Projet Logiciel Transversal

Tactical Wars

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

Sommaire

1 . Objectif

1.1 . Présentation générale

Archétype : Advance Wars/Wargroove

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

Figure 1: Image du jeu « Advanced Wars »

Figure 2: Image du jeu « Wargroove »

1.2 . Règles du jeu et Ressources

Règles :

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

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

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

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

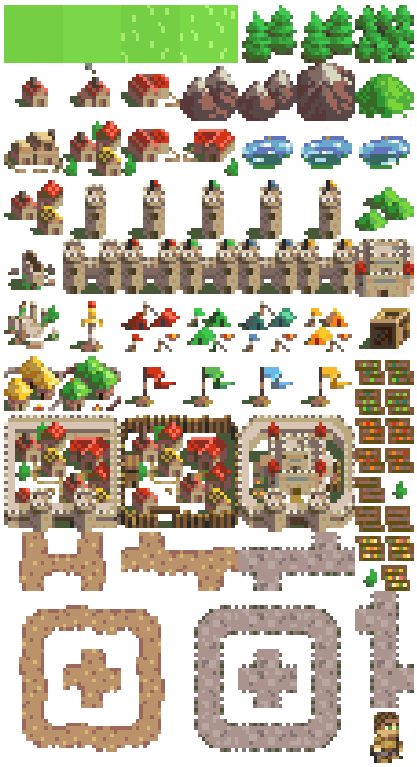


Figure 3: Tuile d'image pour les décors (terrains et bâtiments)

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

Figure 4: Chauve-souris

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

Figure 5: Sorcière

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

Figure 6: Nain

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

Figure 7: Cyclope

Comme dans le cadre de ce projet nous n'allons pas nous servir des animations des unités et que nous n'avons pas besoin de tous les sprites à disposition, nous allons créer un fichier unique (à partir de ceux que nous avons) qui comportera uniquement ce dont nous avons besoin. Ce fichier servira de fichier de texture unique pour l'affichage ce qui permet d'alléger le travail de la carte graphique.

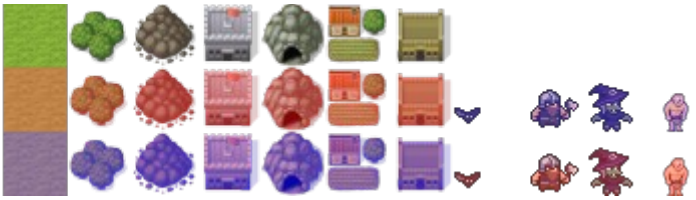


Figure 8: Fichier de texture final

Notons que sur ce fichier de texture final nous avons ajouter les couleurs sur les sprites car nous allons essayer de gérer les couleurs d'affichge selon le personnage auquel ils appartienent.

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

Characters	PV	Portée d'Attaque	Dégats	Portéée de déplacement	Coût en mana
Sorcière	30	100	70	60	70
Cyclop	100	60	100	20	100
Chauve-souris	10	40	10	100	20
Nain	50	20	50	40	50

2 . Description et conception des états

2.1 . Description des états

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

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

2.1.1 . État des éléments fixes sur la carte

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

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

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

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

2.1.2 . État des éléments mobiles sur la carte

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

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

2.1.3 . État des éléments propres au joueur

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

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

2.1.4 . L'état général de la partie

L'état général de la partie doit comporter :

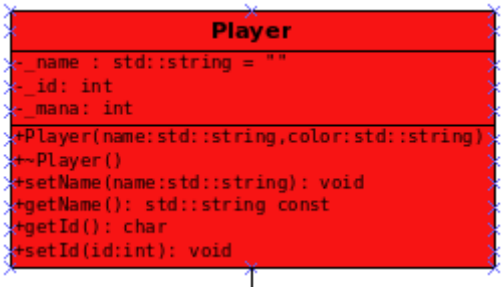
- Le numéro du tour actuel de jeu. (identique pour les deux joueurs)
- Un tour est composé de deux phases : la phase de jeu du Joueur 1 et la phase de jeu du Joueur 2. L'état du jeu doit connaître la phase pour savoir qui doit jouer.
- Le statut de défaite ou victoire des joueurs une fois la partie terminée.

