Donjon de Naheulbeuk.

FIGURE 1.1 – Logo de Donjon de Naheulbeuk

Le jeu est basé sur un PVP multijoueur (pour l'instant 2 joueurs) qui possèderont chacun un nombre de personnages prédéfinis. Le vainqueur est le seul joueur ayant encore des personnages en vie.

1.4 Règles du jeu

1.4.1 Les personnages et leurs maps

Nous avons créé nos propres personnages et nos propres règles.
Chaque joueur doit choisir une équipe de 3 personnages parmi les suivants : Elfe, Indien, Nain, Bandit, Chevalier, Troll et Pirate.

Le jeu comporte 7 maps représentant les 7 thèmes associés aux personnages :

1. Jungle (elfe)
2. Plaine (indien)
3. Montagne (nain)
4. Forêt (bandit)
5. Château (chevalier)
6. Grotte (troll)
7. Bateau (pirate)

.

.

(a) Elfe (b) Indien (c) Nain

(d) Bandit (e) Chevalier (f) Troll

(g) Pirate

FIGURE 1.2 – Liste des personnages possibles

1.4.2 Les attributs principaux, secondaires et les compétences

A chaque personnage est associé des attributs principaux et secondaires. La répartition des points pour chaque attribut principal se fait par un système d'achat de points :

Attributs principaux
$8 \leq Force \leq 15$
$8 \leq Adresse \leq 15$
$8 \leq Endurance \leq 15$
$8 \leq Courage \leq 15$
$8 \leq Intelligence \leq 15$
$8 \leq Arcane \leq 15$

.
Tous les attributs principaux sont à 8 par défaut et le joueur dispose de 27 points pour améliorer les capacités de son choix, chaque capacité n'excédant

pas 15. Si un attribut dépasse 13, il faut alors 2 pts pour passer à 14, puis à 15.

Les points attribués aux attributs secondaires sont calculés automatiquement en fonction des attributs primaires de la manière suivante :

Attributs secondaires	
Points De Vie	3 * Endurance + 2 * Force
Precision	(Adresse + Force + Intelligence + Arcane)/60
Esquive	((Adresse + Intelligence - 16)^2/196) * 0,33
Mouvement	5 + $\lfloor$ 1/12 $\rfloor$ * endurance + $\lfloor$ 1/12 $\rfloor$ * Courage
Initiative	Courage + Intelligence + Arcane

.

Il est aussi possible d'utiliser les compétences des personnages, établies comme suit :

Corps-à-corps	**A distance**	**Magie**
- 2 pt d'actions (ts ls 3 tours) *Le joueur adv ne joue pas*	Pluie de projectiles (3) *Joueur adv essuie 3 attaques*	Téléportation (3)
Pas de mouvement (2)	Rebonds (2) *L'attaque rebondit 2 fois*	Désarmement (2)

.

Chaque compétence vaut 1 point d'action et ne dure qu'un tour. Ces compétences ont un temps de rechargement exprimé en nombre de tours (voir tableau ci-dessus). L'implémentation d'au moins une compétence par personnage sera notre objectif principal.

Voici quelques fonctionnalités qu'on pourra mettre en place si le joueur a accès à une interface graphique permettant le choix des personnages, des statistiques et de la sélection de la map.

1. **Bonus thématique**
   De plus, si un personnage joue dans la map de même thème (ex : Elfe dans la jungle), alors le personnage se voit attribuer un bonus de 5 points de vie.
2. **Bonus Taxon** De même, chaque personnage a un bonus lié à son origine, selon le tableau ci-dessous :

Bonus de Taxon	
Elfe	+1 en Adresse
Indien	+1 en Arcane
Nain	+1 en Endurance
Bandit	+1 en Intelligence
Troll	+1 en Force
Chevalier	+1 en Courage
Pirate	+1 au choix dans un attribut primaire (car c'est un voleur)

1.4.3 Le tour par tour

Perspective du GamePlay : A chaque tour, le joueur dispose de 2 points d'action. Pour chaque point, il peut soit se déplacer, soit attaquer, soit utiliser une compétence. Pour chaque tour, il n'est pas possible de réaliser une attaque et une compétence ou deux attaques.

Les attaques

Trois types d'attaques sont disponibles : le corps-à-corps, attaque à distance, ou attaque magique. Pour chaque arme, une portée minimale et maximale est donnée dans le tableau ci dessous :

Types d'attaques								
Corps-à-corps			**A distance**			**Magique**		
	Portée			**Portée**			**Portée**	
Arme	Min	Max	**Arme**	Min	Max	**Arme**	Min	Max
Epée	1	2	Arc	2	8	Baguette	3	4,5
Hache	1	1,5	Arbalette	2	7	Bâton	3	4
Lance	1	2,5	Fronde	2	6,5	Bracelets	3	5

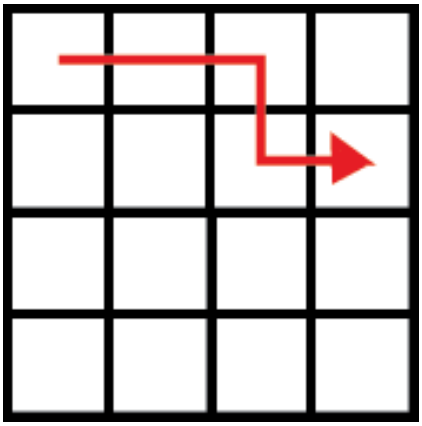
Les types d'arme ainsi que les dégâts associés sont présentès sur la liste suivant :

Arme	Dégâts
Epée	13
Hache	14
Lance	12
Arc	6
Arbalette	7
Fronde	8
Baguette	10
Bâton	11
Bracelet	9

.

Les déplacements

Chaque personnage peut se déplacer de 5 unités de distance : selon la verticale et/ou horizontale pour une valeur de 1 unité par case. Le joueur peut également avoir un bonus de 1 ou 2 unité de distance s'il choisit de maximiser les attributs tels que l'endurance et courage.



