



# Projet **Royal War**

David KHA, Mazhou HU, Florian  
Palmier, Frederick Omgba Abenah



<b>1 Présentation Générale</b>	<b>4</b>
1.1 Archétype	4
1.2 Règles	4
1.3 Textures	4
1.3.1 Terrain	4
1.3.2 Décor	5
1.3.3 Nature	5
1.3.4 Projectile	5
1.3.5 Bâtiment	6
1.3.6 Personnages	7
1.4 Résultat	8
<b>2 Description et conception des états</b>	<b>9</b>
2.1 Description des états	9
2.1.1 Etats éléments fixes	9
2.1.2 Etats éléments mobiles	10
2.1.3 Etat général	11
2.2 Conception Logicielle	11
2.2.1 La classe "World"	11
2.2.2 La classe "WorldHanlder"	13
2.2.3 La classe "Player"	16
2.2.4 La classe "Manager"	16
2.2.5 La classe "Manageables"	17
2.2.6 La classe "Actor"	19
2.2.7 Le diagramme de classe pour le state	20
<b>3 Rendu : Stratégie et Conception</b>	<b>22</b>

3.1 Stratégie de rendu d'un état	22
3.2 Conception logicielle	26
3.2.1 Classe MainFrame	26
3.2.2 Classe FileHandler	27
<b>4 Règles de changement d'état et moteur du jeu</b>	<b>29</b>
4.1 Changements externes	29
4.1.1 Mise en place des acteurs et actions associées	29
4.1.2 Script associé à chaque carte	32
4.1.3 Commande sélection des acteurs	33
4.1.4 Commande des actions	34
4.2 Changement autonome	35
4.3 Conception logicielle	35
<b>5 Intelligence Artificielle</b>	<b>39</b>
5.1 Stratégie	39
5.1.1 IA aléatoire	39
5.1.2 IA Heuristique	39
5.1.3 IA avancé	40
Rollback	40
Deep_ai	41
5.1.4 Comportement	44
Partie "RandomAI"	44
Partie IA heuristique	44
AI vs AI	45
5.1.5 Tests des IAs	46
5.2 Conception logicielle	46
<b>6 Modularisation</b>	<b>52</b>

6.1 Organisation	52
6.1.1 Répartition sur différents threads	52
6.1.2 Répartition sur différentes machines	53
Fonctionnement global	53
Flux	55
Serveur JS	57
Heartbeat	59

# 1 Présentation Générale

## 1.1 Archétype

L'objectif de ce projet est de créer un jeu comme Advanced War avec nos propres règles.

## 1.2 Règles

- Jeu en 1 vs 1, et contre l'IA
- Chaque personne possède une base et a la possibilité de faire spawn des unités différentes : soldat, lancier, cavalier, archer, dragon, mage et ballista, . Chaque personnage a un coût différent pour le faire spawner, a un nombre de mouvement limité suivant le terrain et a des dégâts différents
- La map est parsemée de villages, chaque village génère 100 or. Plus le joueur contrôle de village, plus il est facile de spawner une armée rapidement.
- Le but est de détruire la base de l'ennemi
- Chaque classe a ses propres caractéristiques

## 1.3 Textures

Le jeu se déroule sur une carte séparée par une grille dont les composants font 16x16 pixels. On trouvera ci-dessous les textures nécessaires au développement de la carte de jeu, qui se trouvent dans le dossier `./res / texture`

### 1.3.1 Terrain



Figure 2- Texture pour les terrains

### 1.3.2 Décor

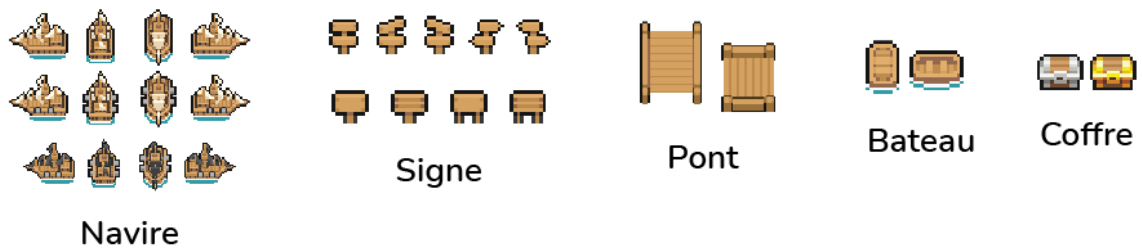


Figure 3 -Texture pour les décors

### 1.3.3 Nature



Figure 4 -Texture pour la nature

### 1.3.4 Projectile



Figure 5 -Texture pour la projectile

### 1.3.5 Bâtiment

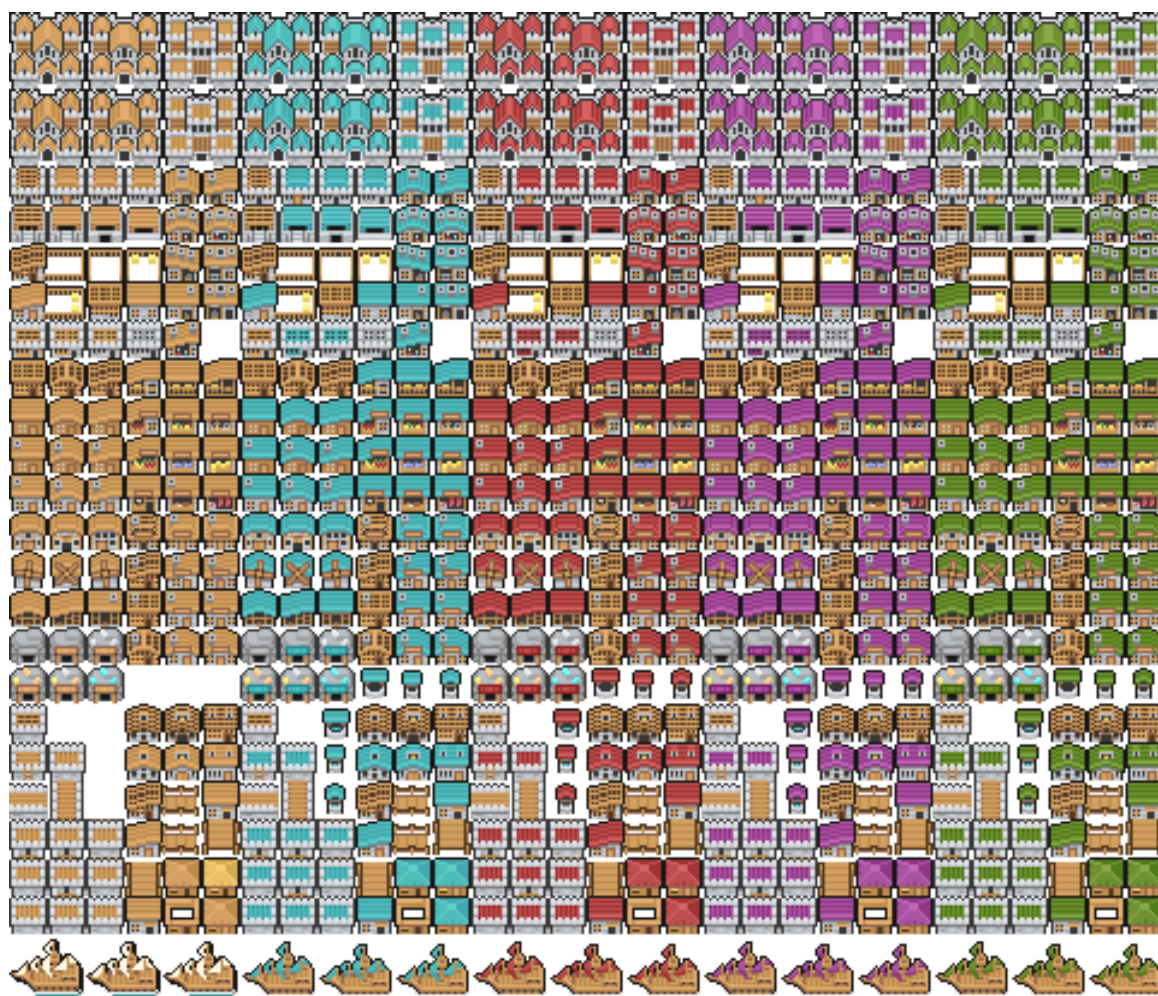


Figure 6 -Texture pour les bâtiments

### 1.3.6 Personnages

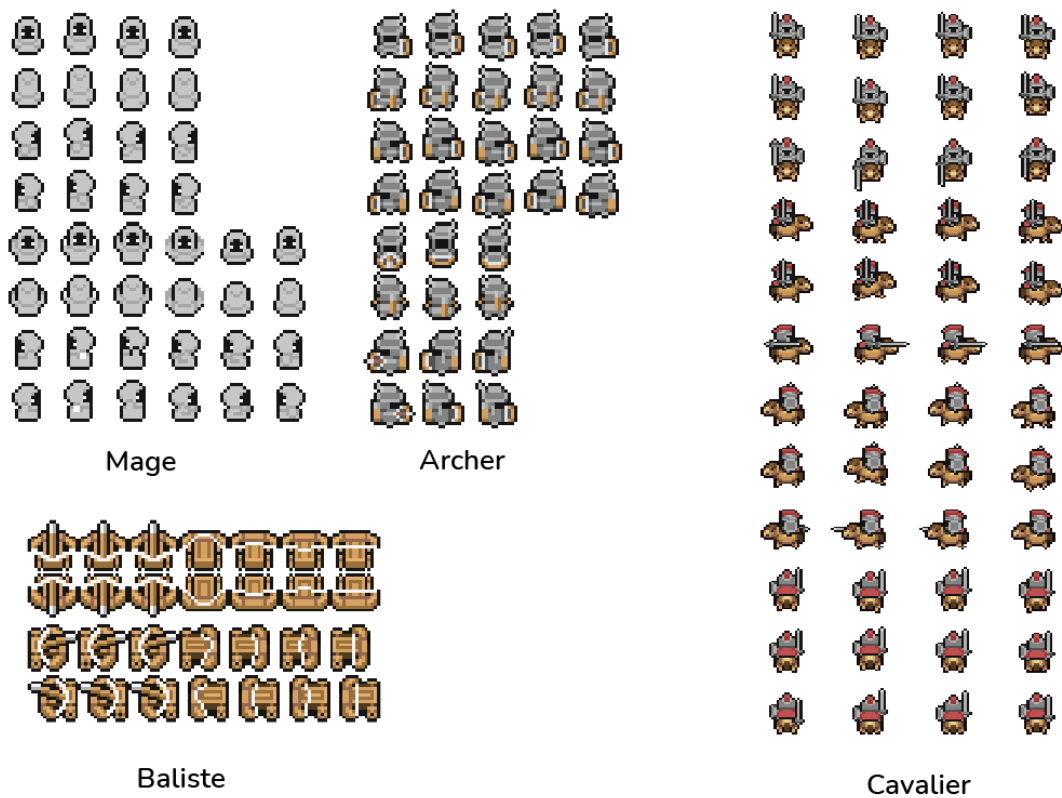


Figure 7.1 -Texture pour les personnage

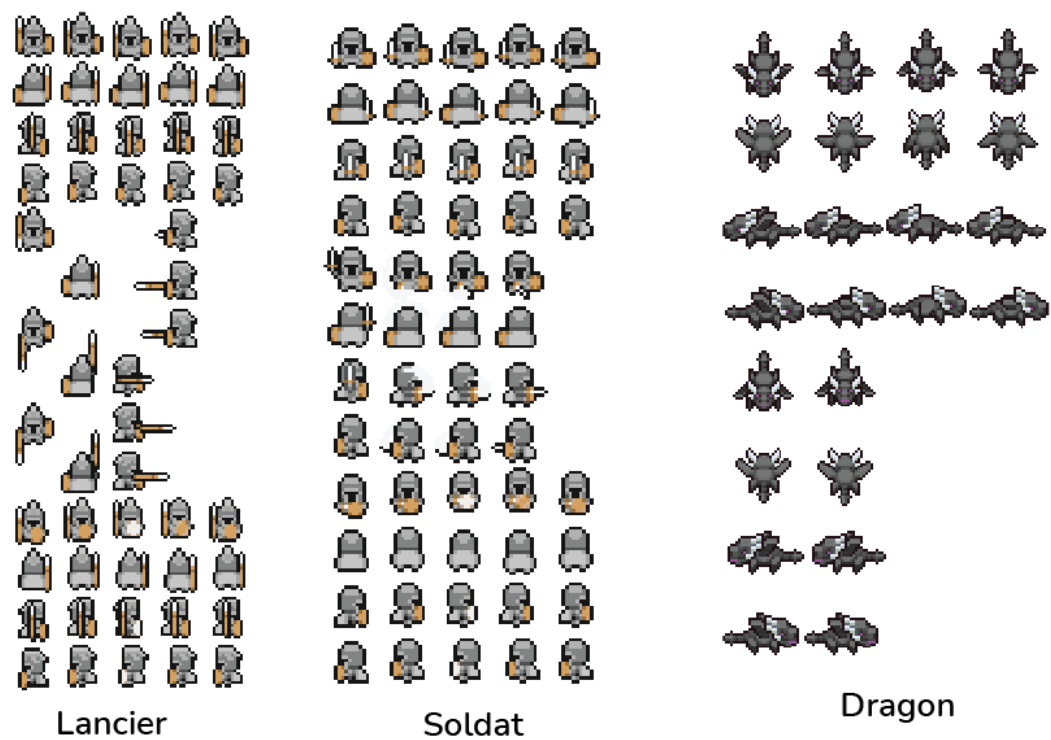


Figure 7.2 -Texture pour les personnages



## 1.4 Résultat

En utilisant ces ressources, on obtiendra une map qui ressemblera à l'image ci-contre.



Figure 8 -l'exemple de la carte créée

## 2 Description et conception des états

### 2.1 Description des états

L'état du jeu est représenté par un ensemble d'éléments fixes, la carte du jeu et les bâtiments sont déjà présents sur la carte. Un ensemble d'éléments mobiles qui sont les différents personnages peuvent se déplacer. Ces éléments ont en commun les caractéristiques suivantes.

- une coordonnées (x, y) dans la grille de la carte
- un identifiant qui permet de les différencier entre eux

#### 2.1.1 Etats éléments fixes

La carte est divisée en plusieurs cases par une grille de 16x16 pixels.