2.2 . Conception Logiciel

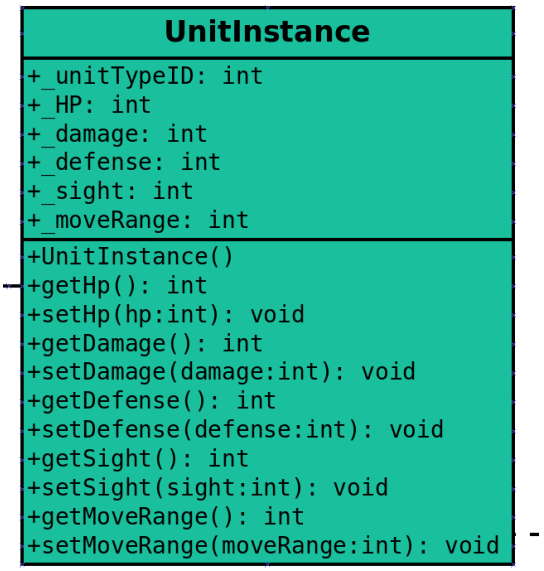
Le state est fondé sur une classe state qui récapitule l'état générale du jeu :

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

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



Attribut	Type	Explication
_name	String	Nom du joueur
_id	Int	Identifiant du joueur
_mana	Int	Quantité de mana



La classe **Unit** contient plusieurs attributs : size qui contient le nombre de combattants dans l'unité, positions qui continent la position de l'unité, singleHpUnit qui contient les points de santé d'un unique membre de l'unité, globalHp qui contient les points de santés de l'unité complète, attackrange qui permet d'obtenir la portée de l'attaque d'une unité, sightRange qui permet d'obtenir la puissance de la vision d'un membre de l'unité, globaleDamage qui permet de quantifier les dégâts totaux infligés par une unité, moveRange qui permet de quantifier les déplacements d'une

unité, unitID pour identifier l'unité de façon unique et typeId pour identifier la race de l'unité. Nous avons également ajouté un attribut globalID afin de générer des id unique pour chaque unité.

La classe contient plusieurs méthodes :

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

De cette classe hérite les différents types d'unités :

- **Bat**
- **Dwarf**
- **Witch**
- **Cyclop**

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

De cette classe hérite :