(a) 4 cases de distance

1.5 Conception Logiciel

FIGURE 1.4 – Texture des personnages

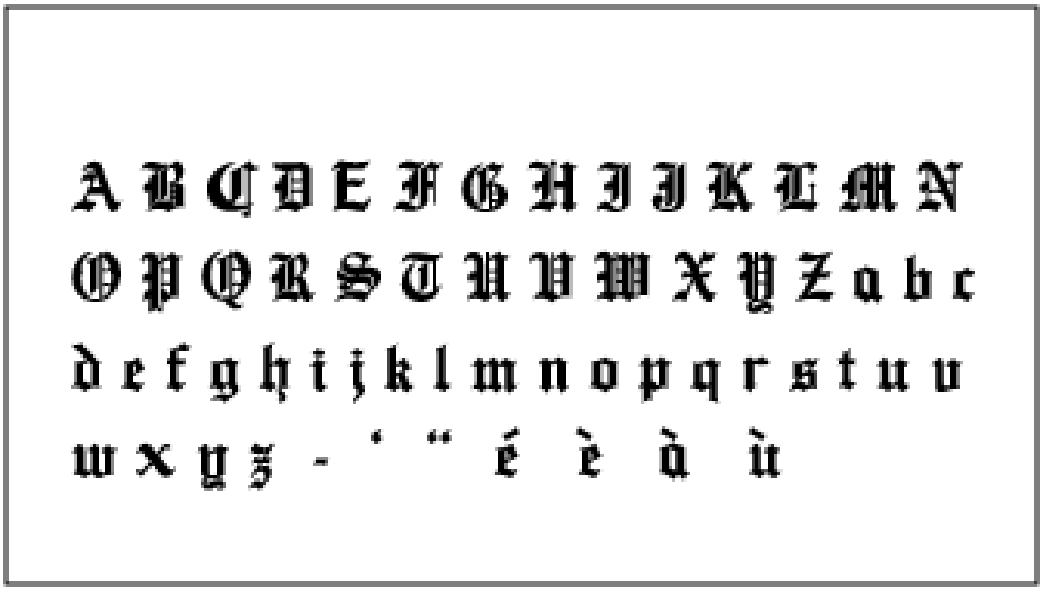
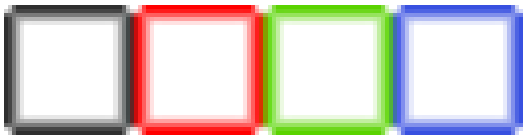


FIGURE 1.5 – Texture de la police

Textures de Fond



Textures des curseurs

FIGURE 1.7 – Textures des personnages

Chapitre 2

Description et conception des états

2.1 Etat du jeu

Un état du jeu est formée par deux équipes de personnage (entre 3 et 6 personnages par équipe) et une map. Tous les éléments (map et personnages) sont représentés par des carrés composants la map. Ces carrés possèdent :
— Une coordonnée (x,y)
— Un identifiant

Les map sont au nombre de 7, rectangulaires et de tailles identiques.
Le fond
Il y a toutes sortes de texture qui s'adaptent aux décors voulu ; texture de forêt dense pour la jungle, de l'herbe pour les plaines, des rochers pour les montagnes, des arbres pour la forêt, des pierres pour le château fort, de la terre pour la grotte, et des planche de bois pour le sol du bateau. La map constitue le premier layer du rendu.

Les décors
Par ailleurs, des décors seront apposé sur cette texture de fond (comme des tonneaux, des trons d'arbres ou des rochers imposants. Ces décors seront infranchissable par les personnages et bloqueront la portée de leur arme. Les décors contituent le deuxième layer du rendu.

Les personnages
Les différents personnages n'intéragissent pas avec la texture de fond. Ils constituent le 3ème layer du rendu.

2.2 Diagramme de classes

Le diagramme des classes est représenté ci-dessous. Voici les classes qui le compose :

(Dans chaque classe, on retrouve un constructeur et un destructeur ainsi que les setters et les getters, car nos attributs sont privés)

Classe Player

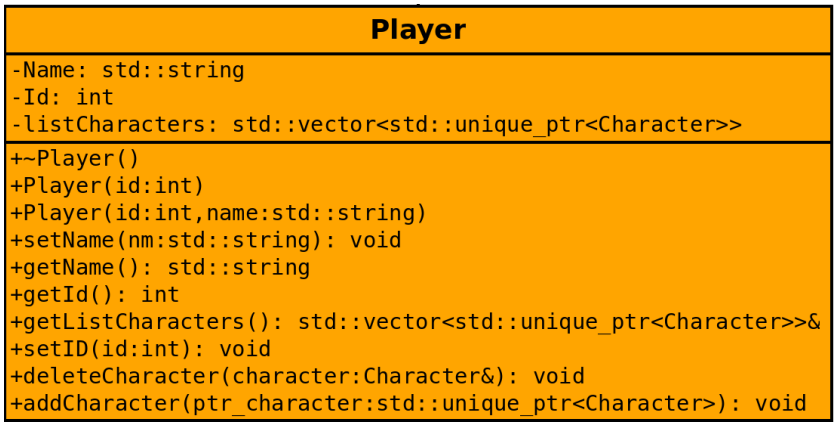


FIGURE 2.1 – Class Player

La classe Player permet de créer les joueurs du jeu (qui sont au nombre de 2). Chaque joueur à un *name*, un *Id* et une *ListCharacters* qui contient entre 3 et 6 personnages. Ces attributs sont privés. Nous avons ajouté les accesseurs pour ces attributs (le fait qu'ils soient privés nous implique de creer des getters pour récupérer la valeur de ces attributs).
La fonction *addCharacter* prend en paramètres un *unique_ptr*. cette fonction vient ajouter une valeur dans *listCharacters*, qui est un vecteur redimensionnable composé de *unique_ptr*.
A l'inverse, *deleteCharacter* permet de supprimer un personnage, si le joueur s'est trompé dans la selection d'un ou plusieurs personnage(s).

Classe Character

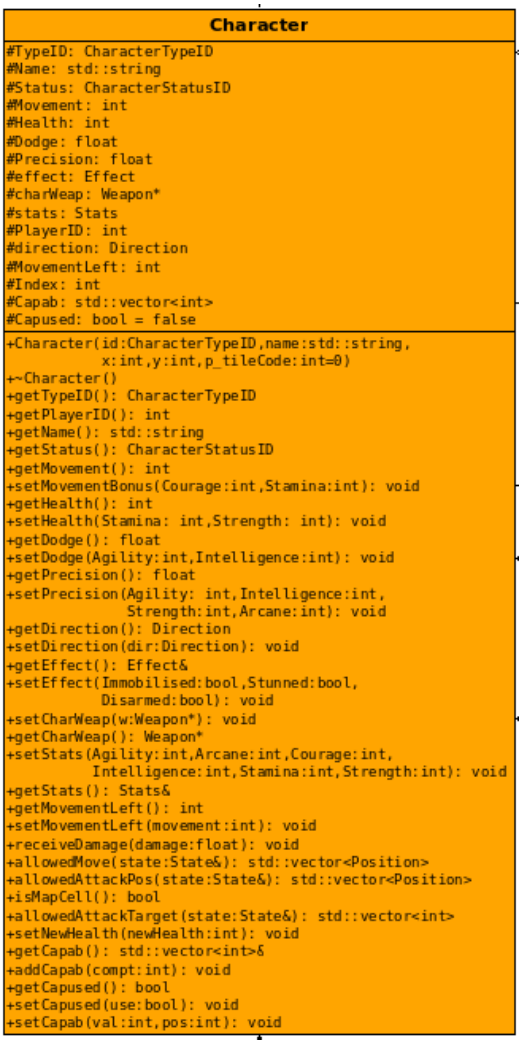


FIGURE 2.2 – Class Character

Chaque personnage possède : un *TypeID*, dont les valeur sont données grâce à une énumération. On utilise les énumération pour optimiser le code. Il en est de même pour les attributs *Status stats* et *direction*.
Un personnage possède également un *Name*, un *Movement* pour donner son nombre de case de déplacement maximal, un *Health* pour ses points de vie, des attributs *Dogde* et *Precision* pour la capacité d'esquive et de présicion lors d'un attaque, un *effect* qui dépend de la classe Effect, et pour finir un *charWeap* qui permet à chaque personnage de porter une arme.