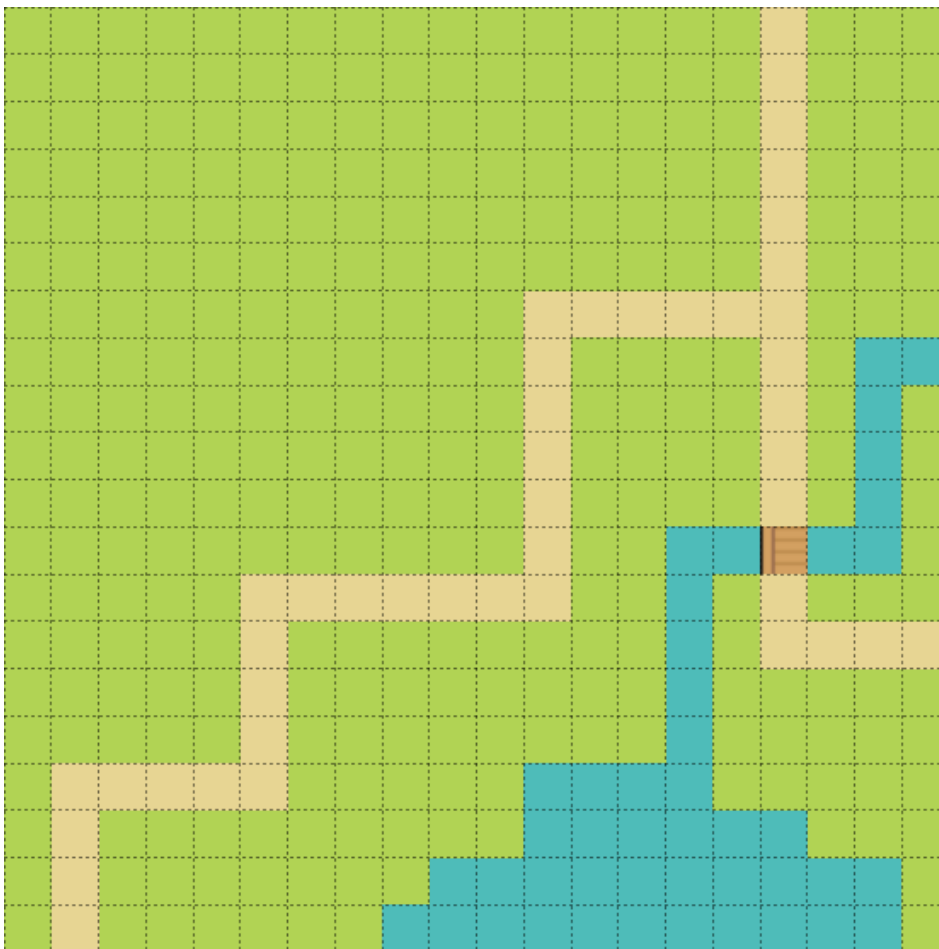


Figure 9 -La carte divisée en plusieurs cases

Il existe plusieurs types de cases, et qui ont différentes caractéristiques.

- Les cases “sol” où les personnages peuvent se déplacer dessus. On y retrouve plusieurs sous catégories : côte, colline, sol, route, forêt et pont. Mis à part le côté esthétique, elles confèrent des bonus aux unités qui les traversent. Par exemple, un personnage se déplace plus vite sur la route mais moins vite en forêt
- Les cases “eau” où les personnages qui ne peuvent pas voler ne peuvent pas se déplacer dessus
- Les cases “bâtiment” qui ont des points de vie et des dégâts et où les personnages ne peuvent pas se déplacer dessus mais peuvent effectuer des actions dessus (attaquer, capturer). On distingue deux types de bâtiment : la base et le reste des bâtiments. La base est l'élément à défendre, elle possède donc plus de points de vie et de dégâts que les autres

### **2.1.2 Etats éléments mobiles**

Les éléments mobiles sont les soldats qu'on peut faire apparaître à chaque tour. Ils possèdent tous les mêmes attributs, mais pas les mêmes valeurs. Les attributs sont les suivants.

- points de vie
- dégâts(points d'attaque)
- peut se déplacer
- peut attaquer
- peut capturer des bâtiments
- peut ignorer le terrain i.e. peut se déplacer sur les cases d'eau
- texture

Ils sont tous contrôlables par le joueur et peuvent uniquement se déplacer case par case. Il s'agit du joueur, de l'adversaire ou de l'IA qui décident de leur déplacement.

### 2.1.3 Etat général

Nous avons ensuite la classe “WorldHanldler” qui gère le nombre de tours, met la fin de jeux, juge quelle joueur gagne etc.

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

- “turn” qui indique le nombre de tours.
- “end” qui indique la fin du jeu.
- “win” qui indique que le joueur a gagné.
- “lose” qui indique que le joueur a perdu.

Ces propriétés seront encodées dans un entier pour déterminer l'état de la partie.

## 2.2 Conception Logicielle

Dans un premier temps nous allons concevoir la classe qui représente tout élément affichable et sélectionnable de notre jeu.

### 2.2.1 La classe “World”

Une classe “World” qui permet d'afficher la carte. Elle contient de nombreux attributs concernant les paramètres de la carte, décrits dans le tableau suivant.. Elle a une relation d'agrégation avec la classe “WorldHandler”.

Tab 1-Le tableau des attributs contenus dans la classe “World”

Attributs	Type	Explication
_Name	String	Nom de la map
_ResPath	String	Chemin d'accès du fichier CSV pour générer la map
_CellSize	Vecteur de dimension deux (matrice)	Représente la dimension d'une cellule dans la map
_CellN	Vecteur de dimensions deux (matrice)	Représente le nombre de cellules en longueur et en largeur de la map
_SolidTexture	Liste de texture	Contient les nombres Textures sur la carte

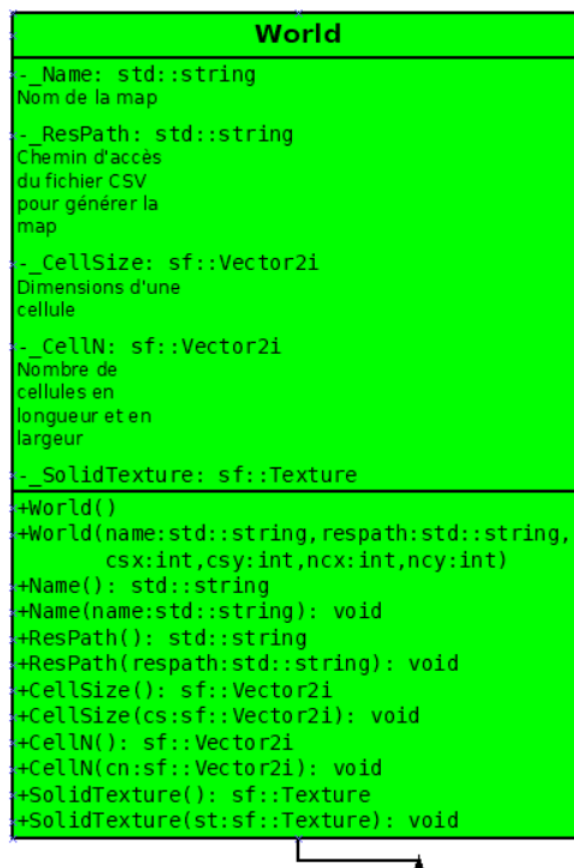


Figure 10-class “World”

### 2.2.2 La classe “WorldHandler”

WorldHandler suit le pattern désigne “Traits”, met à disposition de toutes les classes la possibilité d'exécuter des fonctions lors d'événements.

Lorsque le tour commence, les fonctions contenues dans la liste TurnBeginEvents seront exécutées de même à la fin d'un tour avec TurnEndEvents.

Ainsi une classe peut s'abonner à WorldHandler en définissant les fonctions statiques OnTurnBegin et OnTurnEnd. A chaque début de tour WorldHandler exécutera ces fonctions dans de Routine pour toutes les classes qui sont abonnées.

Le Status de la partie est encodé sur un char :

Si la partie est cours en alors  $\text{Status} = 0x10 + \text{id\_current\_player}$

Si la partie est terminée alors  $\text{Status} = 0x20 + \text{id\_winner}$

Tab 2 -Le tableau des attributs contenus dans la classe “WorldHandler”

Attribut	Type	Explication
CurrentWorld	Pointeur de type de la classe World	Map actuellement chargée
Turn	Entier(int)	Tour de la partie
Players	Liste des classe Player	Liste des joueurs
MyID	Entier( int)	Identifiant du joueur
Instance	Entier (int)	Identifiant des parties en cours qui permet associer les joueurs en chaque partie
Status	Type caractère (char)	Représente le statut du jeu: si le jeu est terminé, si l'on gagne.
TurnBeginEvents	Liste de fonctions	Contient la liste des fonctions qui permettent de manipuler les événements au début des jeux
TurnBeginAsyncEvent	Liste de fonctions	Contient la liste des fonctions asynchrones qui permettent de manipuler les événements au début du jeu
TurnEndEvents	Liste de fonctions	Contient la liste des fonctions qui permettent de manipuler les événements à la fin du jeu
TurnEndAsyncEvent	Liste de fonctions	Contient la liste des fonctions asynchrones qui permet de manipuler les événements à la fin du jeu

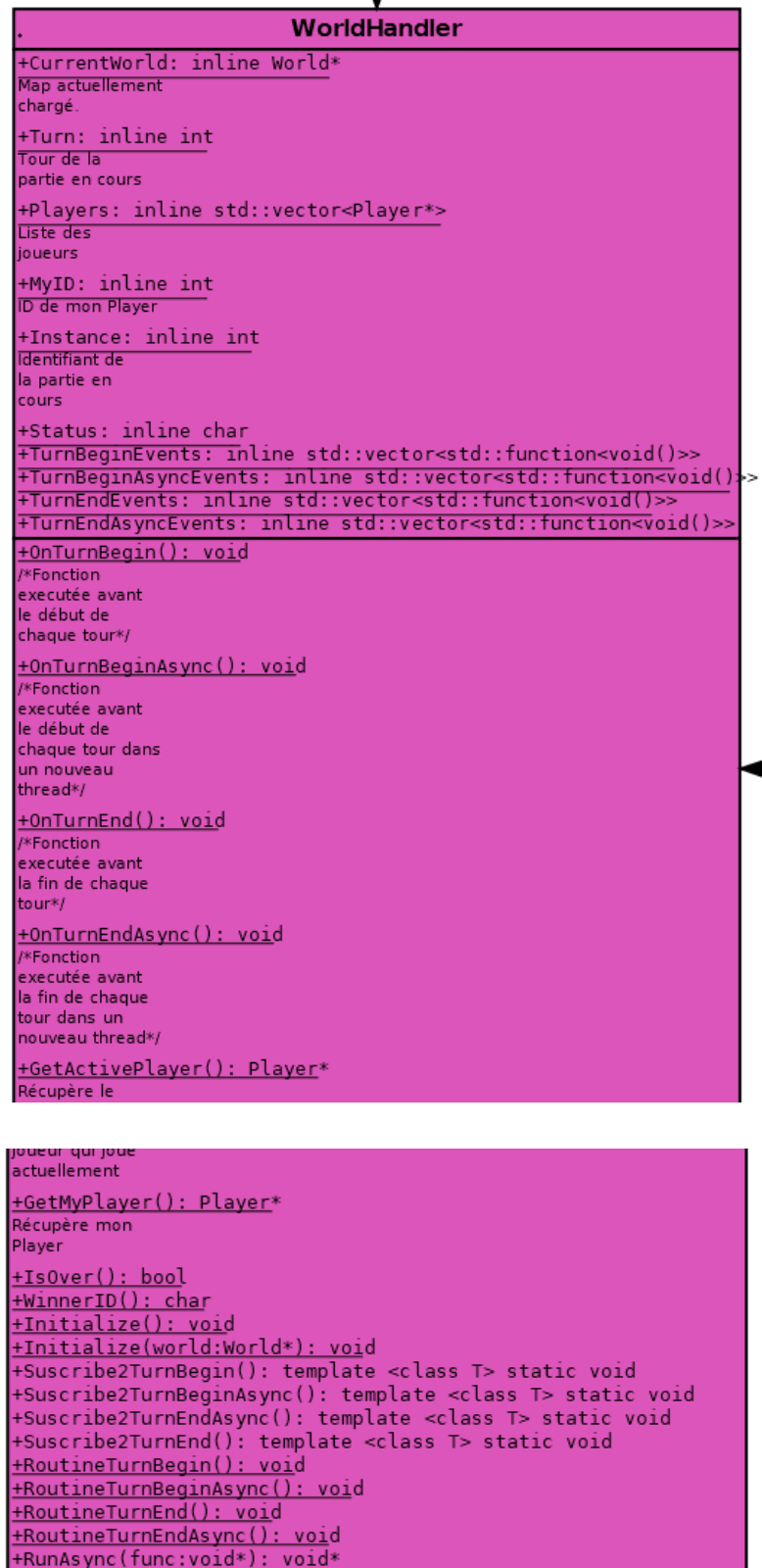


Figure 11: la classe “WorldHandler”



### 2.2.3 La classe “Player”

La classe “Player” contient tous les éléments concernant les joueurs : les noms(\_Name) , les identifiants (\_ID). Elle a une relation d’agrégation avec classe “WorldHandler”.

Tab 3-Le tableau des attributs dans la classe “Player”

Attribut	Type	Fonction
_Name	String	Nom du joueur
_ID	String	Identifiant du joueur

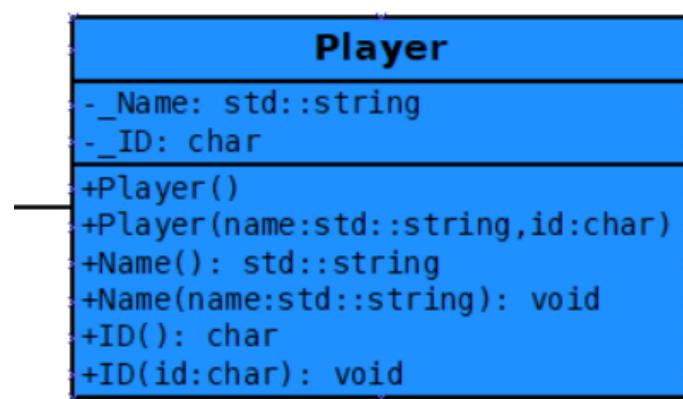


Figure 12 : La class “Player”

### 2.2.4 La classe “Manager”

La classe Manager permet de gérer tous les éléments sur la carte . Il a une relation d’agrégation avec la classe “Worldhandler” . Les Managers mettent à jour les éléments du jeu tout au long de la partie.

Tab 4- Le tableau des attributs contenus dans la classe “Manager”

Attribut	Type	Fonction
_Name	String	Nom du manager
_ID	String	Identifiant du manager
_Elements	Liste de la classe manageable	Listes des 'objets géré par la classe “Manager”
Managers	Vecteur de pointeur de la class “Manager”	Liste référençant tous les manager existants



Figure 13: La class “Manager”

## 2.2.5 La classe “Manageables”

La classe “Manageables” qui contient tous les éléments qui pourraient être manipulés par la classe “Manager”. Les Manageables sont des éléments qui peuvent être affichés et sélectionnés durant la partie. Ceci peuvent être des personnages, des éléments d’interface, les tuiles composants la map.