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

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

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

deux méthodes *setX* et *setY*

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

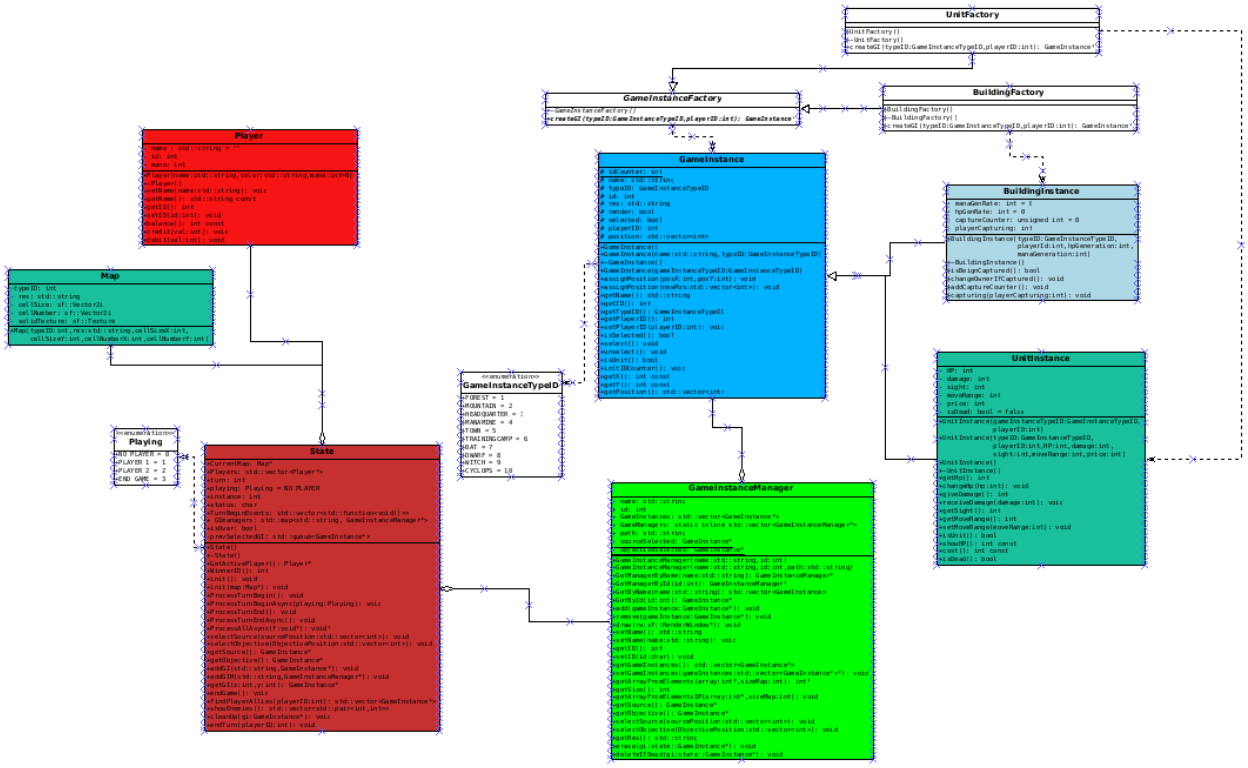


Figure 9: state.dia

3 . Description et conception du rendu

3.1 . Stratégie de rendu d'un état

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

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

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

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

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

Pour l'affichage des états à un instant t, nous appelons un state partiel qui contient uniquement les informations que le client doit connaître à un instant t, une fois ce state généré et passé en référence au render, nous utilisons ses données pour l'afficher. Pour l'instant à chaque changement d'état, tout l'affichage du state est recréé.

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

3.2 . Conception logiciel

Nous organisons notre rendu autour de trois classes :

- **Scene** : ***Scene*** est la classe « fenêtre » de notre render. Il s'agit également de la classe principale qui va stocker tous les ***Layer*** qui représente les différentes couches à afficher. Dans ***Scene*** on appelle également un state masqué qui contient uniquement les informations visibles par le client. à chaque changement d'état la scène est nettoyée et se reconstruit entièrement. Cette classe dépend de **sf::*RenderWindow*** qui nous permet de *draw* nos résultats.

- **Layer** : cette classe se charge d'afficher une couche graphique du jeu en fonction des informations qui lui sont passées. Ces informations sont en fait des listes de **state::GameInstance,** ces objets contiennent des positions et des ID, à chaque type de GameInstance sera associée une tile présente dans le tileset correspondant à la couche graphique, une fois toutes les informations récoltés ont peut générer un array qui sera directement *draw* dans la fenêtre.