Classe Weapon

La classe character dépend de la classe Weapon. Weapon possède les attributs *Owner*, un *typeWeapon* qui utilise l'énumération WeaponID, qui elle-même est en charge de déterminer le type d'arme utilisé (Sword, axe, spear ...).

Pour la classe Weapon, on aurait pu faire une interface et faire hériter chaque arme de la classe mère, et utiliser le polymorphisme, mais l'énumération était plus rapide et tout aussi performante.

Classe Effect

Lorsqu'un personnage subit une attaque, celui-ci passe dans un état qui peut être étourdit, brulé... Cet état est représenté dans la classe Effect.
Il y a une image qui va être supperposée à l'image du personnage. Voir figure 3.1.

Classe Element

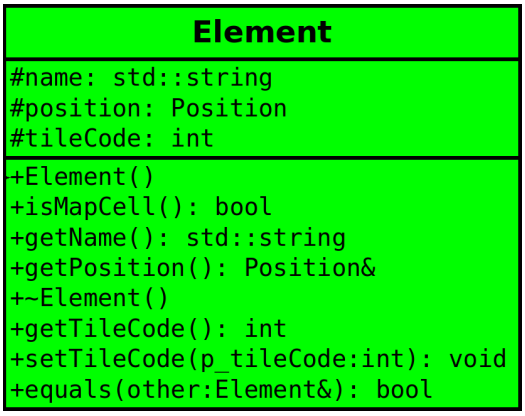


FIGURE 2.3 – Class Element

La classe Element implémente la classe *Cursor* et *MapCell*. *Element* possède plusieurs instances de la classe *Position*.

Classe State

La classe State représente l'état du jeu. State possède toutes les données nécéssaires à l'éxecution d'un état : *listCharacter* pour la liste de personnages par joueur, le *cursor*, la *map*, le numéro du *round*, l'ID du joueur actuel *curPlayerID*, un *GameEnd* qui retourne False tout le long du jeu, et qui passe à True une fois le jeu terminé. Le *mode* permet de choisir le mode de jeu,

FIGURE 2.4 – Class State

c'est-à-dire mode 2 joueurs, mode solo contre une IA, mode IA contre IA.
On a ajouté tout les accesseurs pour toutes les variables (puisqu'elles sont privées).
De plus, on a créer des fonctions d'initialisation de Map (*initMap*) de Character (*initCharacter*) et de Player (*initPlayer*). Nous avons le constructeurs de state et son destructeur.

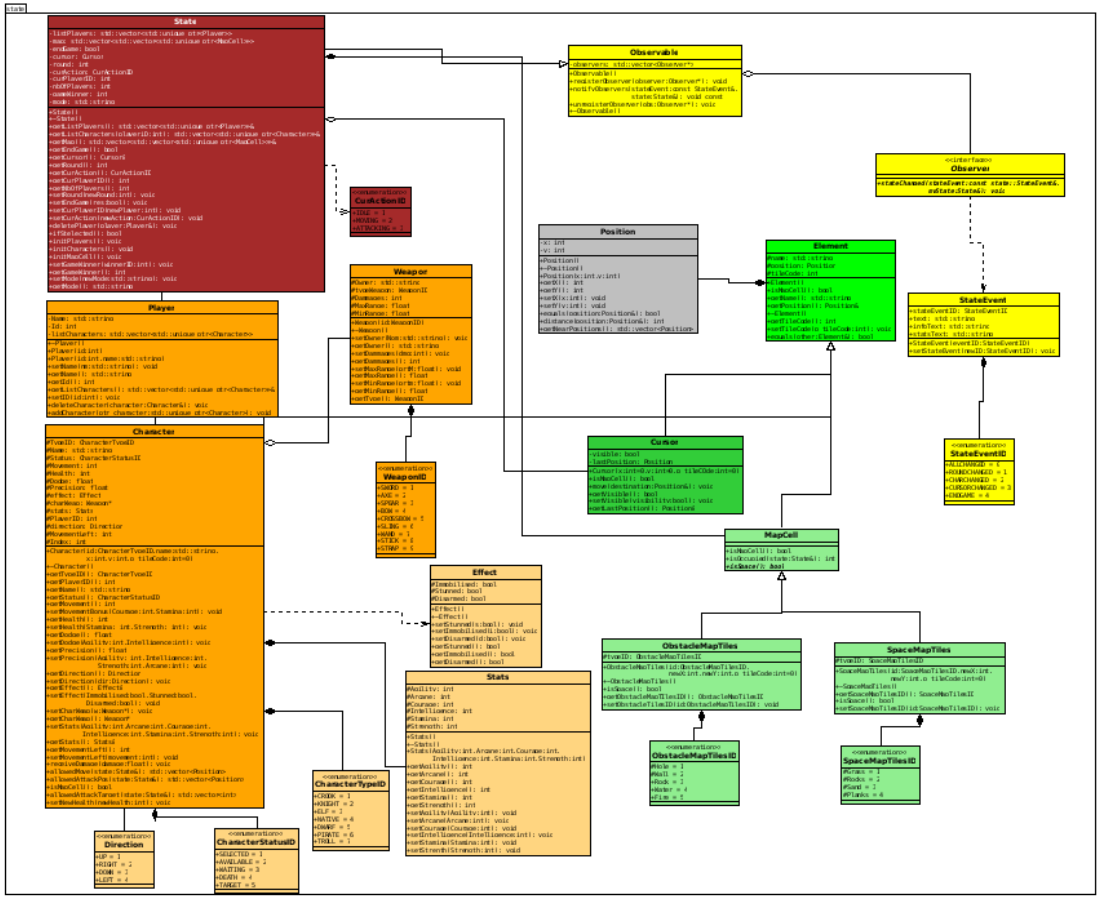


FIGURE 2.5 – Diagramme de classe state

Chapitre 3

Rendu

3.1 Stratégie et Conception

Pour le rendu de l'application : nous avons utilisé Adobe Illustrator pour réaliser les personnages, les états des personnages (étourdissement, immobilité, désarmement), ainsi que le logo Zorglub et l'initialisation du jeu.
Pour les autres décors et les bouttons nous avons utilisé la bibliothèque SFML.

Pour le rendu d'un état, nous avons utilisé 3 layers : un plan pour le sol, selon les différentes map ; un plan pour les différents obstacles que les personnages peuvent rencontrer ; et un plan pour les personnages.