Tab 5-Le tableau des attributs contenus dans la classe “Manageable”

Attributs	Type	Explication
_Name	String	Nom de Manageable
_ResPath	String	Chemin d'accès vers texture
_ID	Type entier (int)	Identifiant des objets de manageable
_Render	Boolean	Indique si l'objet doit être affiché
_Selected	Boolean	Indique si l'objet est sélectionné
_Texture	Vecteur des pointeurs de texture	Contient les pointeurs des textures pour présenter les objets dans la map
_Sprite	Liste des sprites	Liste des objets associées aux textures
_Position	Vecteurs de dimension deux (matrice)	Représente les coordonnées des objets manipulable sur la map
_Scale	Vecteurs de dimension deux (matrice)	Représente la taille des objets manageables.

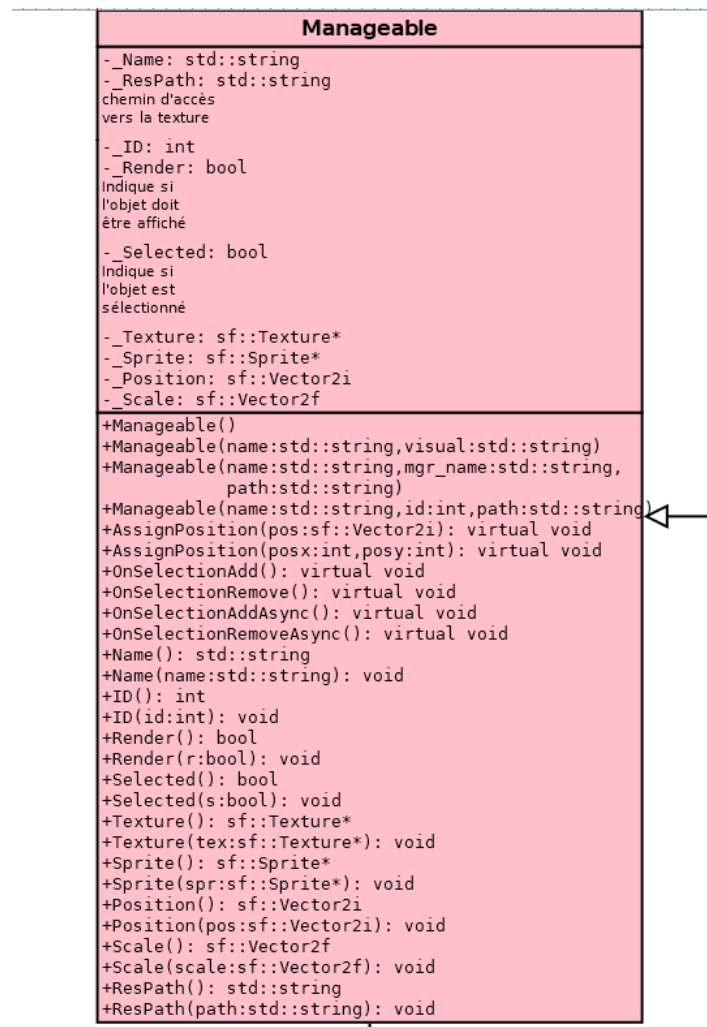


Figure 14 : La classe “Manageable”

## 2.2.6 La classe “Actor”

La classe “Actor” contient toutes les informations des personnages et des bâtiments. Elle reste assez vague pour le moment car les acteurs seront chargés depuis des fichiers.

Tab 6-Le tableau des attributs contenus dans la classe “Actor”

Attribut	Type	Fonction
_HP	Type entier (int)	Point de vie
_DMG	Type entier (int)	Puissance de l'attaque
_DEF	Type entier (int)	Puissance de défense
_AP	Type entier (int)	Point d'action
_MP	Type entier (int)	Point de mouvement

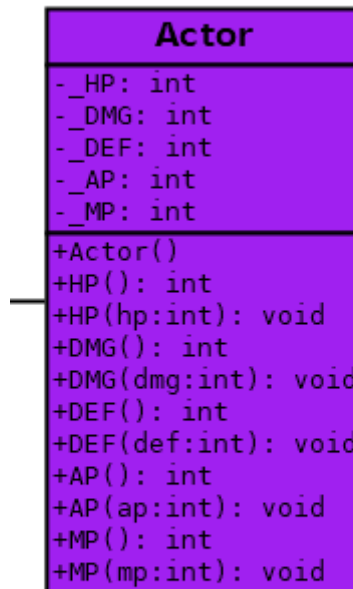


Figure 15: La class “Actor”

### 2.2.7 Le diagramme de classe pour le state

Nous rassemblons toutes les classes citées ci-dessus pour obtenir le diagramme de classe pour l'état state comme la figure 16.

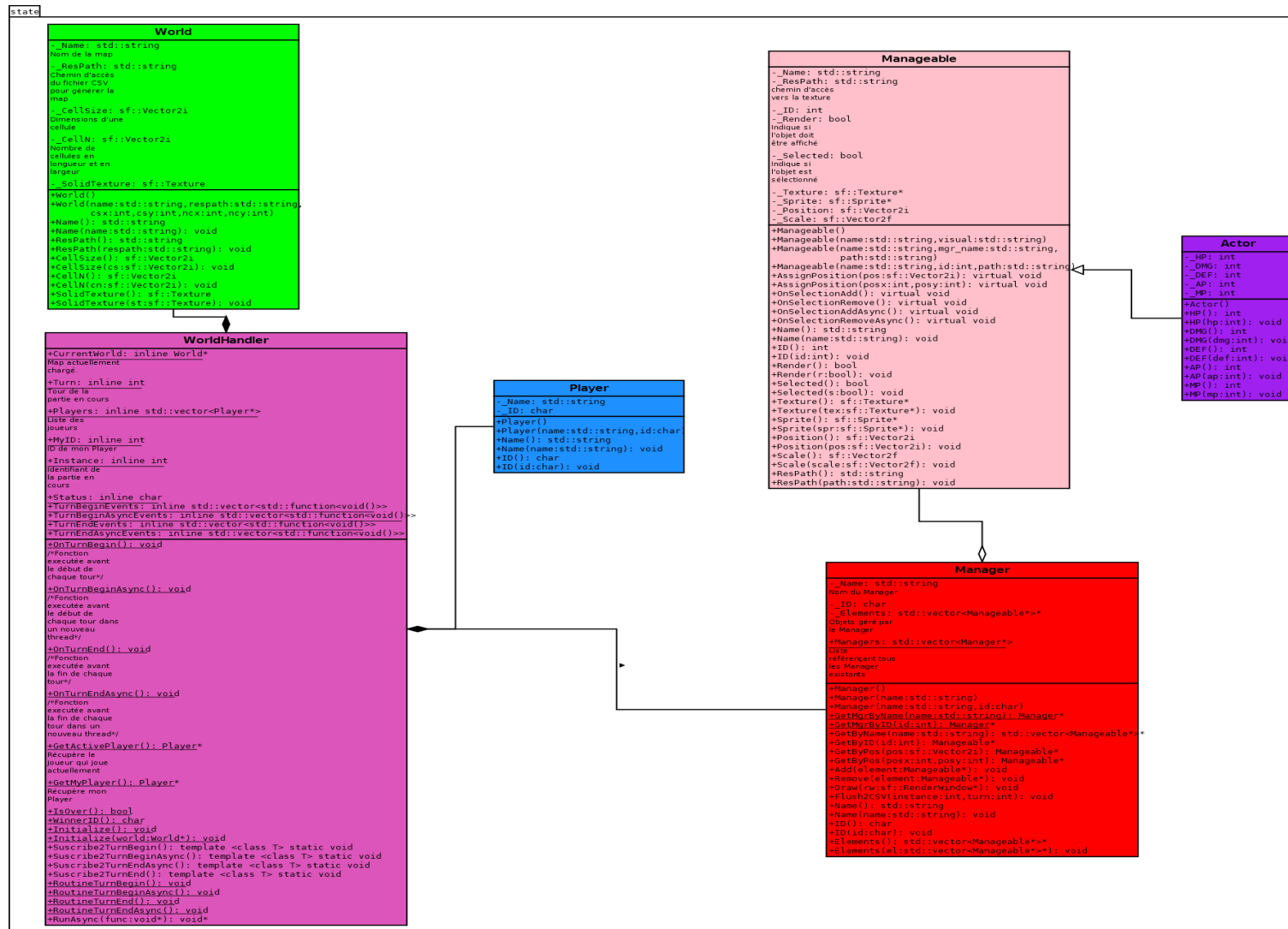


Figure 16: Le diagramme des class

## 3 Rendu : Stratégie et Conception

### 3.1 Stratégie de rendu d'un état

Un état de notre jeu comprend au minimum 3 éléments : la map, les personnages et les informations du jeu (nombre de tours, ...). Selon la situation on peut avoir un autre élément qui est une fenêtre d'actions réalisables par les personnages.

Pour le rendu des états nous utiliserons la bibliothèque SFML ainsi qu'un rendu à l'aide de tuiles. Notre affichage se fait en utilisant plusieurs couches superposées les unes aux autres:

- La couche 0 est une couche qui charge toutes les ressources du jeu. Pour affecter une texture, il suffit d'utiliser un pointeur vers la texture.
- La 1ère couche est la map du jeu
- La 2ème couche est celle où se trouve les personnages ainsi que les bâtiments
- La 3ème couche servira à afficher le menu des actions
- La dernière couche sera celle où sera affiché les informations du jeu

Utiliser plusieurs couches va nous permettre de superposer les éléments et de faire des sélections traversantes nécessaires pour la mise en surbrillance, l'affichage d'un effet visuel ou l'affichage d'un personnage sur une tile. Le nombre de couches est déterminé dans le fichier *Managers.csv*, on peut en ajouter autant qu'on veut.

Pour la construction de la map nous associons un id à un certain sprite afin de construire une tuile. Nous mettons ces id ensuite dans un fichier CSV. Le changement de la map se fait donc en changeant l'id de la tuile dans le fichier CSV.

Les images des tuiles sont mises dans le dossier *res/Texture*. Nous construisons dans le dossier *src/client/table* un fichier *<<ManageablesVisuals.csv >>* où est contenu chaque identité de chaque tuile constituant la map comme dans la figure 17. Chaque identité associe un attribut *MV\_ID* (une chiffre ) qui permet de construire la map.

```
src > client > tables > ManageablesVisuals.csv
1  #MV_NAME,MGR_NAME,MV_ID,PATH,SCALE_X,SCALE_Y
2  BG_TILE_GRASS,ASSET_MGR,0,res/texture/grass.png,0.5,0.5
3  BG_TILE_SAND,ASSET_MGR,3,res/texture/road.png,0.5,0.5
4  BG_TILE_WATER,ASSET_MGR,1,res/texture/water.png,0.5,0.5
5  BG_TILE_BRIDGE,ASSET_MGR,2,res/texture/bridge.png,0.5,0.5
6  ACTOR_KNIGHT,ASSET_MGR,4,res/texture/knight.png,0.036,0.036
7  ACTOR_MAUSOLEUM,ASSET_MGR,5,res/texture/mausoleum.png,0.5,0.5
8  ACTOR_MAUSOLEUM2,ASSET_MGR,6,res/texture/mausoleum2.png,0.5,0.5
9  ACTOR_CYANKEEP,ASSET_MGR,9,res/texture/CyanKeep.png,0.5,0.5
10 ACTOR_REDKEEP,ASSET_MGR,10,res/texture/RedKeep.png,0.5,0.5
11 BG_TILE_STONE,ASSET_MGR,7,res/texture/stone.png,0.5,0.5
12 BG_TILE_GRASS2,ASSET_MGR,8,res/texture/grass2.png,0.5,0.5
13 BG_TILE_CLIFF11,ASSET_MGR,11,res/texture/cliff11.png,0.5,0.5
```

Figure 17 : ManageablesVisuals.csv

Tab7- Attributs des éléments dans le fichier ManageablesVisuals.csv

Attribut	Fonction	Exemple
MV_NAME	Nom de la map	BG_TILE_GRASS
MGR_NAME	Nom qui permet à la classe Manager de manipuler les tuiles	ASSET_MGR
MV_ID	Chiffre permet de distinguer les images tuiles et donc de construire la map	0
PATH	Chemin pour accéder les images des tuiles	res/texture/grass.png
SCALE_X	Coefficient en X qui permet de convertir la taille de l'image	0.5
SCALE_Y	Coefficient en Y qui permet de convertir la taille de l'image	0.5

Nous construisons un fichier csv dans le dossier *src/client/map* qui contient les "MV\_ID" de chaque tuile pour créer la map. Nous avons un exemple dans la figure 18.





CELL_X	Dimension sur l'axe X d'une tile (width)	41
CELL_Y	Dimension sur l'axe Y d'une tile (height)	41
N_CELL_X	Nombre de tuile affichées sur l'abscisse axe X	19
N_CELL_Y	Nombre de tuiles affichées sur l'ordonnée axe Y	19

Afin d'afficher la map, nous devons changer la première ligne du fichier *LaunchArgs.csv* situé dans le dossier *src/client/tables* comme dans la figure 20. Nous mettons à chaque fois le nom de la map après *SCENE* pour afficher la map.


```
src > client > tables >  LaunchArgs.csv
1  SCENE,WORLD_TEST
2  IP_SERVER,127.0.0.1
3  IP_PORT,XXXX
4  FLUSH_PATH,src/client/BoardSnapshot
```

Figure 20: Fichier LaunchArgs.csv

## 3.2 Conception logicielle

Nous utilisons principalement deux classes pour le rendu d'un état.

### 3.2.1 Classe MainFrame

La classe *MainFrame* va créer la fenêtre qui contiendra notre rendu.

Tab 9-Attributs de la classe MainFrame

Attribut	Type	Fonction
Name	String	Nom de la fenêtre
Path	String	Chemin
Width	Int	Largeur de la fenêtre
Height	Int	Longueur de la fenêtre
FrameRate	Int	FPS de la fenêtre

```
MainFrame  
Fenetre du jeu  
- _Name: std::string  
- _Height: int  
- _Width: int  
- _Framerate: int  
- _Window: sf::RenderWindow*  
+MainFrame(name:std::string,height:int,width:int)  
+~MainFrame()  
+Tick(): void  
fonction  
exécutée à  
chaque frame.  
+Draw(): void  
Dessine les  
objets contenus  
dans les  
Managers  
+Start(): void  
Fonction  
exécutée au  
lancement de la  
fenetre  
+Name(n:std::string): void  
+Name(): std::string  
+Height(): int  
+Height(h:int): void  
+Width(): int  
+Width(w:int): void  
+Framerate(): int  
+Framerate(f:int): void  
+InitWorld(): void  
+InitActors(): void  
+Window(win:sf::RenderWindow*): void  
+Window(): sf::RenderWindow*
```

Figure 21: La classe MainFrame

Tick() est une fonction exécutée à chaque frame.

Draw() est une fonction qui permet de dessiner les objets contenus dans les managers.

Start() est une fonction exécutée une fois au lancement de la fenêtre.

InitWorld() est une fonction qui permet d'initialiser la carte.

InitActors() est une fonction qui permet d'initialiser les acteurs.

Les restes des fonctions sont des "getter" et "setter" pour les attributs.

### 3.2.2 Classe FileHandler

La classe FileHandler est celle qui va lire nos fichiers CSV et ainsi créer nos éléments pour pouvoir les afficher.

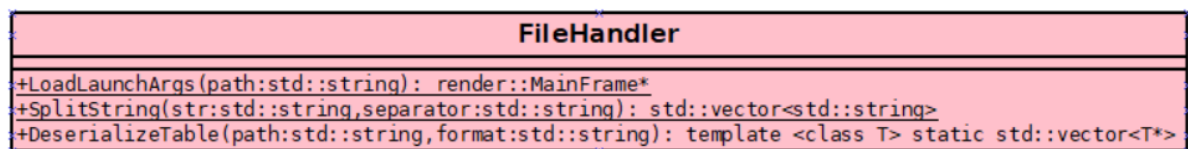


Figure 22: Classe FileHandler.

SplitsString() est une fonction qui permet de séparer les chaînes.

LoadLaunchArgs() est une fonction qui lance une fenêtre selon les paramètres du fichier de configuration *LaunchArgs.csv*.

DeserializeTable() est une fonction pour désérialiser les lignes d'un fichier dans une liste d'objet de la classe spécifiée en paramètre de template (exemple `DeserializeTable<Actor>("Actors.csv","CSV")`). Cette fonction nous permet de charger les tables aisément.

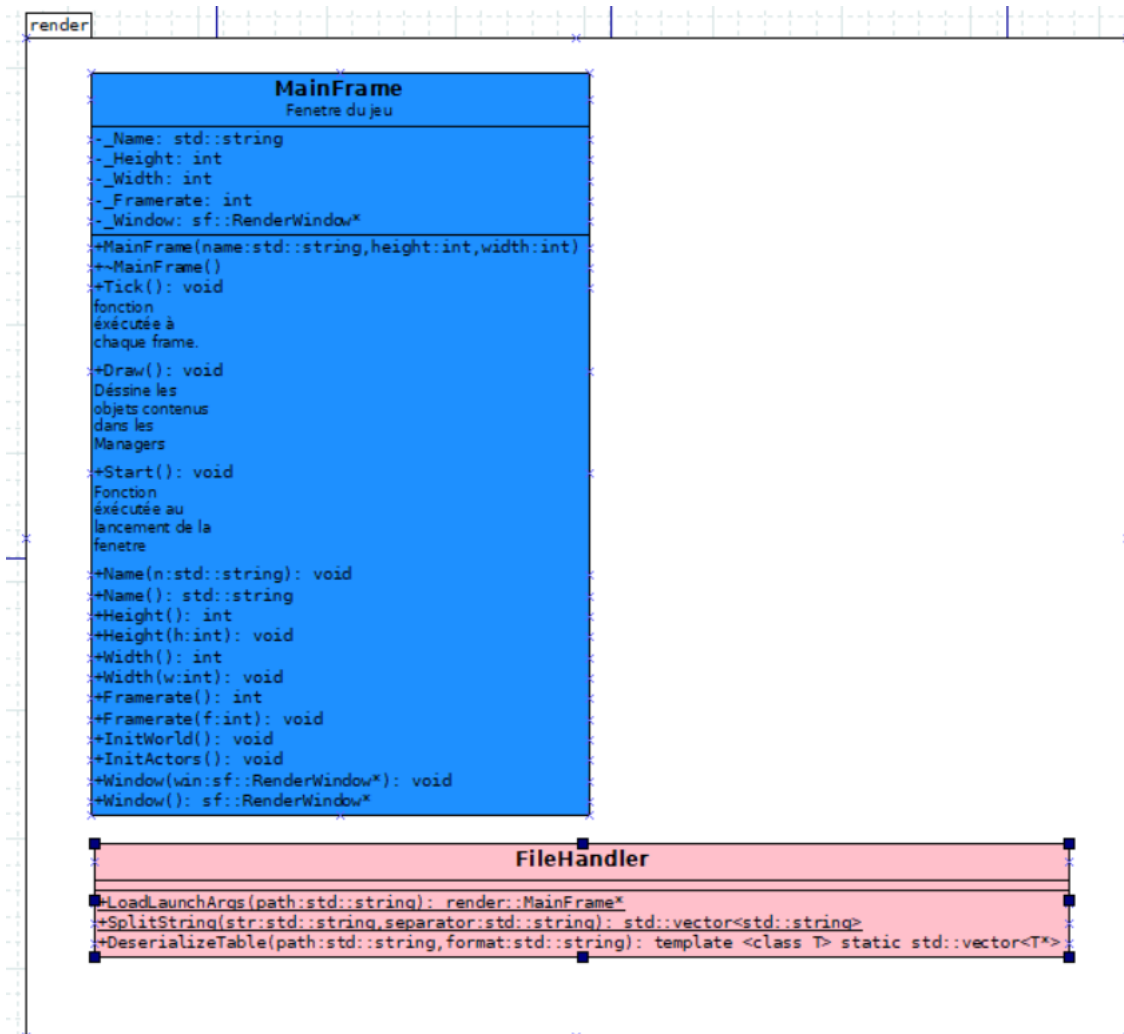


Figure 23: Diagramme du namespace rende

## 4 Règles de changement d'état et moteur du jeu

### 4.1 Changements externes

Les changements extérieurs sont provoqués par des commandes, en particulier la pression d'une touche de clavier ou le clic d'une souris. Nous distinguons 2 types de commandes extérieures :

- Commande de sélection des acteurs.
- Commande des actions

#### 4.1.1 Mise en place des acteurs et actions associées

Tous les personnages sont définis sous le fichier *Actors.csv* dans le dossier *src/client/tables*, comme montré ci-dessous.

```
1  #ACTOR_NAME,VISUALS,ACTOR_HP,ACTOR_DMG,ACTOR_DEF,ACTOR_ACT_PTS,ACTOR_MOV_P
2  BUILDING_MAUSOLEUM,ACTOR_MAUSOLEUM,100,30,60,2,0,STD_ATTACK;STD_MOVE
3  BUILDING_MAUSOLEUM2,ACTOR_MAUSOLEUM2,100,30,60,2,0,STD_ATTACK;STD_MOVE
4  BUILDING_CYANRESSOURCE,ACTOR_CYANRESSOURCE,100,40,50,2,0,STD_ATTACK;STD_MO
5  BUILDING_REDRESSOURCE,ACTOR_REDRESSOURCE,100,40,50,2,0,STD_ATTACK;STD_MOVE
6  BUILDING_CYANKEEP,ACTOR_CYANKEEP,100,20,70,2,0,STD_ATTACK;STD_MOVE
7  BUILDING_REDKEEP,ACTOR_REDKEEP,100,20,70,2,0,STD_ATTACK;STD_MOVE
8  HERO_CYANBOWMAN,ACTOR_CYANBOWMAN,100,55,25,3,3,STD_ATTACK;STD_MOVE
9  HERO_REDBOWMAN,ACTOR_REDBOWMAN,100,55,25,3,4,STD_ATTACK;STD_MOVE
10 HERO_CYANMAGE,ACTOR_CYANMAGE,100,70,35,5,3,STD_ATTACK;STD_MOVE
11 HERO_REDMAGE,ACTOR_REDMAGE,100,70,35,5,3,STD_ATTACK;STD_MOVE
12 HERO_REDSPEARMAN,ACTOR_REDSPEARMAN,100,80,35,5,6,STD_ATTACK;STD_MOVE
13 HERO_CYANSPEARMAN,ACTOR_CYANSPEARMAN,100,80,35,5,6,STD_ATTACK;STD_MOVE
14 HERO_CYANKNIGHT,ACTOR_CYANKNIGHT,150,80,45,6,8,STD_ATTACK;STD_MOVE
15 HERO_REDKNIGHT,ACTOR_REDKNIGHT,150,80,45,6,8,STD_ATTACK;STD_MOVE
16 HERO_CYANDRAGON,ACTOR_CYANDRAGON,200,90,65,5,10,STD_ATTACK;STD_MOVE
17 HERO_REDDRAGON,ACTOR_REDDRAGON,200,90,65,5,10,STD_ATTACK;STD_MOVE
```

Figure 24 : Fichier "Actors.csv"

Tab 10-Attributs des éléments du fichier Actors.csv

Attribut	Type	Fonction
ACTOR_NAME	String	Nom de l'acteur
VISUALS	String	Nom de sa texture qui a été déclaré au préalable dans le fichier ManageablesVisuals.csv
ACTOR_HP	Int	Point de vie de l'acteur
ACTOR_DMG	Int	Point d'attaque de l'acteur
ACTOR_DEF	Int	Point de défense de l'acteur
ACTOR_ACT_PTS	Int	Point d'action de l'acteur
ACTOR_MOV_PTS	Int	Point de mouvement de l'acteur
ACTIONS	String	Scripts des actions à associer avec cet acteur, par exemple un script pour attaquer, se défendre ou spawn des personnages <b>Chaque script différent doit être séparé par une virgule</b>

Concernant les scripts des actions, ils sont déclarés dans le fichier *Actions.csv*, comme ci-dessous.

```
src > client > tables > Actions.csv
1  #ACTION_NAME,OPCODE,ACTION_POINTS,MOVEMENT_POINTS,PATTERN,NET_CMD
2  STD_ATTACK,0x1000,1,0,PATTERN_DIAMOND4,NET_CMD_ATTACK
3  STD_MOVE,0x2000,0,1,PATTERN_DIAMOND2,NET_CMD_MOVE
4  STD_INVOKE_CYAN,0x3000,1,0,PATTERN_CROSS,NET_CMD_INVOKE_CYAN
5  STD_INVOKE_RED,0x3001,1,0,PATTERN_CROSS,NET_CMD_INVOKE_RED
```

Figure 25 : Fichier "Actions.csv"

Tab 11-Attributs des éléments du fichier Actions.csv

Attribut	Type	Fonction
ACTION_NAME	String	Nom de l'action
OPCODE	String	Id de la l'action
ACTION_POINTS	Int	Point d'action que l'action va coûter
MOVEMENT_POINTS	Int	Point de mouvement que l'action va consommer
PATTERN	Int	Pattern associé à l'action
NET_CMD	Int	Nom de la commande à gérer par le moteur

L'attribut OPCODE est l'ID de l'action. Le bit de poids fort sert à identifier le type d'action : on distingue trois principaux types, se déplacer, attaquer et invoquer. Le bit de poids faible va distinguer la sous catégorie de l'action, comme par exemple l'action invoquer pour deux acteurs différents.

L'attribut NET\_COMMANDE sert à envoyer au moteur le nom de la fonction à traiter.

Chaque joueur va avoir au début de chaque tour un nombre de points d'actions limités, afin de ralentir la partie et de bien choisir les actions à effectuer. L'attribut MOVEMENT\_POINTS va uniquement servir pour l'action de déplacement, puisqu'il indique jusqu'à où il est possible de se déplacer.

L'attribut PATTERN est référencé dans la fichier *Pattern.csv*, où on va déclarer les pattern des mouvements, comme ci-dessous.


```
src > client > tables >  Patterns.csv
1  #PATTERN_NAME,PATTERN_PATH,SIZE_X,SIZE_Y
2  PATTERN_CROSS,src/client/actions_patterns/PatternCrossSimple.csv,3,3
3  PATTERN_DIAMOND4,src/client/actions_patterns/PatternDiamond4.csv,9,9
4  PATTERN_DIAMOND2,src/client/actions_patterns/PatternDiamond2.csv,5,5
```

Figure 26 : Fichier "Pattern.csv"



Tous les différents pattern se trouvent dans le dossier *src/client/actions\_patterns/*. Par exemple, prenons le premier pattern. Le fichier csv associé est le suivant.


```
src > client > actions_patterns >  PatternCrossSimple.csv
1    0,1,0
2    1,A,1
3    0,1,0
```

Figure 27 : Fichier "Pattern.csv"

On va déclarer la taille du motif, qui est du 3x3, et dessiner littéralement le motif. Les 1 correspondent aux cases associées au motif, et les 0 ceux qui ne le sont pas.

Ce système reste flexible, ce qui nous permet de déclarer autant d'actions avec des patterns associés que l'on souhaite, sans pour autant devoir redéfinir le dia à chaque modification.

#### 4.1.2 Script associé à chaque carte

Nous définissons ensuite la position des personnages et les fonctions associées dans différents fichiers scripts. Par exemple, nous avons créé un fichier script *Layer\_Actor.bhv* pour positionner les personnages. Nous définissons ensuite où se trouve ce script dans le fichier *Scripts.csv* du dossier *src/client/tables* comme montré ci-dessous.


```
src > client > tables >  Scripts.csv
1    #SCRIPT_NAME,PATH
2    #IS_BHV,src/client/scripts/InputSourceKeyboard.bhv
3    STD_WORLD,src/client/scripts/STD_WORLD.bhv
4    STD_SCENARIO,src/client/scripts/StandardScenario.bhv
5    LAYER_ACTOR,src/client/scripts/Layer_Actor.bhv
6    REGLE,src/client/scripts/Regle.bhv
7    SCRIPT_ACTIONS,src/client/scripts/Script_Actions.bhv
8    HOME,src/client/scripts/Home.bhv
9
```

Figure 28 : Fichier Scripts.csv

Nous associons ensuite un script à la map où nous voulons ajouter les personnages. Pour cela, nous ajoutons le nom du script déclaré dans le fichier *Scripts.csv* dans la définition des maps dans le fichier *Worlds.csv* comme ci-dessous.

```

Worlds.csv X
src > client > tables > Worlds.csv
1  #WORLD_NAME,MAP_PATH,CELL_X,CELL_Y,N_CELL_X,N_CELL_Y,SCRIPT
2  STD_WORLD,src/client/maps/DefaultWorld.csv,40,40,19,19,STD_WORLD
3  O_WORLD,src/client/maps/OtherWorld.csv,40,40,19,19,0
4  EVERGREEN,src/client/maps/Evergreen.csv,40,40,10,10,0
5  POOL,src/client/maps/Pool.csv,40,40,11,10,0
6  WORLD_TEST,src/client/maps/WorldTest.csv,40,40,30,19,LAYER_ACTOR
7  BEACH,src/client/maps/Beach.csv,40,40,26,12,LAYER_ACTOR
8  HOME,src/client/maps/Home.csv,256,64,5,9,0

```

Figure 29 : Fichier “Worlds.csv”.

Par exemple, nous associons à la map BEACH le script LAYER\_ACTOR.

### 4.1.3 Commande sélection des acteurs

Au début de chaque tour, tous les acteurs, que ce soit un personnage ou un bâtiment, possèdent un statut disponible. Le joueur sélectionne un personnage avec le curseur que nous pouvons déplacer avec la souris. Une fois un acteur sélectionné, il est ajouté comme [caster].