Classe TextureArea

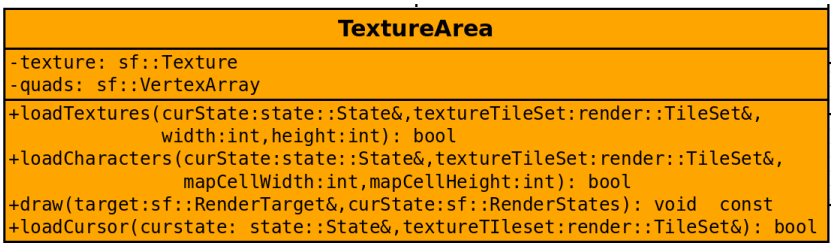


FIGURE 3.1 – Class TextureArea

La classe *TextureArea* permet de sélectionner, choisir et appliquer les textures pour les différentes tuiles. Ses méthodes de type *load* servent à les charger et la méthode *draw* les applique.

TileSet

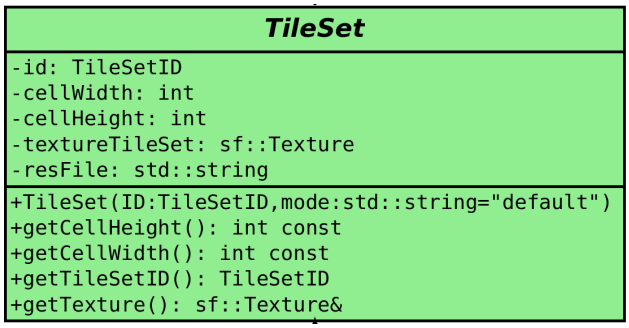


FIGURE 3.2 – Class TileSet

La classe TileSet permet de récuperer des tuiles et de les instancier. Cette classe associe la tuile à sa texture liée à *sf*. Elle n'a pour méthodes que les accesseurs et constructeurs classiques.

TileSetID

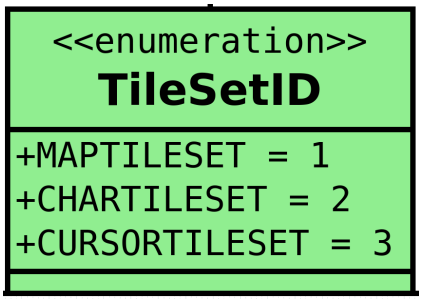


FIGURE 3.3 – Class TileSetID

La classe *TileSetID* est une simple énumération des différents types de tuiles : carte, personnage ou curseur.

Classe StateLayer

La classe *StateLayer* permet de concaténer les rendus et de gérer l'affichage complet une fois que ceux-ci ont été générés.

Pour chaque état, on met à jour la position du personnage et son état (effet des compétences). On met également à jour les points de vie des personnages et les points d'action.

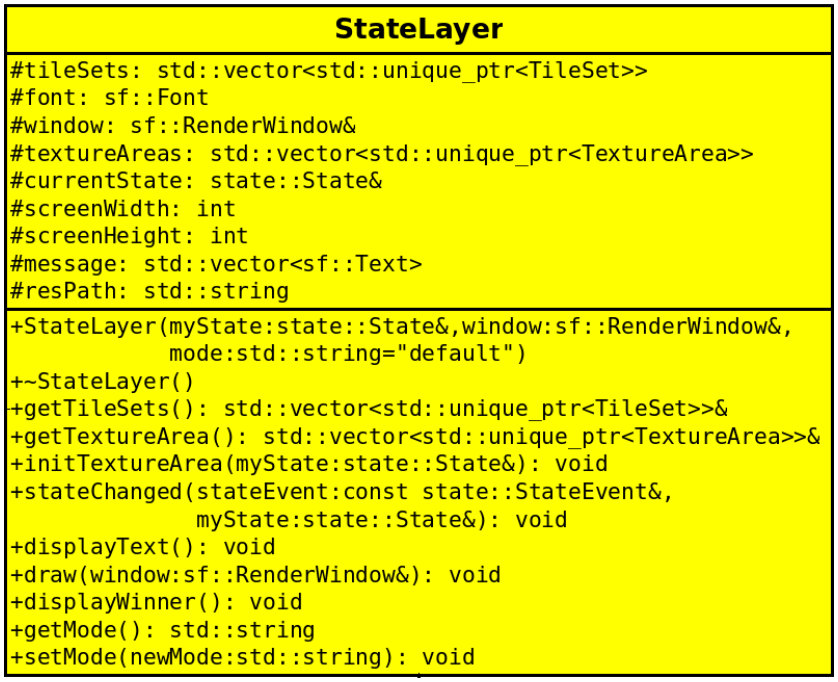
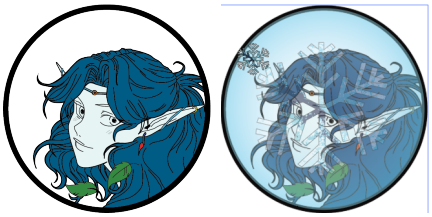