Par exemple, le dragon bleu est sélectionné.

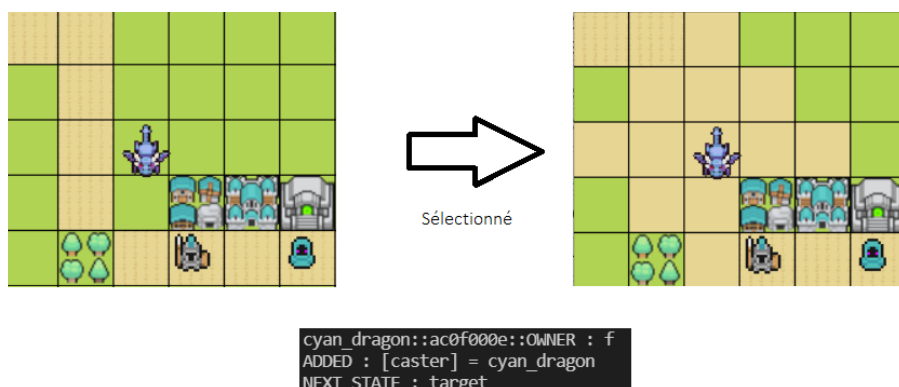


Figure 30 : le démarche de sélection

#### 4.1.4 Commande des actions

Les acteurs ont différentes actions possibles :

- Les personnages ont des actions de déplacement , d'attaque.
- Les bâtiments ont des actions de récolte et peuvent invoquer des personnages.

Toutes ces méthodes seront définies dans le fichier *Script\_Actions.bhv* dans le dossier *src/client/scripts* comme ci-contre.

```
FUNCTION Move
CALL FUNC::CheckPosition 3 ARGS:1 ARGS:2
NEQUAL RET:CheckPosition 0 JMP:3
LTE ACTOR_MGR::ARGS:0.MP 0 JMP:2
PROPERTY ACTOR_MGR::ARGS:0.X ARGS:1
PROPERTY ACTOR_MGR::ARGS:0.Y ARGS:2
END FUNCTION

FUNCTION Attack
CALL FUNC::CheckPosition 3 ARGS:1 ARGS:2
EQUAL RET:CheckPosition 0 JMP:7
INT dmg = 0
INT hp = 0
ADD dmg ACTOR_MGR::ARGS:0.DMG
ADD hp ACTOR_MGR::RET:CheckPosition.HP
SUB dmg ACTOR_MGR::RET:CheckPosition.DEF
SUB hp dmg
PROPERTY ACTOR_MGR::RET:CheckPosition.HP hp
PRINT_INT hp
END FUNCTION

FUNCTION Invoke_cyan
CALL FUNC::CheckPosition 3 ARGS:1 ARGS:2
NEQUAL RET:CheckPosition 0 JMP:5
INT ap = 100
SUB ap ACTOR_MGR::ARGS:0.AP
GT ap 0 JMP:2
ACTOR cyan_base BUILDING_CYANRESSOURCE ACTOR_MGR ARGS:1 ARGS:2
END FUNCTION
```

Figure 31 : Regle.bhv

Notre principale devise durant le développement du jeu est la flexibilité du code. En utilisant un script, il est possible de changer les attributs des personnages, actions ou fonctionnalités sans changer directement le code en C++. Si on souhaite modifier les règles du jeu ou ajouter de nouvelles fonctionnalités, par exemple un coup critique, on peut le faire très simplement en modifiant le script.

**Remarque :** Les acteurs ne peuvent pas faire leurs réactions si ce n'est pas son tour. Nous pourrions utiliser la touche "T" pour changer le tour.

## 4.2 Changement autonome

Les changements autonomes sont appliqués à chaque création ou mise à jour d'un état, après les changements extérieurs. Ils sont exécutés dans l'ordre suivant :

1. Si l'un des bâtiment de type base (Keep) est détruit, c'est à dire le nombre de HP du bâtiment est égale à 0, on affiche "GAME OVER"
2. Nous mettons à jour les attributs des acteurs de joueurs en fonction des règles du jeu
3. Nous appliquons les règles de déplacement du joueur et l'IA
4. Si le joueur a gagné, on affiche "GAME WON"

## 4.3 Conception logicielle

La diagramme des classes pour le moteur se constitue des classes suivantes :

- La classe **SelectionHandler** : gère la sélection, lors d'une action, SelectionHandler remplit une commande avec les paramètres souhaités. Il verrouille la couche de sélection.

```

SelectionHandler
+Selection: inline std::vector<state::Manageable**>
+FilteredSelection: inline std::map<std::string, state::Actor*>
+SelectionMask: inline std::vector<std::string>
+SelectionState: inline std::string
+Behaviour: inline Script*
+Packet: inline std::pair<std::string, std::string>
+Add(m: state::Manageable**): void
+Remove(m: state::Manageable**): void
+OnMouseLeft(x: int, y: int): void
+OnMouseRight(x: int, y: int): void
+Trash(): void
+ProcessSelection(m: state::Manageable**): int
+PrintSelection(): void
+NetFormat(): std::string
+Flush(packet: std::string): void
+ChangeAction(new action: std::string): void

```

Figure 32: La classe SelectionHandler

- La classe **InputHandler** : cette classe propose des événements (souris + clavier), la classe Player est abonnée à cette classe

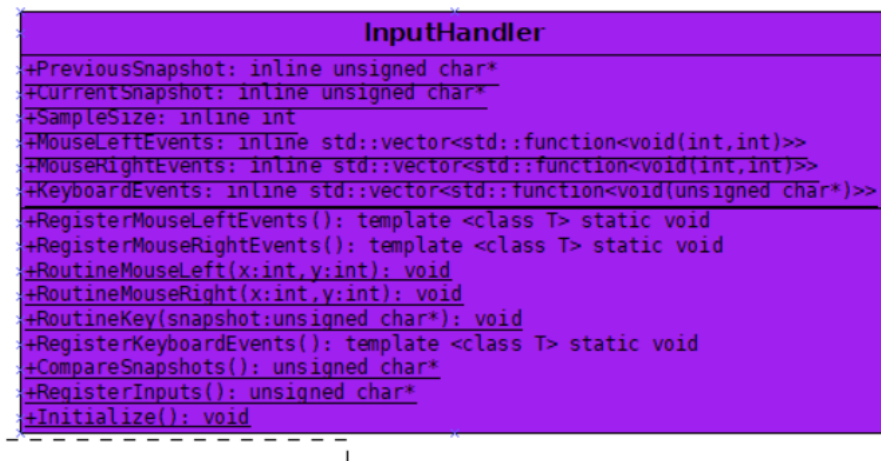


Figure 33: La classe InputHandler

- La classe **pattern** gère le motif, il génère la position des tiles à afficher.

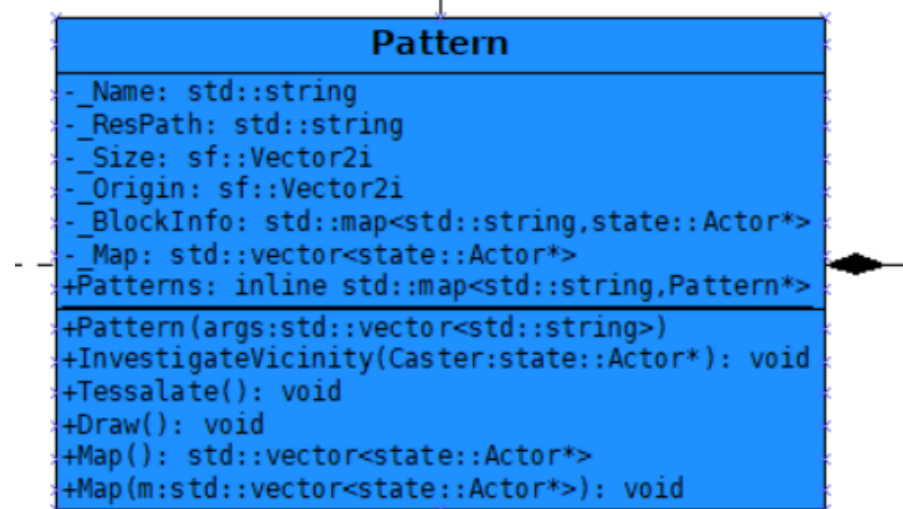


Figure 34: La classe Pattern

- La classe **NetCommand** : format des commandes à envoyer au moteur (dans le futur serveur) pour qu'il exécute une fonction située dans un script.

```

#NET_MSG,FORMAT
NET_CMD_MOVE,Move:$X;$X;$X,caster.ID;target.X;target.Y
NET_CMD_ATTACK,Attack:$X;$X;$X,caster.ID;target.X;target.Y
NET_CMD_INVOKE_CYAN,Invoke_cyan:$X;$X;$X,caster.ID;target.X;target.Y
NET_CMD_INVOKE_RED,Invoke_red:$X;$X;$X,caster.ID;target.X;target.Y
NET_CMD_TURN_FINISHED,OnTurnEnd,NONE

```

Figure 35: Le fichierNetMessage.csv

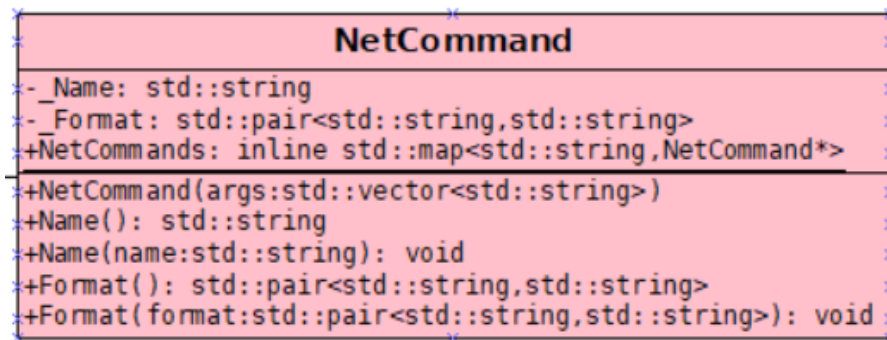


Figure 36: La classe NetCommand

- La classe **Action** : classe gérant les actions, par le biais d'un objet NetCommand , une action est liée à une fonction contenu dans un script.



Figure 37: La classe Action



# 5 Intelligence Artificielle

## 5.1 Stratégie

### 5.1.1 IA aléatoire

Notre IA contrôle l'équipe rouge. Nous contrôlons l'équipe bleue. Au début de la partie, nous choisissons notre action. Une fois que nous terminons notre partie, nous appuyons sur la touche "T". Cela va passer le tour et va permettre à IA de jouer. L'IA va choisir entre différentes actions : se déplacer ou attaquer pour chaque personnage. Il ne peut attaquer que lorsqu'il y a des ennemis dans l'intervalle où il peut l'attaquer.

**Remarque :** Nous avons changé une règle : quand un acteur d'une équipe est mort, le jeu est terminé. La première équipe à réussir à tuer l'acteur en premier gagne.

### 5.1.2 IA Heuristique

Nous avons changé notre règle initiale : la première équipe qui réussit à détruire un bâtiment de l'autre équipe a gagné. Notre IA heuristique va faire avancer notre soldat vers les soldats ennemis. Lorsque les soldats tuent les soldats adverses, il va bouger vers la base et l'attaquer. Une fois le bâtiment détruit, c'est la victoire.

Notre IA heuristique va réfléchir à chaque fois au chemin le plus court vers les acteurs de l'autre équipe, soit un personnage, soit un bâtiment. Pour l'instant, nous avons décidé que chaque personnage va se diriger vers le personnage ennemi le plus proche. Pour cela, notre IA va choisir les positions possibles (le chemin) pour minimiser la distance entre le personnage allié et un acteur ennemi avec la formule suivante :

$$C^* = \underset{c \in S}{\operatorname{Argmin}} (D(c, p, p_e))$$

avec  $D$  la distance,  $p$  la position des personnages,  $p_e$  la position de l'ennemi,  $c$  un des chemins possible,  $S$  l'ensemble des chemins que nos personnages peuvent prendre et  $C^*$  le chemin qui permet de minimiser la distance relative.



Cette fonction est réalisée à l'aide de plusieurs fonctions, *Norm1*, *Direction*, *MoveToward* qui sont listées dans la classe *heuristics*.

Le code de l'IA heuristiques est décrit dans le script *STD\_AI\_BHV* qui se trouve dans le fichier *Heuristics.bhv*

Notre IA heuristique se fait toujours sur l'équipe rouge, l'équipe bleu est contrôlée par nous ; il faut chaque fois appuyer sur "T" pour changer le tour à l'IA heuristique. IA heuristique va faire réagir sur l'équipe rouge, il passe automatiquement à nous.

#### Remarque:

- Nous avons changé les règles de temps en temps selon notre avancement pour tester ce que nous voudrions. Nous avons donc différentes règles de gagner pour partie IA aléatoire et partie IA Heuristiques.
- Quand c'est notre tour, il faut appuyer sur "M" pour changer l'action "mouvement", ensuite cliquer sur l'endroit où nous voudrions y aller.

### 5.1.3 IA avancé

Cette partie se sépare en deux : *rollback* et *deep\_ai*. *Rollback* permet à notre système de retourner vers un état précédent, *deep\_ai* laisse l'intelligence artificielle avancé jouer notre jeu.

#### Rollback

Pour réaliser la fonction *rollback*, nous ajoutons une fonction *onTurnEnd* sur notre fichier scripte *Regle.bhv* dans le dossier *scr/client/scripts*.

```
#when build_keep is ruined, game over.
FUNCTION OnTurnEnd
  STRING actor_mgr_save_path =ACTOR_MGR::
  CONCAT_INT actor_mgr_save_path TURN_TRACKER
  SAVE_MGR CSV 3 actor_mgr_save_path
  INC TURN_TRACKER
  LTE TURN_TRACKER 3 EXIT
  STRING actor_mgr_save_path =ACTOR_MGR::
  CONCAT_INT actor_mgr_save_path 0
  LOAD_MGR CSV 3 actor_mgr_save_path
  INT TURN = 0
  INT TURN_TRACKER = 0
END FUNCTION
```

Figure 39: La fonction *onTurnEnd*

Cette fonction va générer un nouveau fichier csv qui va permettre d'enregistrer l'état des acteurs dans le dossier `scr/client/BoardSnapshot` lorsqu'on change le tour (donc lorsque nous appuyons sur la touche "T"). Cette fonction permet à notre jeu de revenir à l'état du tour n après k tours (k il faut être nombre impair), comme nous avons mis k=3 et n=0, donc elle va revenir dans l'état initial après 2 tours.

### Deep\_ai

Dans cette partie, l'idée de base est de créer un arbre avec deux ou trois profondeur. Nous construisons un nœud pour chaque action d'un personnage. Nous attribuons une valeur pour représenter les actions des ennemies. Plus cette valeur est grande, plus l'ennemie approche la victoire. Donc nous prenons la valeur minimum. Ensuite nous prenons la valeur maximale pour estimer la pire situation. Nous choisissons ensuite l'action selon la valeur. Dans le schéma suivant, nous choisissons action A2.

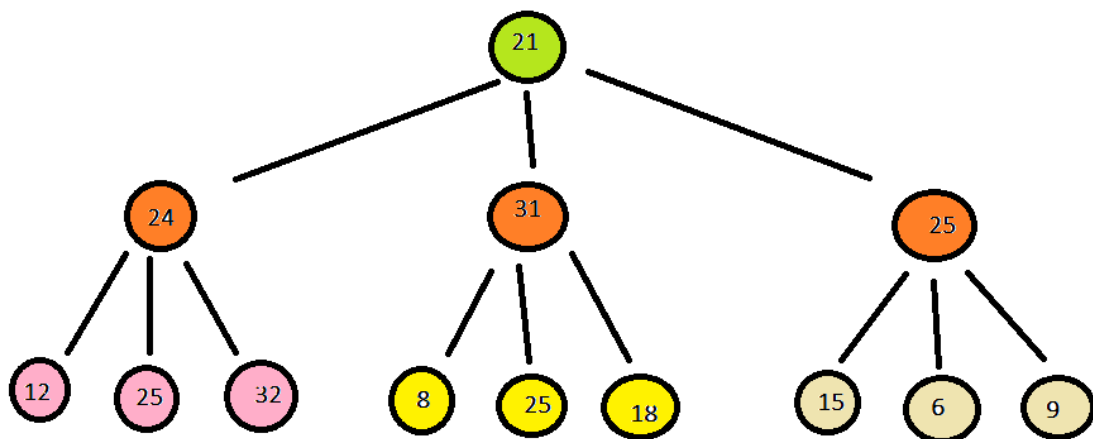


Figure 40: Arbre

Chaque nœud de l'arbre contient va contenir un poids et un comportement associé. Un nœud est représenté par la classe `BehaviourLeaf` représenté ci-dessous :

Tab 12-les propriété des nœuds

Propriété	Description
Caster	Personnage exécutant l'action
Target	Personnage recevant l'action

Input	Type d'action (Offense ou Defense)
Outcome	Poids associé

Pour chaque `Actor`, un arbre est généré et les poids sont calculés selon des coefficients fixes, qui se séparent en deux catégories : un coefficient de défense et un coefficient d'offense. Elles permettent d'influencer directement le comportement

```
#define OFFENSE_BIAS 10
#define DEFENSE_BIAS 2
```

Figure 41: Définition de `OFFENSE_BIAS` et `DEFENSE_BIAS`

La différence entre `OFFENSE_BIAS` et `DEFENSE_BIAS` rend l'IA plus ou moins offensive. On distingue deux cas :

- Si `DEFENSE_BIAS` >> `OFFENSE_BIAS` alors l'IA sera ultra défensive et ne fera que défendre. La probabilité de défaite sera proche de 0 et celle de victoire aussi (car il faut attaquer pour gagner).
- Si `OFFENSE_BIAS` >> `DEFENSE_BIAS` alors l'IA sera ultra offensive et attaquera. La probabilité de victoire est élevée mais celle de défaite aussi car l'IA ne protège plus ses arrières.

Selon la configuration (placement des pions) il faut jouer sur ces coefficients pour rendre l'IA efficace.

**Remarque** : le cas offensif est complété par une estimation des dégâts causés à l'adversaire. Si l'ennemi est à portée, on ajoute au poids les dégâts potentiels infligés.

```

cyan_bowman::ROOT = 0
1|--cyan_bowman::Defense = 2
  2|--red_knight::Defense cyan_bowman = 0
    3|--cyan_bowman::Defense red_knight = 2
    3|--cyan_bowman::Offense red_knight = 10
  2|--red_knight::Offense cyan_bowman = -53
    3|--cyan_bowman::Defense red_knight = -51
    3|--cyan_bowman::Offense red_knight = -43
  2|--red_mage::Defense cyan_bowman = 0
    3|--cyan_bowman::Defense red_mage = 2
    3|--cyan_bowman::Offense red_mage = 10
  2|--red_mage::Offense cyan_bowman = -8
    3|--cyan_bowman::Defense red_mage = -6
    3|--cyan_bowman::Offense red_mage = 2
  2|--red_bowman::Defense cyan_bowman = 0
    3|--cyan_bowman::Defense red_bowman = 2
    3|--cyan_bowman::Offense red_bowman = 10
  2|--red_bowman::Offense cyan_bowman = -8
    3|--cyan_bowman::Defense red_bowman = -6
    3|--cyan_bowman::Offense red_bowman = 2
1|--cyan_bowman::Offense = 10
  2|--red_knight::Defense cyan_bowman = 8
    3|--cyan_bowman::Defense red_knight = 10
    3|--cyan_bowman::Offense red_knight = 18
  2|--red_knight::Offense cyan_bowman = -45
    3|--cyan_bowman::Defense red_knight = -43
    3|--cyan_bowman::Offense red_knight = -35
  2|--red_mage::Defense cyan_bowman = 8
    3|--cyan_bowman::Defense red_mage = 10
    3|--cyan_bowman::Offense red_mage = 18
  2|--red_mage::Offense cyan_bowman = 0
    3|--cyan_bowman::Defense red_mage = 2
    3|--cyan_bowman::Offense red_mage = 10
  2|--red_bowman::Defense cyan_bowman = 8
    3|--cyan_bowman::Defense red_bowman = 10
    3|--cyan_bowman::Offense red_bowman = 18
  2|--red_bowman::Offense cyan_bowman = 0
    3|--cyan_bowman::Defense red_bowman = 2
    3|--cyan_bowman::Offense red_bowman = 10

```

```

cyan_dragon::ROOT = 0
1|--cyan_dragon::Defense = 2
  2|--red_knight::Defense cyan_dragon = 0
    3|--cyan_dragon::Defense red_knight = 2
    3|--cyan_dragon::Offense red_knight = 45
  2|--red_knight::Offense cyan_dragon = -43
    3|--cyan_dragon::Defense red_knight = -41
    3|--cyan_dragon::Offense red_knight = 2
  2|--red_mage::Defense cyan_dragon = 0
    3|--cyan_dragon::Defense red_mage = 2
    3|--cyan_dragon::Offense red_mage = 45
  2|--red_mage::Offense cyan_dragon = -8
    3|--cyan_dragon::Defense red_mage = -6
    3|--cyan_dragon::Offense red_mage = 37
  2|--red_bowman::Defense cyan_dragon = 0
    3|--cyan_dragon::Defense red_bowman = 2
    3|--cyan_dragon::Offense red_bowman = 45
  2|--red_bowman::Offense cyan_dragon = -8
    3|--cyan_dragon::Defense red_bowman = -6
    3|--cyan_dragon::Offense red_bowman = 37
1|--cyan_dragon::Offense = 45
  2|--red_knight::Defense cyan_dragon = 43
    3|--cyan_dragon::Defense red_knight = 45
    3|--cyan_dragon::Offense red_knight = 88
  2|--red_knight::Offense cyan_dragon = 0
    3|--cyan_dragon::Defense red_knight = 2
    3|--cyan_dragon::Offense red_knight = 45
  2|--red_mage::Defense cyan_dragon = 43
    3|--cyan_dragon::Defense red_mage = 45
    3|--cyan_dragon::Offense red_mage = 88
  2|--red_mage::Offense cyan_dragon = 35
    3|--cyan_dragon::Defense red_mage = 37
    3|--cyan_dragon::Offense red_mage = 80
  2|--red_bowman::Defense cyan_dragon = 43
    3|--cyan_dragon::Defense red_bowman = 45
    3|--cyan_dragon::Offense red_bowman = 88
  2|--red_bowman::Offense cyan_dragon = 35
    3|--cyan_dragon::Defense red_bowman = 37
    3|--cyan_dragon::Offense red_bowman = 80

```

Figure 42 : Exemple d'arbres générés

### 5.1.4 Comportement

Partie “RandomAI”

On scripte le comportement de l'IA dans un fichier script .bhv

On commence tout d'abord par définir les joueurs dans le fichier script des règles du jeu (/src/client/scripts/Regles.bhv ).

```
PLAYER PLAYER_2 0 MNK
PLAYER PLAYER_1 1 RandomAI
```

Figure 43 : Définitions des joueurs

Le comportement de PLAYER\_1, qui représente l'IA, est dans le script “RandomAI”. Le contenu du script est ci-dessous.

```
FUNCTION TurnBegin
CALL STD_AI_BHV red_spearman.ID
CALL STD_AI_BHV red_bowman.ID
CALL STD_AI_BHV red_mage.ID
CALL STD_AI_BHV red_knight.ID
CALL STD_AI_BHV red_dragon.ID
CALL FUNC::EndTurn PlayerID
END FUNCTION

FUNCTION STD_AI_BHV
CALL FUNC::RandomInt -2 2
INT r0 = 0
ADD r0 RET:RandomInt
CALL FUNC::RandomInt -2 2
INT r1 = 0
ADD r1 RET:RandomInt
ADD r0 ACTOR_MGR::ARGS:0.X
ADD r1 ACTOR_MGR::ARGS:0.Y
CALL FUNC::CheckPosition 3 r0 r1
EQUAL RET:CheckPosition -1 JMP:6
EQUAL RET:CheckPosition 0 JMP:2
SEND STD_ATTACK ACTOR_MGR::ARGS:0.ID ACTOR_MGR::RET:CheckPosition.X ACTOR_MGR::RET:CheckPosition.Y
NEQUAL RET:CheckPosition 0 JMP:3
SUB r0 ACTOR_MGR::ARGS:0.X
SUB r1 ACTOR_MGR::ARGS:0.Y
SEND STD_MOVE_RELATIVE ACTOR_MGR::ARGS:0.ID r0 r1
END FUNCTION
```

Figure 44 : Script “RandomAI.bhv”

Au début de chaque tour, la classe `state::WorldHandler` appelle la fonction `TurnBegin` de chaque script. Ainsi quand le tour de l'IA vient, sa fonction `TurnBegin` est appelée et les actions sont effectuées.

Partie IA heuristique

Nous avons créé également un script `Heuristix.bhv` pour les règles de IA heuristique.

La fonction `STD_AI_BHV` est la principale fonction qui gère l'IA heuristique, c'est lui qui fait appelle plusieurs sous fonctions `MoveToward`, `IsInReach` et `STD_Attack`:

- `MoveToward` permet de faire avancer les personnages dans la direction qu'il faut
- `IsInReach` permet de vérifier s'il y a des personnages adversaires dans la zone d'attaque
- `STD_ATTACK` est l'action de l'attaque

Donc `STD_AI_BHV` fait avancer les personnage dans la direction qu'il faut, et il faut attaquer si les acteurs adversaires dans la zone d'attaque.

La fonction `TurnBegin` fait appelle à la fonction `STD_AI_BHV` pour tous les personnages de l'équipe rouge. Il fixe à tous les personnages la direction vers les personnages adversaires correspondant. Par exemple, un lancier rouge va se diriger vers le lancier bleu et ainsi de suite.

```
src > client > scripts > Heuristix.bhv
1  FUNCTION TurnBegin
2  CALL STD_AI_BHV red_spearman.ID cyan_spearman.X cyan_spearman.Y
3  CALL STD_AI_BHV red_bowman.ID cyan_bowman.X cyan_bowman.Y
4  CALL STD_AI_BHV red_mage.ID cyan_mage.X cyan_mage.Y
5  CALL STD_AI_BHV red_knight.ID cyan_knight.X cyan_knight.Y
6  CALL STD_AI_BHV red_dragon.ID cyan_dragon.X cyan_dragon.Y
7  CALL FUNC::EndTurn PlayerID
8  END FUNCTION
9
10 FUNCTION STD_AI_BHV
11 CALL FUNC::MoveToward 3 ARGS:0 ARGS:1 ARGS:2
12 #CALL FUNC::CheckPosition 3 ARGS:1 ARGS:2
13 CALL FUNC::IsInReach 3 ARGS:0 ARGS:1 ARGS:2 1000
14 EQUAL RET:IsInReach 0 JMP:1
15 #LT RET:CheckPosition 1 JMP:1
16 SEND STD_ATTACK ARGS:0 ARGS:1 ARGS:2
17 END FUNCTION
```

Figure 45: Le script "Heuristics.bhv"

## AI vs AI

Pour déterminer quel joueur est une AI, on modifie le fichier script `/src/client/scripts/Regles.bhv` (règle du jeu). On aura plusieurs archétypes disponibles pour le type de joueur :

- `MNK` : MouseNKeyboard, correspond à un joueur humain
- `RandomAI` : IA random

- HEURISTIXCYAN : IA heuristique pour le joueur cyan
- HEURISTIXRED : IA heuristique pour le joueur rouge
- DEEPRED : IA avancé pour le joueur rouge

```
PLAYER PLAYER_2 0 HEURISTIXCYAN
PLAYER PLAYER_1 1 RandomAI
```

Figure 46 : Définitions des joueurs

### 5.1.5 Tests des IAs

Pour tester le bon fonctionnement de nos IAs on a réalisé des simulations de notre jeu en faisant s'affronter les IA les unes contre les autres.

Dans le cas de l'IA random contre l'IA heuristique l'IA heuristique gagne à chaque fois car l'IA random ne fait que des déplacements aléatoires donc la possibilité qu'elle arrive jusqu'à la base ennemie et arrive à la détruire est quasiment nulle.

Dans le cas où deux IA heuristiques s'affrontent, c'est celle qui joue en première qui gagne à tous les coups car elle nécessite de faire moins de déplacements.

L'IA avancé gagne à chaque fois contre les deux types d'IA peu importe qui commence la partie. Ceci s'explique par le fait que l'IA avancée est plus sélective dans son comportement offensif.

Il faut prendre en compte que notre jeu inclut des déplacements, ce qui fait que l'efficacité des IAs sont étroitement liés au placement des personnages sur la carte.

## 5.2 Conception logicielle

Nous avons créé le diagramme des classes suivant :

- La classe `RandomAI` propose des fonctions statiques que nous appelons dans le script.
  - o La fonction `RandomInt` fait un tirage au sort d'un nombre d'entier aléatoire entre les cadres `[param[0], param[1]]`
  - o La fonction `IsInRange` permet de vérifier si les personnages adversaires sont donc le cadre d'attaque ou le déplacement. Il est utile aussi dans la partie Heuristics.