FIGURE 3.4 – Class StateLayer



(a) Elfe sans effets

(b) Elfe immobilisée

(c) Elfe désarmée

(d) Elfe étourdie

FIGURE 3.5 – Liste des effets possibles sur les personnages

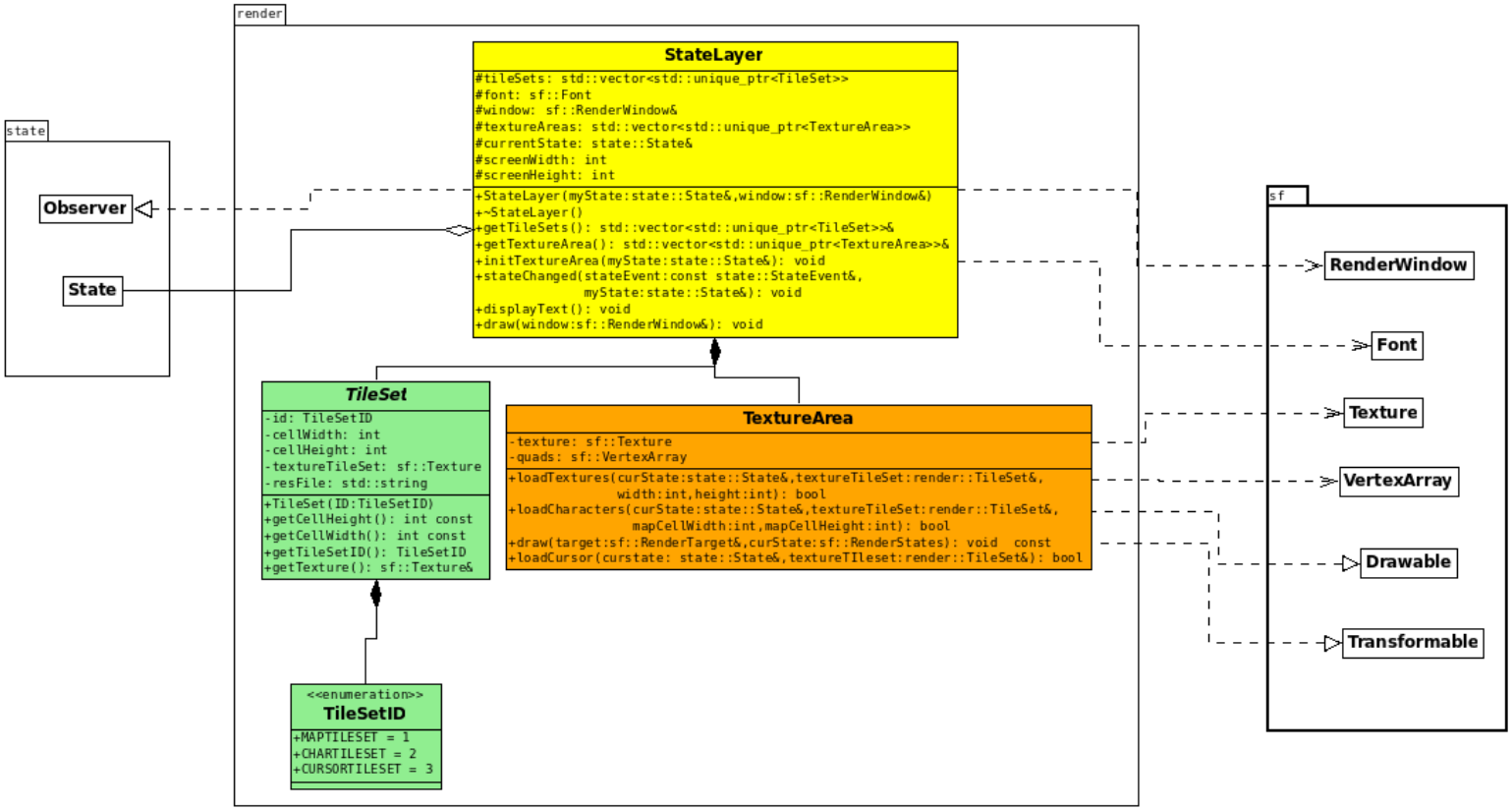


FIGURE 3.6 – Diagramme de rendu du jeu

Chapitre 4

Moteur de Jeu

4.1 Action de l'utilisateur

Le joueur dispose d'un menu, sur lequel il peut choisir (avec un clic de souris sur un boutton) d'attaquer, d'utiliser une compétence, ou de se déplacer. Il peut :
- Attaquer et se déplacer
- Se déplacer et attaquer
- Utiliser sa compétence et se déplacer
- Se déplacer et utiliser sa compétence
- Se déplacer deux fois

S'il attaque ou s'il utilise sa compétence, le joueur clique sur le boutton associé et il doit choisir la case sur laquelle il veut l'utiliser avec un clic de souris.
De même, s'il choisit de se déplacer, il clique sur le boutton associé et il doit choisir sur quelle case il veut aller, dans la limite des cases dont il a le droit.

4.2 Actions internes

Chaque point d'action est immédiatement exécuté. Lors d'un déplacement, la position du joueur est immédiatement actualisée ; lors d'une attaque, les Points de vie de l'adversaire sont diminués, de même pour les compétences.
Un compteur est incrémenté à chaque point de vie. C'est utile pour les compétences ; certaines ne peuvent être utilisées que tout les 2 ou 3 tours.
Enfin on vérifie que chaque joueur possède au moins encore un personnage capable de se battre, c'est à dire avec des Points de vie non nuls. Ceci est vérifié avec la fonction Check_Win.

4.3 Conception du logiciel

Classe Command
Cette classe est une classe abstraite qui décrit toutes les commandes. Elle ne possède qu'un attribut, l'identifiant de la classe de commande spécifique qui sera instanciée. Elle implémente les accesseurs pour toutes les classes qui en héritent puisqu'aucune n'a besion de changer l'attribut. En revanche la méthode *exec* est virtuelle car chaque classe fille l'implémentera spécifiquement en fonction des actions qu'elle doit effectuer.

CommandID
La classe commandID est une simple énumération des différentes commandes. C'est une table de correspondance entre les commandes et leur identifiant.

Move_Command
Cette classe commande le déplacement des personnages. Elle hérite de la classe commande. Ses attributs spécifiques sont *targetedChar* qui représente le personnage qui va se déplacer et *targetedPos*, la position qu'il veut atteindre. La méthode *exec* vérifie que le personnage est sélectionné, qu'il n'est pas immobilisé et qu'il lui reste des mouvements en réserve. Puis, elle s'assure que la position visée est atteignable. Si oui, elle déplace le personnage et affiche sa position. Sinon, elle prévient que le déplacement est impossible.

Attack_Command
La classe Attack_Command est la classe qui permet d'attaquer un personnage adverse. Elle hérite de la classe commande. Ses attributs spécifiques sont *attacker* qui identifie le personnage attaquant et *target* qui identifie sa cible. La méthode *exec* vérifie que le personnage est sélectionné, que la cible peut être attaquée et qu'il ne s'agit pas d'un tir allié. Puis, en fonction de la précision de l'attaquant et de l'esquive de la cible, la fonction modifie les dégats reçus.

Capab_Command
La classe Capab_Command est la classe qui permet d'utiliser ses capacités spéciales. Elle hérite de la classe commande. Ses attributs spécifiques sont *user* qui identifie le personnage actif, *target* sa cible éventuelle et *targetedPos* sa nouvelle position éventuelle. La méthode *exec* vérifie que le personnage

est sélectionné, qu'il peut utiliser ses pouvoirs (compteur de 3 tours) et effectue l'actuion correspondant à sa capacité. La méthode peut immobiliser un adversaire en influant sur la classe *Effect* ou bien réaliser une attaque triple selon le mécanisme d'*Attack_ Command* ou bien encore téléporter l'utilisateur vers la position souhaitée.

Sel_Char_Command
La classe Sel_Char_Command est la classe qui permet de sélectionner un personnage. Elle hérite de la classe commande. Son attribut spécifique est *targetedChar* qui identifie le personnage sélectionné. La méthode *exec* vérifie que le personnage ciblé est bien celui à sélectionner dans l'état actuel du jeu.

Finish_Turn_Command
La classe Finish_Turn_Command est la classe qui permet de terminer un tour. Elle hérite de la classe commande. La méthode *exec* vérifie que toutes les étapes du tour ont été effectuées, elle passe au tour du joueur suivant et update le render.

Check_Win_Command
La classe Check_Win_Command est la classe qui permet de vérifier s'il y a un gagnant. Elle hérite de la classe commande. La méthode *exec* vérifie si un joueur n'a plus de personnages et si oui, déclare l'autre vainqueur.

Classe Engine
Cette classe gère les états successifs du jeu et les commandes à executer. Elle possède les attributs *currState*, qui est l'état actuel du jeu et *currCommands*, une liste de pointeurs vers les commandes à exécuter. Outre les accesseurs de ces attributs, cette classe possède la méthode *update* qui met à jour l'état du jeu en fonction des commandes précitées

Classe engineObservable
Cette classe est la classe qui rend compte des diverses commandes effectuées. Elle est similaire à l'*Observable* présent dans le package *State* (cf. Supra). De même, l'interface *engineObserver* est identique à son alter ego de *State* (cf. Supra).