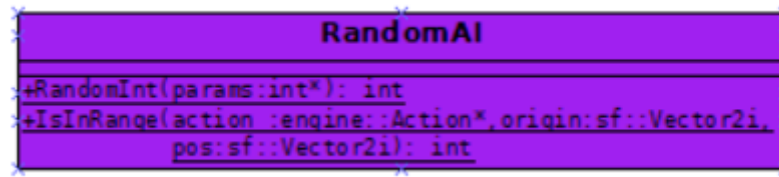


Figure 47: La classe “RandomAI”

- La classe **Heuristics** qui dépend de la classe **Actor**. Il contient plusieurs fonction utile comme par exemple :
  - La fonction `Norm1` qui calcule les normes des différentes position des personnage ou la distance entre les différentes actors (par exemple:les personnages ou les bâtiment)
  - La fonction `MoveToward` permet de faire avancer les personnages dans la direction qu’il faut. Il sera être appelé dans scripte `Heuristique.bhv`
  - `Direction` permet de déterminer les nombres de la case vertical et horizontal entre les personnages départ vers destinataires.

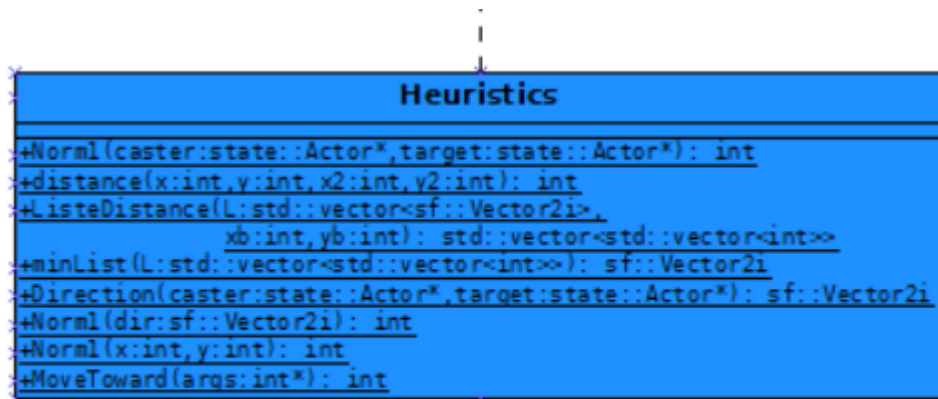


Figure 48: La classe “Heuristics”

Nous obtenons le diagramme de la classe pour AI comme le figure 52.

- Les classes pour faire AI avancé
  - classe `deep_AI` qui permet d’effectuer AI avancé.



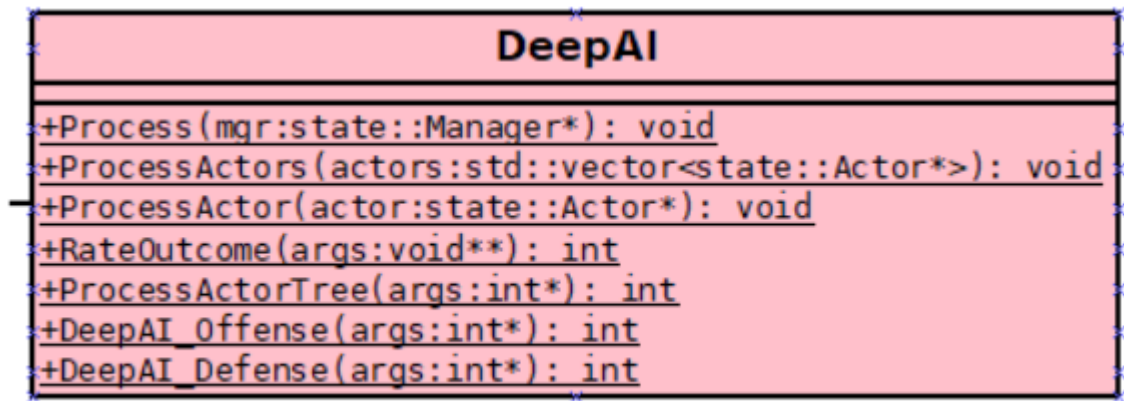


Figure 49: La classe “DeepAI”

→ La class ActionLeaf qui met les actions comme les feuilles de notre arbre

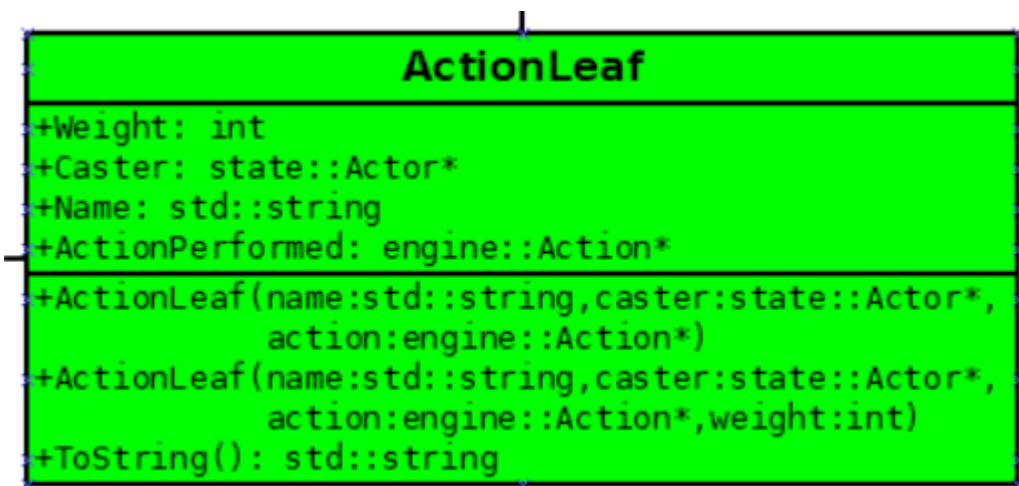


Figure 50: La classe “ActionLeaf”

→ La classe BehaviorLeaf

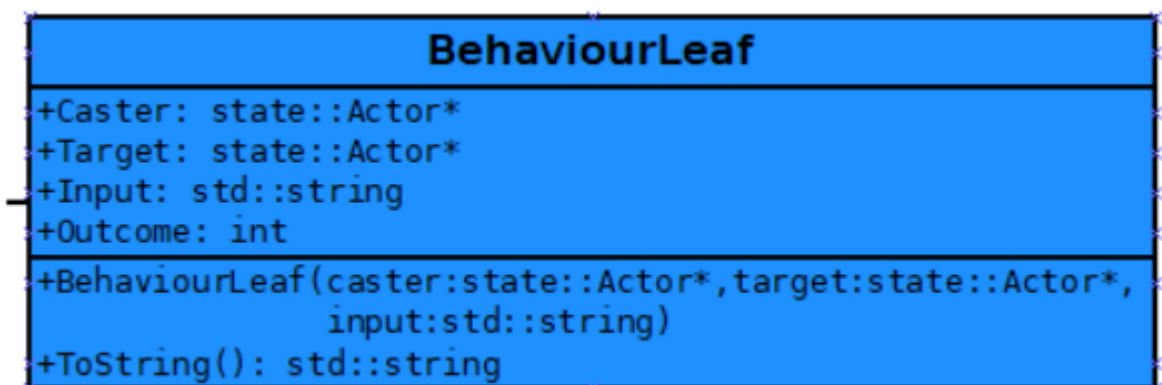


Figure 51: La classe “BehaviorLeaf”

→ La classe BehaviorTree

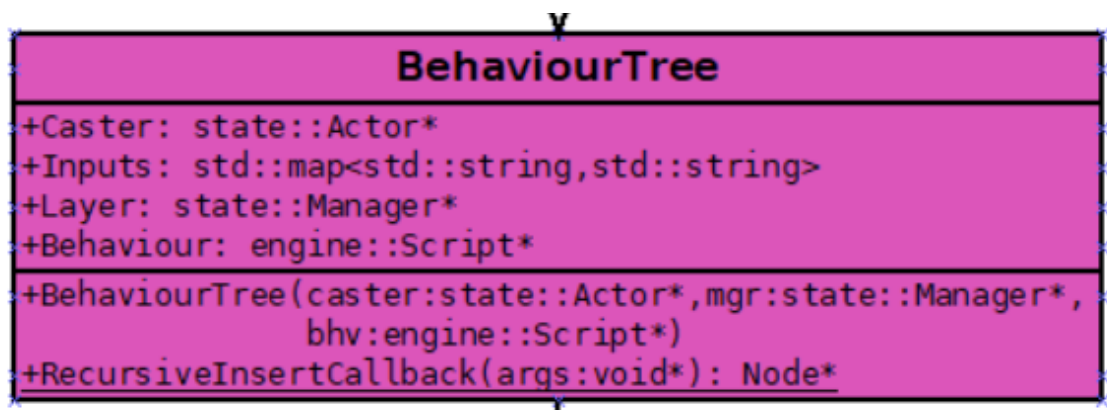


Figure 52: La classe "BehviorTree"

→ La classe Tree qui permet de construire notre arbre/

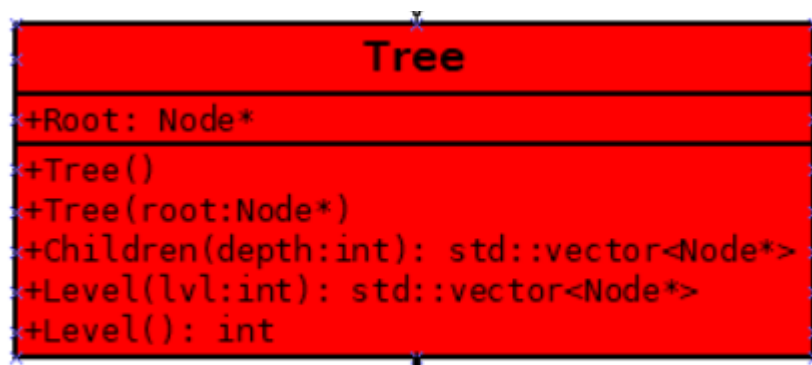


Figure 53: La classe "Tree"

→ La classe Node qui construit les nœuds de notre arbre.

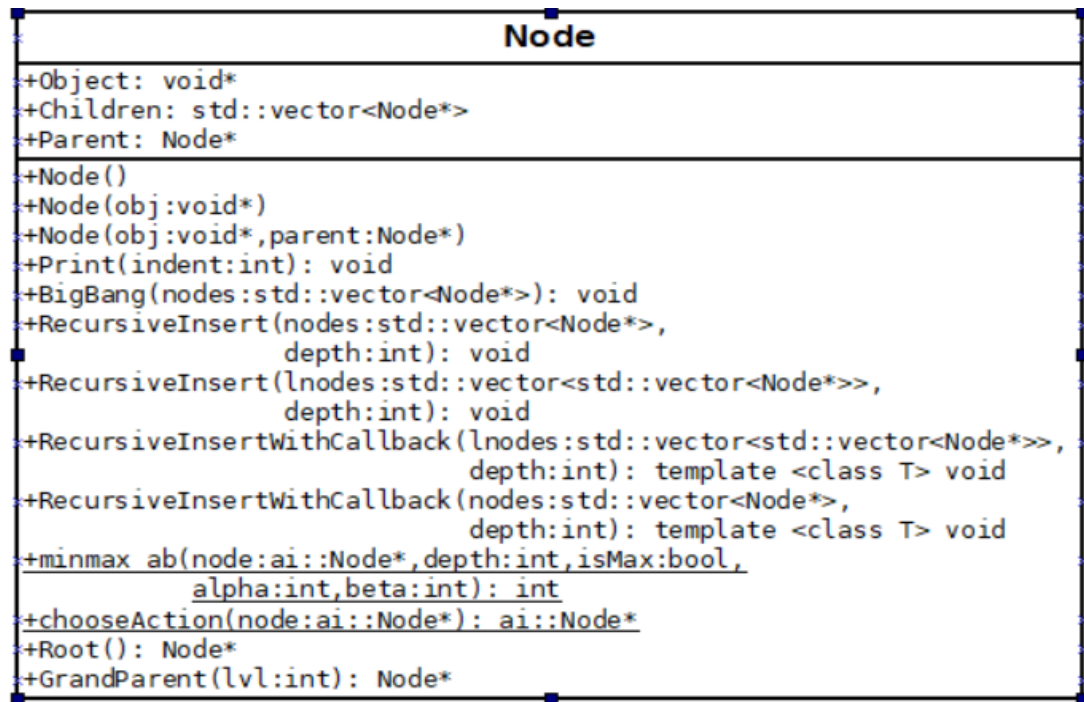


Figure 54: La classe "Node"

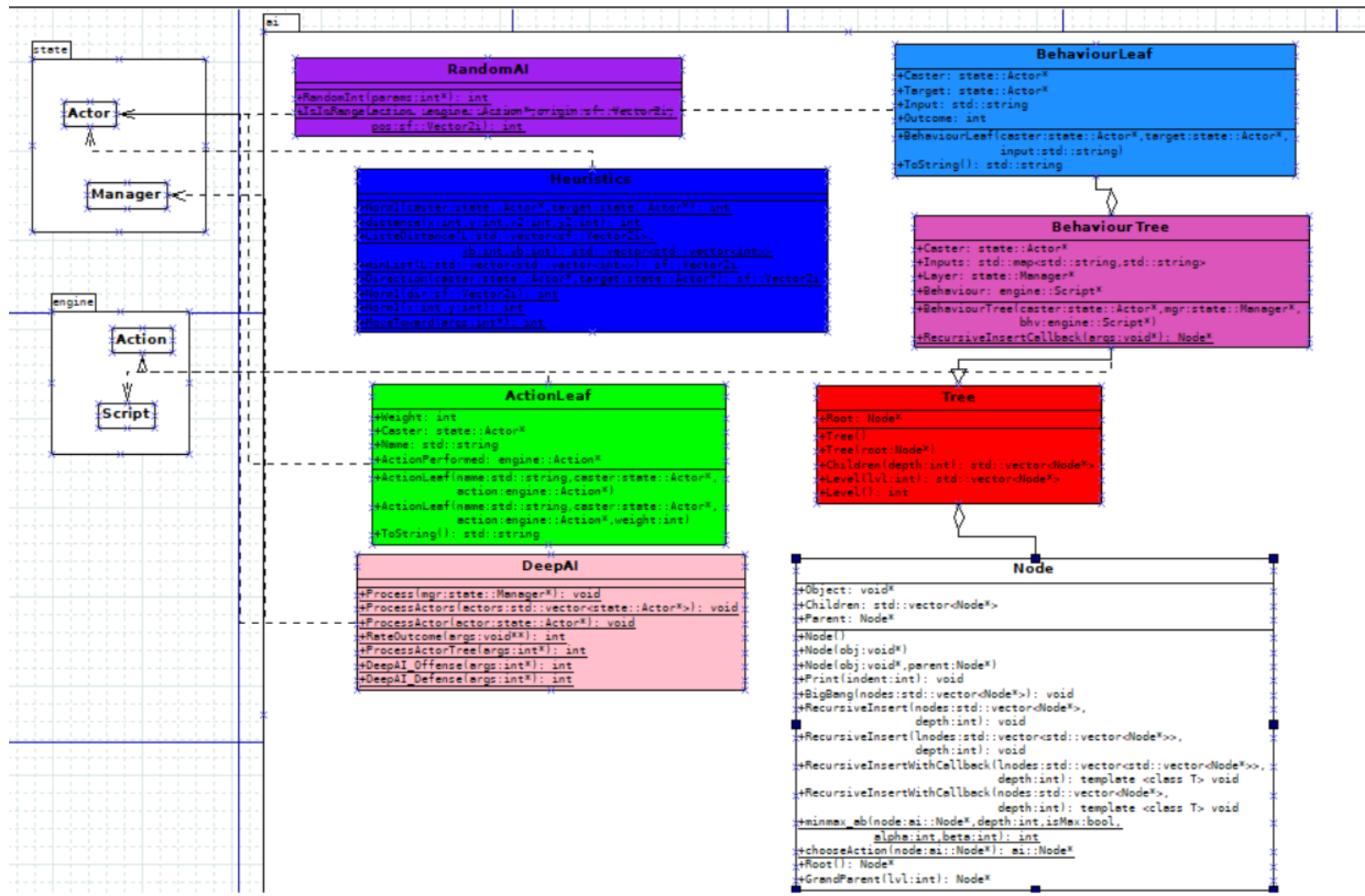


Figure 55: Le diagramme des classes AI proposé des fonctions statiques que nous appelons dans le script

# 6 Modularisation

## 6.1 Organisation

### 6.1.1 Répartition sur différents threads

Nous avons jusqu'à maintenant, la partie Engine ,Render et State sont exécutés séquentiellement comme la figure suivante:

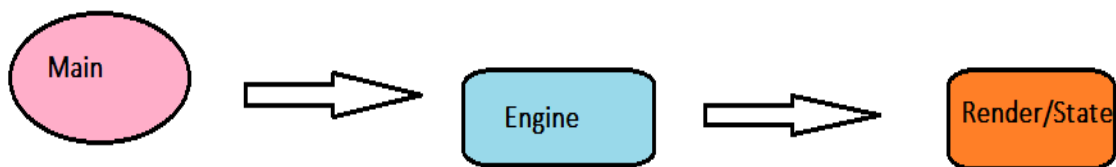


Figure 56: Schéma représentative des fonctionnement séquentielle

Cela va provoquer un ralentissement du jeu. En effet, il prend le temps de passer d'une étape à l'autre.

Au lieu de l'exécution séquentielle ci-dessus nous allons exécuter chaque commande dans un thread séparé. Lors de la réception d'une commande par l'engine celle-ci est traitée dans un thread (Thread Spawning). Sachant que le volume de commande à traiter est relativement faible (max 2-3 en simultanées), il n'est pas nécessaire de faire un système pour gérer une "pool" de threads.

### 6.1.2 Répartition sur différentes machines

#### Fonctionnement global

Notre partie serveur, qui va pouvoir permettre à différentes machines de se connecter et de jouer en même temps, se compose de la manière suivante :

- des clients en C++
- un host en C++
- un serveur en JavaScript qui va jouer le rôle de passerelle entre les clients et le host

Les clients et le serveur vont utiliser des WebSockets pour communiquer entre eux au lieu de simples requêtes HTTP. En effet, au vu du type de jeu que nous développons, nous voulions implémenter du temps réel pour que les deux joueurs puissent observer les déplacements et les actions réalisés de manière instantanée. Les WebSockets sont un protocole qui semble parfaitement adapté à la situation.

Un WebSockets est une connexion persistante entre le client et le serveur. Ce protocole va ouvrir un canal de communication bidirectionnel sur un socket TCP. Ce canal bidirectionnel est ce qui le démarque du HTTP qui est un protocole unidirectionnel. Avec HTTP, toute données envoyées depuis le serveur doit d'abord être demandée par le client, ce qui est contraignant si on souhaite réaliser des applications en temps réel comme un chat. Avec WebSockets, on peut envoyer des données depuis le serveur ou le client. Ainsi, le client peut être notifié dès un changement d'état du serveur. Il utilise HTTP en tant que mécanisme de transport initial, mais garde la connexion TCP ouverte pour qu'on puisse l'utiliser pour envoyer d'autres requêtes.

Nous avons choisi d'implémenter cette structure car à l'heure où nous devons implémenter la partie multijoueur du jeu, nous n'avons plus beaucoup de temps. Nous étions déjà familier avec le concept des websockets ainsi que des serveurs JavaScript, c'est pourquoi il a été jugé plus préférable de partir là-dessus.

Le fonctionnement de notre architecture se résume sur le schéma ci-dessous.

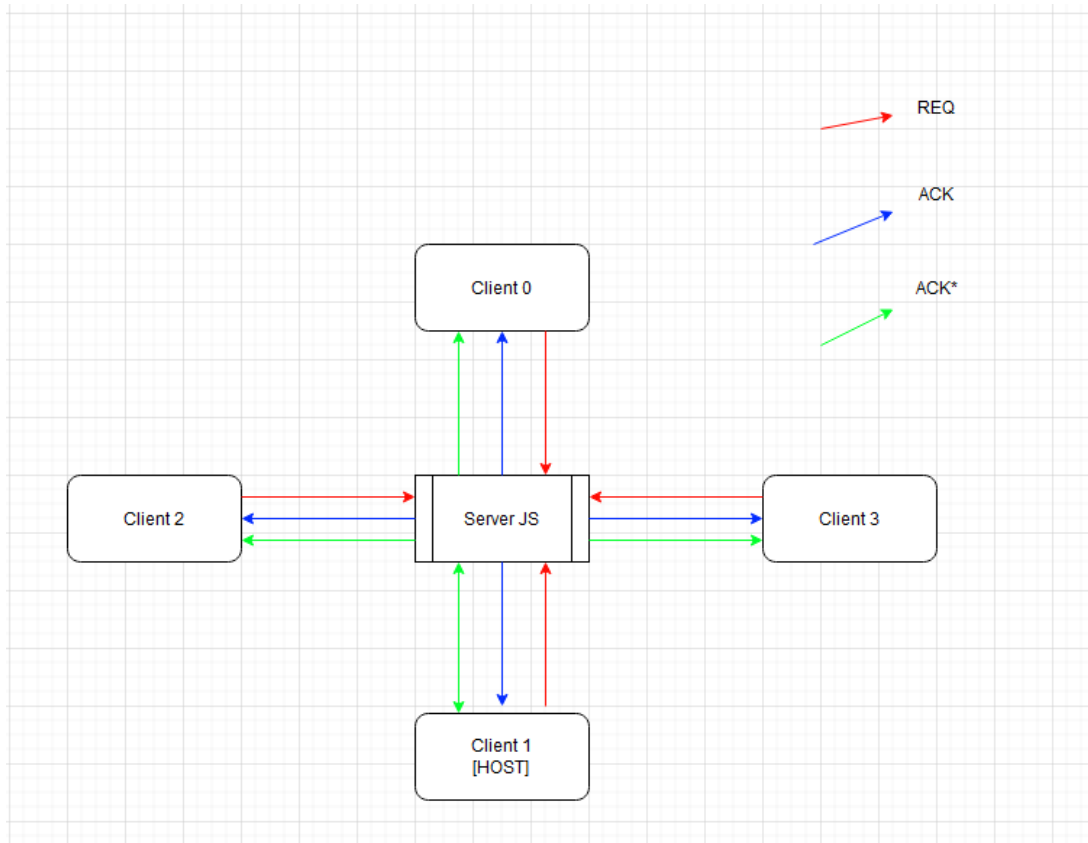


Figure 57: Architecture Client-Serveur

On suppose ici que nous avons trois clients qui vont vouloir jouer à une partie. Nous avons le host et le serveur JS. Ces trois entités vont échanger des flux entre elles. On distingue plusieurs types de sous-catégories, représentées par les couleurs sur le schéma ci-dessus :

- REQ (rouge) : ce sont les flux qui représentent les requêtes initiées par un client.. Par exemple. Un utilisateur va demander de se connecter au serveur. On va donc envoyer depuis l'utilisateur une requête avec l'en-tête REQ : `req_create_user`

- **ACK (bleu)** : il s'agit de la réponse initiée par le serveur à une requête initié par un client. Un flux va toujours avoir une requête et sa réponse. Par exemple, lors de la création du joueur dans le serveur, nous allons envoyer une réponse de avec ack en-tête : `ack_create_user`
- **ACK\* (vert)** :il s'agit de flux initiés par le HOST . Par exemple, les réponses au `net_cmd` ou les actions réalisées par un client comme bouger un personnage, sont envoyées vers le host depuis le client en utilisant le serveur JS comme intermédiaire. La réponse est alors transmise à tous les clients connectés à la salle

## Flux

Le serveur javascript va jouer le rôle de passerelle entre les clients et le host, et pour communiquer entre eux nous allons définir des messages pour envoyer les données que l'on souhaite. Cela fonctionne à peu près comme des événements.

On peut par exemple prendre pour exemple un flux `create_user` pour créer un utilisateur. Nous allons définir deux sous-catégories pour chaque flux, comme expliqué précédemment. Une requête avec l'en-tête REQ et une réponse avec l'en-tête ACK. Cela va se passer pour chaque flux que nous avons défini dans le tableau suivant.



Tab 13-Flux

Nom	Préfix	Initiateur	Passerelle	Destinataire	Données à envoyer	Description
create_user	req	User[N]	/	Serveur JS	"<ROOM_NAME>;<HOST_NAME>"	Création d'un utilisateur
	ack	Server JS	/	User[N]	Status code	
create_room	req	User[N]	/	Serveur JS	username	Création d'une salle
	ack	Server JS	/	User[N]	Status code	
join_room	req	User[N]	/	Serveur JS	"<roomName>"	Rejoindre une salle spécifique
	ack	Serveur JS	/	User[N] room.host	Status code	
rooms	req	Dashboard	/	Serveur JS	/	Récupère la liste des salles
	ack	Serveur JS	/	Dashboard	rooms	
clients	req	Dashboard	/	Serveur JS	/	Récupère la liste des clients
	ack	Serveur JS	/	Dashboard	clients	
net_cmd	req	User[N]	Serveur JS	room.host	cmd	Commande users i.e. attaquer, se déplacer ...
	ack	room.host	Serveur JS	room.players	cmd	
heartbeat	/	User[N]	/	Serveur JS	/	Heartbeat
start_game	req	User[N] (host only)	/	Serveur JS	roomName	Notifier les utilisateurs d'un début de partie
	ack	Serveur JS	/	room.players & host	room.players	

## Serveur JS

Le serveur JavaScript va fonctionner de la manière suivante.

Nous avons besoin de prendre en compte deux objets : un objet `User` et un objet `Room` dont leurs classes sont représentées ci-contre.

Tab 14-Classe User

Propriétés	Type	Description
<code>name</code>	<code>string</code>	Nom de l'utilisateur
<code>id</code>	<code>int</code>	Identifiant unique qui correspond à l'ID du socket
<code>counter</code>	<code>int</code>	Compteur qui va servir au heartbeat
<code>room</code>	<code>string</code>	Nom de la salle que l'utilisateur a rejoint

Tab 15-Classe Room

Propriétés	Type	Description
name	string	Nom de la salle
players	string[]	Tableau qui contient le nom des joueurs qui ont rejoint cette salle
host	string	Nom du client qui s'occupe d'héberger la partie
status	int	Représente le statut de la salle : 0 : en attente  -1 : partie en cours  1 : joueur 1 gagne  2 : joueur 2 gagne

Nous allons décrire le comportement du serveur JavaScript lors d'un scénario où un utilisateur souhaite se connecter au serveur, créer une salle et rejoindre la salle.

1. Les utilisateurs rejoignent le lobby général avec la requête "create\_user", ils sont alors placés dans la liste des Clients.
2. Un utilisateur peut créer une Room avec la requête "create\_room". Le createur d'une Room ne peut pas rejoindre la room en tant que joueur, il est forcément le HOST, c'est un utilisateur mais qui ne joue pas, son rôle est d'héberger la partie.
3. Une fois la room créée, les utilisateurs en attente dans le lobby peuvent rejoindre la Room avec la requête "join\_room". Le HOST est prévenu chaque fois qu'un utilisateur rejoint la room.
4. Lorsqu'il y a 2 joueurs dans la Room, le HOST démarre la partie avec la requête "start\_game".

5. Tout au long de la partie le serveur JS redirige les requêtes vers le HOST, qui traite les commandes puis les renvoie aux autres clients à travers le serveur JS. C'est le serveur JS qui redirige les commandes vers les clients concernés (plusieurs parties peuvent être jouées en même temps).

Les requêtes `rooms` et `clients` sont utilisées par le dashboard qui va afficher la liste des salles et clients en temps réel.

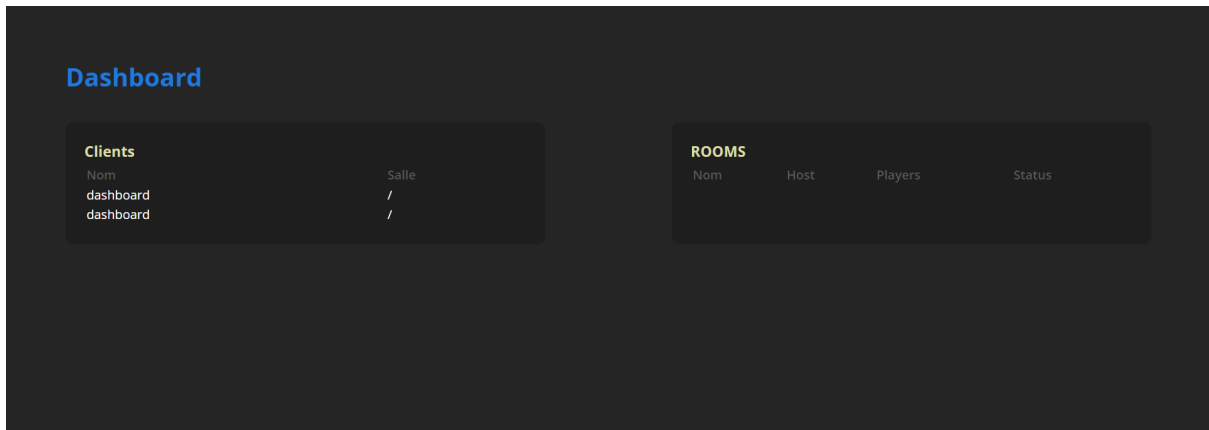


Figure 58: L'affichage de salle de Jeu.

### Heartbeat

Afin de s'assurer que l'utilisateur reste connecté au serveur JavaScript et ne soit pas déconnecté automatiquement, nous allons implémenter un heartbeat. Le principe est le suivant.

La classe `User` possède une propriété qui s'appelle `counter`. Lors de la connexion d'un utilisateur, le `counter` vaut 5. Le serveur JavaScript va décrémenter cette valeur toutes les 2 secondes. Lorsque la valeur atteint 0, on enlève le client de la liste des clients connectés et on le déconnecte.. Pour que l'utilisateur reste connecté, il va envoyer lui aussi toutes les 2 secondes au serveur une requête `heartbeat` pour augmenter de 1 `counter`.