FIGURE 4.1 – Etat MOVING engine

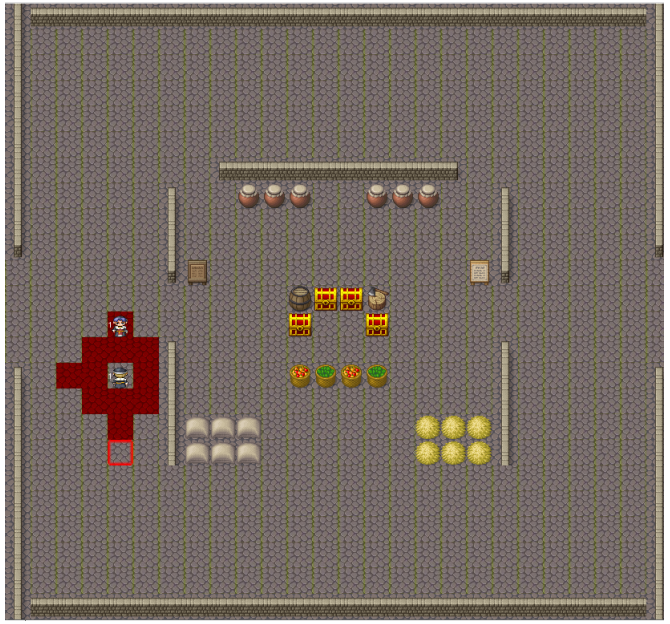


FIGURE 4.2 – Etat ATTACKING engine

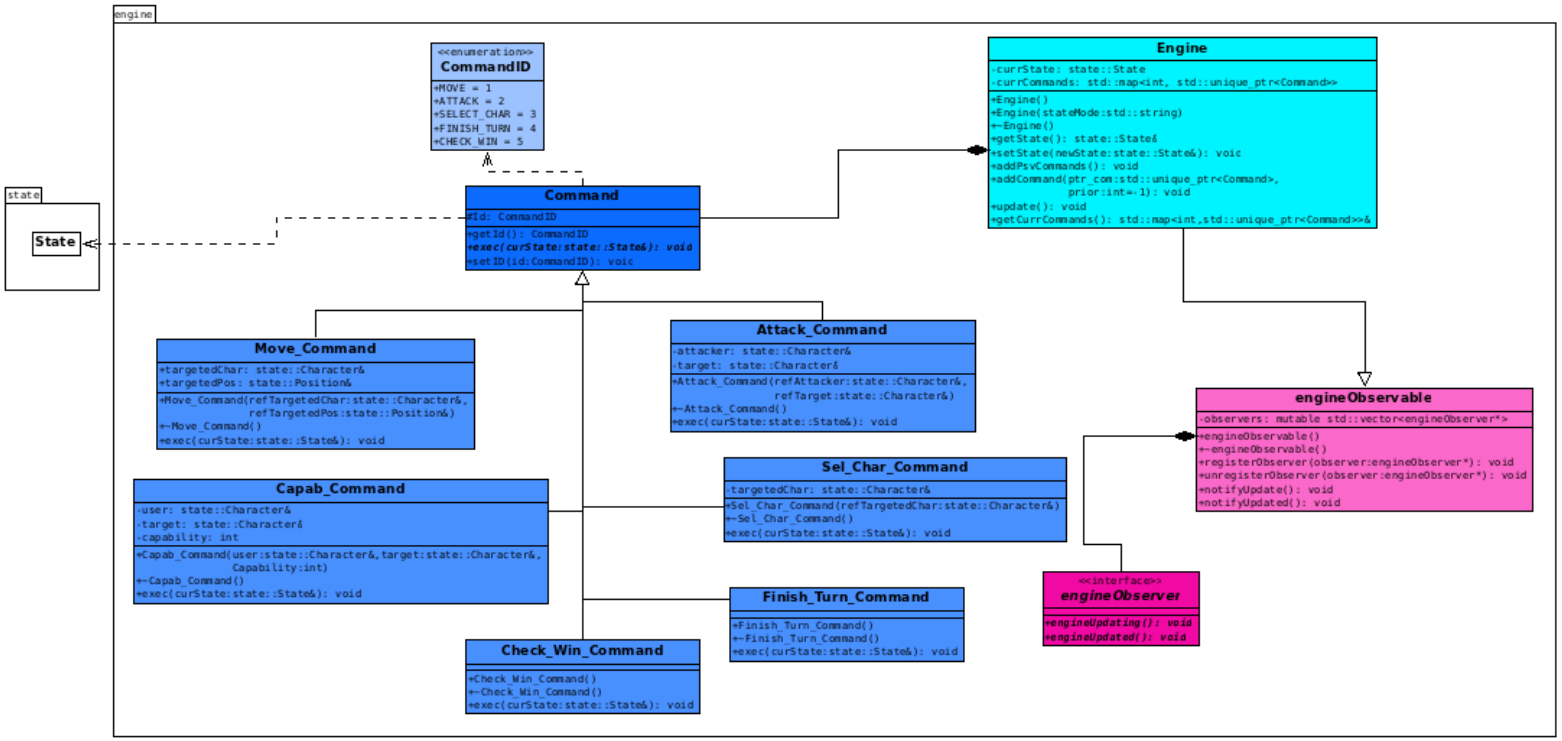


FIGURE 4.3 – Diagramme engine

Chapitre 5

Intelligence artificielle

5.1 Intelligence artificielle aléatoire

La stratégie de notre intelligence aléatoire suit simplement le diagramme de flux suivant :

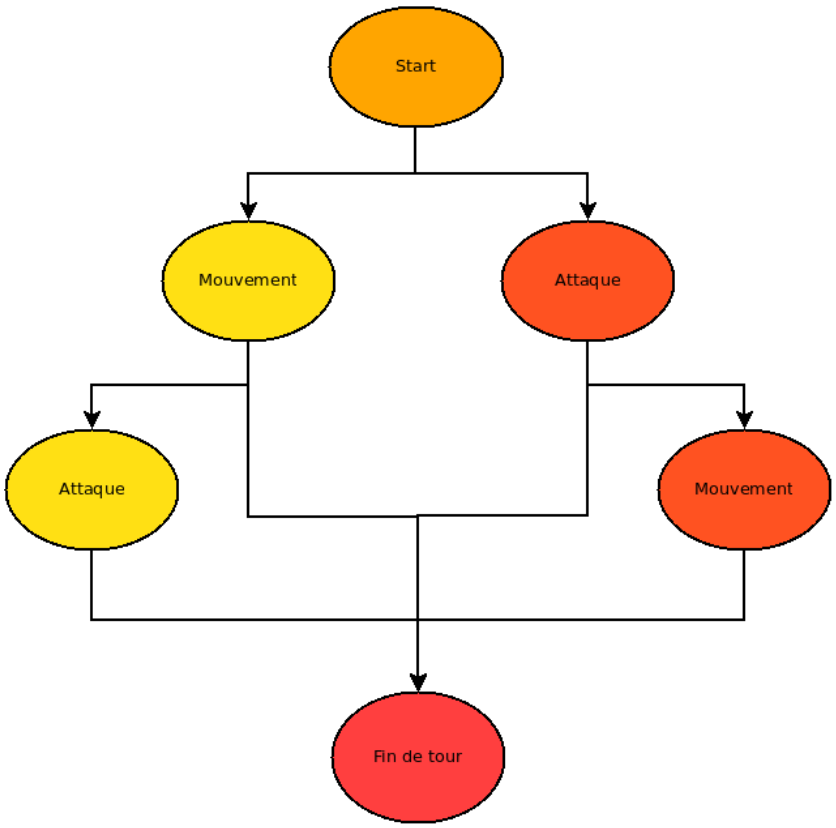


FIGURE 5.1 – Diagramme de flux de RAI

1. L'IA va aléatoirement sélectionner un personnage jouable de son équipe.
2. Elle va ensuite regarder si elle peut effectuer une attaque. Dans ce cas elle choisit aléatoirement une cible parmi celles disponibles.
3. Sinon elle bougera sur une case adjacente de manière répétée jusqu'à ce qu'elle est dépensé tout ses points de mouvement ou qu'une opportunité d'attaque se présente.
4. Si elle a d'abord attaqué elle va ensuite choisir entre finir son tour immédiatement ou se déplacer avec une probabilité uniforme.

On a implémenté deux cas de figure pour l'application de l'intelligence artificielle aléatoire :

1. **RandomAI vs RandomAI** : Les deux joueurs sont des intelligences artificielle aléatoire.
2. **Player vs RandomAI** : Le premier joueur est joué par un utilisateur et le deuxième par une intelligence artificielle aléatoire.

5.2 Intelligence artificielle heuristique

Nous proposons ensuite une IA heuristique qui va permettre d'améliorer le comportement de l'IA random. Ainsi, on donne quelque règles simples pour permettre à l'IA d'être plus « intelligente » et de ne pas seulement compter sur le hasard.
On va orienté les mouvement de l'IA, et choisir judicieusement quel adversaire attaquer.

1. On commence par regarder si un des personnage ennemis est a portée de l'un de nos personnage.
2. Si c'est le cas, on sélectionne le personnage qui est capable d'attaquer l'ennemi avec le moins de Points de Vie.
3. Ce personnage attaque.
4. Si aucun ennemis ne peut être attaqué directement, on sélectionne le personnage allié le plus proche d'un adversaire.

   Pour cela, on utilise un algorithme de parcours en largeur (BFS). Cet algorithme permet à notre IA de trouver le plus court chemin jusqu'à l'ennemi, sachant que sur notre Map, il y a des obstacles, et des trous. C'est la fonction *FindPath* qui effectue cette recherche, en explorant tous les chemins possibles.

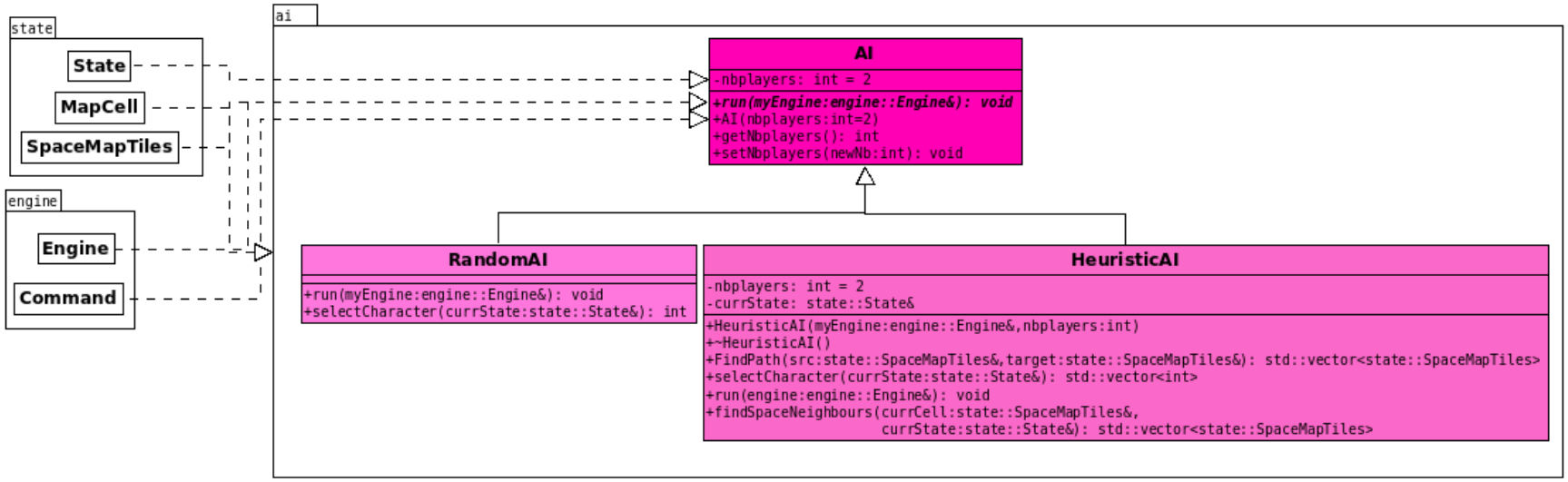


FIGURE 5.2 – Diagramme de classe IA

5.3 Intelligence artificielle avancée

5.3.1 Rollback

Pour faire le rollback,nous avons décidé d'opter pour la première méthode où on clone les états du jeu. Cette méthode nécessitait néanmoins l'implémentation d'autre classe en raison des "unique$_p tr$" $qu'on a instancié dans notre State et qui ne peuvent t recopi$
D'ou on a crée deux classes **CopyState** qui garde une copie du State et **MemoryStates** qui est la mémoire des états sauvegardé.

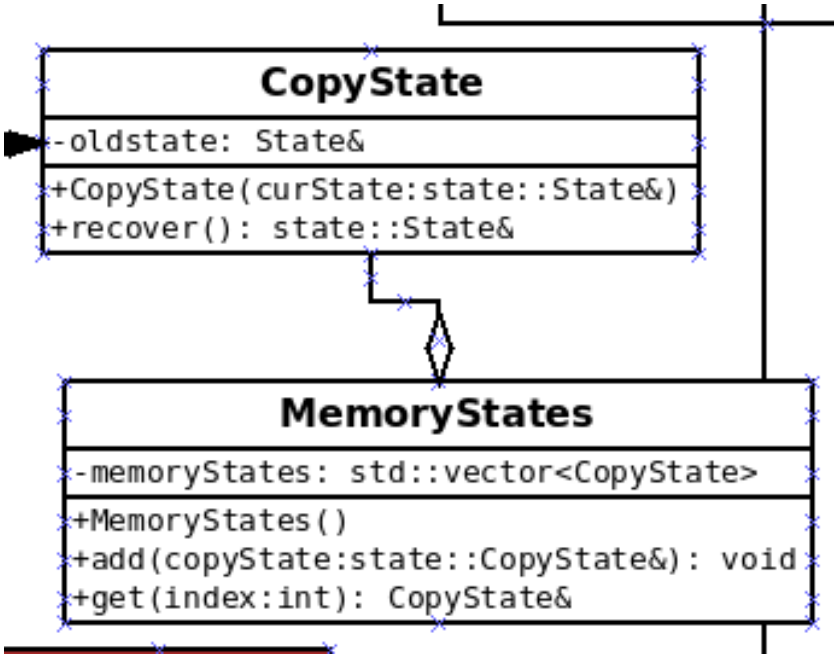


FIGURE 5.3 – Class StateLayer

Ensuite il suffisait d'implémenter deux fonctions **save et load** afin de sauvegarder l'état courant et charger l'éetat sauvegarder dans l'état courant.

5.3.2 Deep AI

Chapitre 6

Modularisation

6.1 Threads

6.1.1 Stratégie : Répartition des Threads

L'objectif de l'utilisation de threads est l'optimisation et parallélisation des calculs. Nous avons donc choisi de séparer les calculs du moteur de jeu, qui se charge de notifier les observeurs pour mettre à jour le rendu, et les calculs lié aux AIs "Heuristic". En effet notre jeu étant au tour par tour, il ne nécessite pas de mettre à jour le rendu aussi souvent qu'un jeu en temps réel. Nous avons implémenté les commandes à exécuter, pour mettre à jour le moteur du jeu, de telle manière que lorsque la méthode "run" de "HeuristicAI" appelle la méthode "update" de "Engine" : la liste de commandes à exécuter est vide, rien ne sera donc exécuté. Ainsi le thread de calcul des AIs n'est pas en charge du moteur ou rendu. Les calculs seront plus rapides

6.1.2 Conception

Nous avons commencé par créé une classe "Client" (voir figure 6.1), qui nous servira de base lors de la modularisation Réseau. L'utilisation d'un client local permet ici de contenir la création des threads ainsi que certaines variables locales, utile lors de la communication entre threads.

La principale méthode de la classe "Client", héritant de "engineObserver", est "run", qui permet d'exécuter les 2 threads mentionnés précédemment. En plus nous avons inclus des booléens pour que le thread de calcul des AIs notifie le thread principal que l'état a été mis à jour.

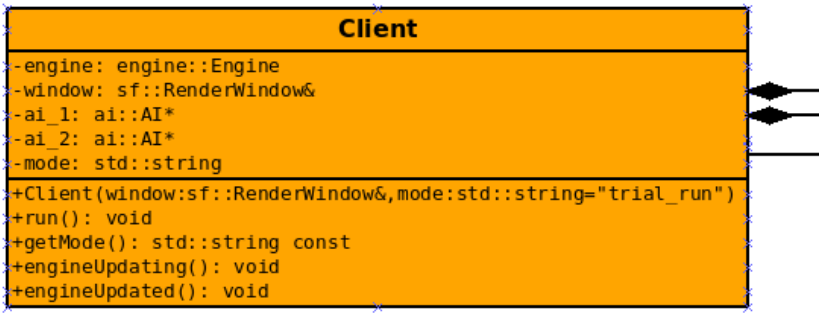


FIGURE 6.1 – Classe Client

6.2 Record

L'objectif de la commande *record* est de sauvegarder toutes les commandes pendant une minute de jeu. Pour cela, on initialise le jeu (c'est-à-dire les joueurs, leurs personnages et les maps). On utilise deux IA heuristiques pour le Test . On modifie notre classe engine.cpp comme indiqué en figure 6.2

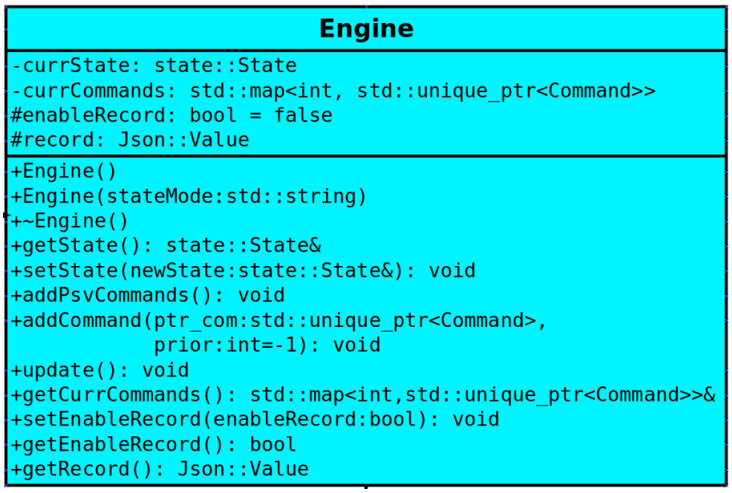


FIGURE 6.2 – Classe Engine

Les étapes de l'enregistrement se font par l'intermédiaire de la libraire json qui va stocker les donnés des commandes sous formt json pour pouvoir les réutiliser lors de la commande play. On crée un fichier *res/Record/replay.txt* et pour chaque commande utilisée dans le jeu, on l'ajoute dans le fichier. En figure 6.3b sont représenté les capture d'écran du terminal et du fichier *replay.txt*.
On a bien une correspondance : le pirate a bougé en case de coordonnées

(2,11). Dans le fichier *replay.txt*, on sauvegarde l'*id*, le *player*, La *target* ainsi que la position en x et en y. Ce fichier va nous permettre de reconstituer la partie dans le play.

6.3 play

L'objectif de la commande play est de récupérer les commandes de *replay.txt* et de les rejouer. On commence par initialiser le jeu, avec les joueurs, les personnages et la map puis on vérifie que le fichier Json *replay.txt* existe bien, sinon on renvoie une erreur. Pour chaque action, on regarde l'ID de la commande :

1. Sélection de personnage ($SELECT\_CHAR$)
2. Choix de bouger ($MOVE$)
3. Choix d'attaquer ($ATTACK$)
4. Choix d'utiliser une capacité ($CAPAB$)

Pour chacune de ses actions, des informations différentes ont été enregistrée dans le fichier Json. Par exemple, lorsque la commande $MOVE$ est appelée, le jeu demande la position en x et en y, et ordonne une commande de déplacement. Ceci permet de rejouer la minute de jeu enregistrée.

```
<<< Record >>>
HAi vs HAIres/record/replay.txt
<<< Début de l'enregistrement >>>
Adding passive commands ...
Executing commands from turn 1

THE SELECTED CHARACTER IS CROOK1

Adding passive commands ...
Executing commands from turn 1
STATE MOVING
Move left5
The character CROOK1 has been moved to (2, 11)
```

(a) Texte associé aux actions dans le terminal

```
<<< ##########################################
 ENREGISTREMENT DANS LE FICHIER REPLAY.TXT
<<< ##########################################
{
        "CommandArray" :
        [
                {
                        "id" : 4,
                        "player_id" : 1,
                        "target_index" : 0
                },
                {
                        "id" : 1,
                        "player_id" : 1,
                        "target_index" : 0,
                        "x" : 2,
                        "y" : 11
                },
```

(b) Texte inscrit dans le fichier replay.txt

FIGURE 6.3 – Test du record

Bibliographie