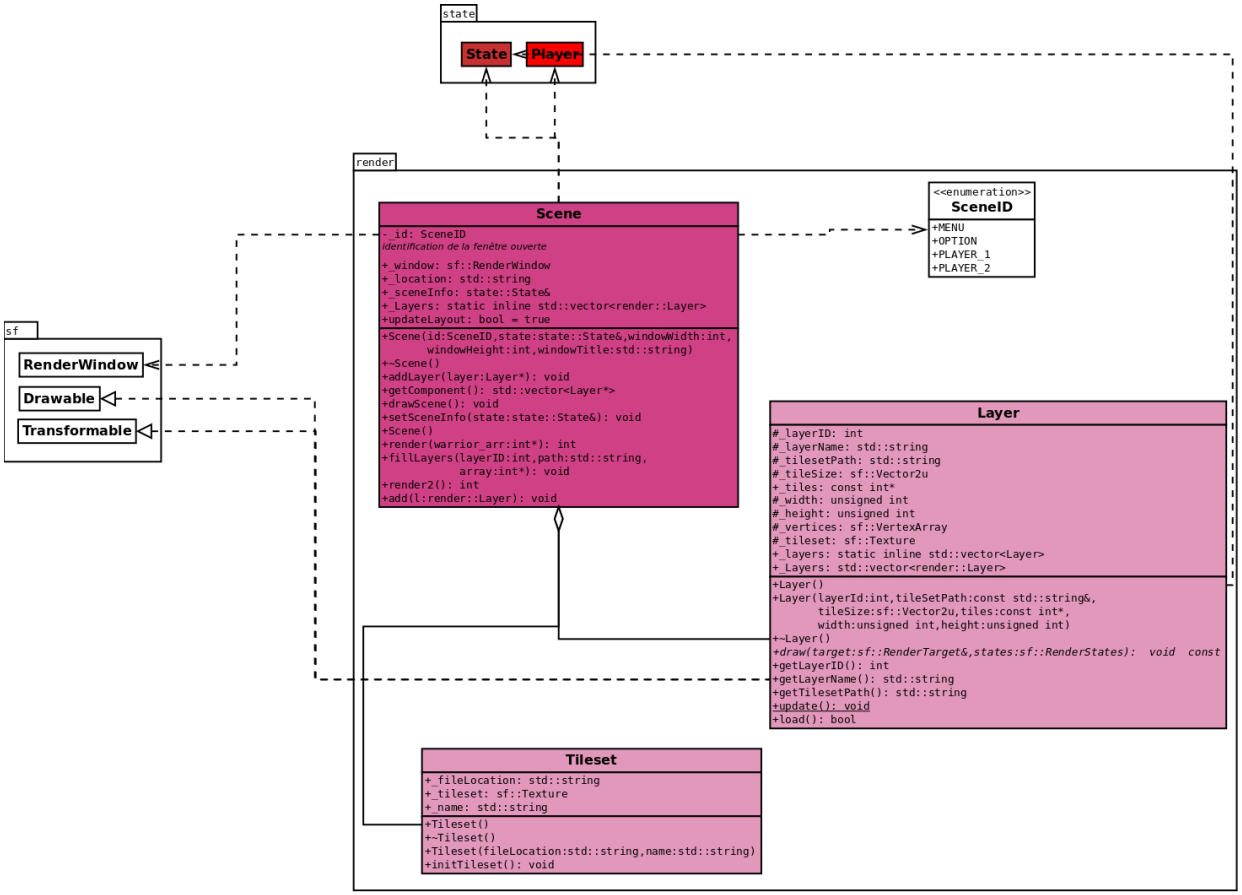


Figure 10: render.dia

4 . Règles de changement d’état et moteur de jeu

4.1 . Changement extérieur

Les changements extérieurs sont provoqués par des commandes extérieurs, comme la pression sur une touche ou un clique sur la souris.
La commande principale est la sélection d’un personnage, delà s’en suit les commandes d’actions qui se succèdent comme suit :

- Commande de sélection : si le joueur a sélectionné un élément contrôlable propose les actions possibles, sinon la commande ne fait rien.
- Commande de Construction : si le joueur a sélectionné un camp d’entraînement, il aura la possibilité de dépenser son mana pour se payer de nouvelles unités.
- Commande de mouvement : si le joueur clique sur le bouton de commande et et clique sur la case où il veut se rendre, le personnage se déplacera si cela lui est possible.
- Commande d’attaque : même principe, après le mouvement le joueur pourra attaquer avec son unité mais devra s’astreindre au restriction qui incombent à son unité sélectionnée.

4.2 . Conception Logiciel

Le diagramme des classes pour le moteur du jeu est présenté ci dessous. L’ensemble du moteur de jeu repose sur un patron de conception de type Command, et a pour but la mise en œuvre différée de commandes extérieures sur l’état du jeu.

Classes Command. Le rôle de ces classes est de représenter une commande. On définit un CommandTypeID pour toutes les classes qui héritent de l’interface Command.
A ces classes, on a défini un type de commande avec CommandTypeId pour identifier précisément la classe d’une instance.

Class Engine. Cette classe fait office de tour de contrôle pour les commandes, elle stocke les commandes et les exécute.

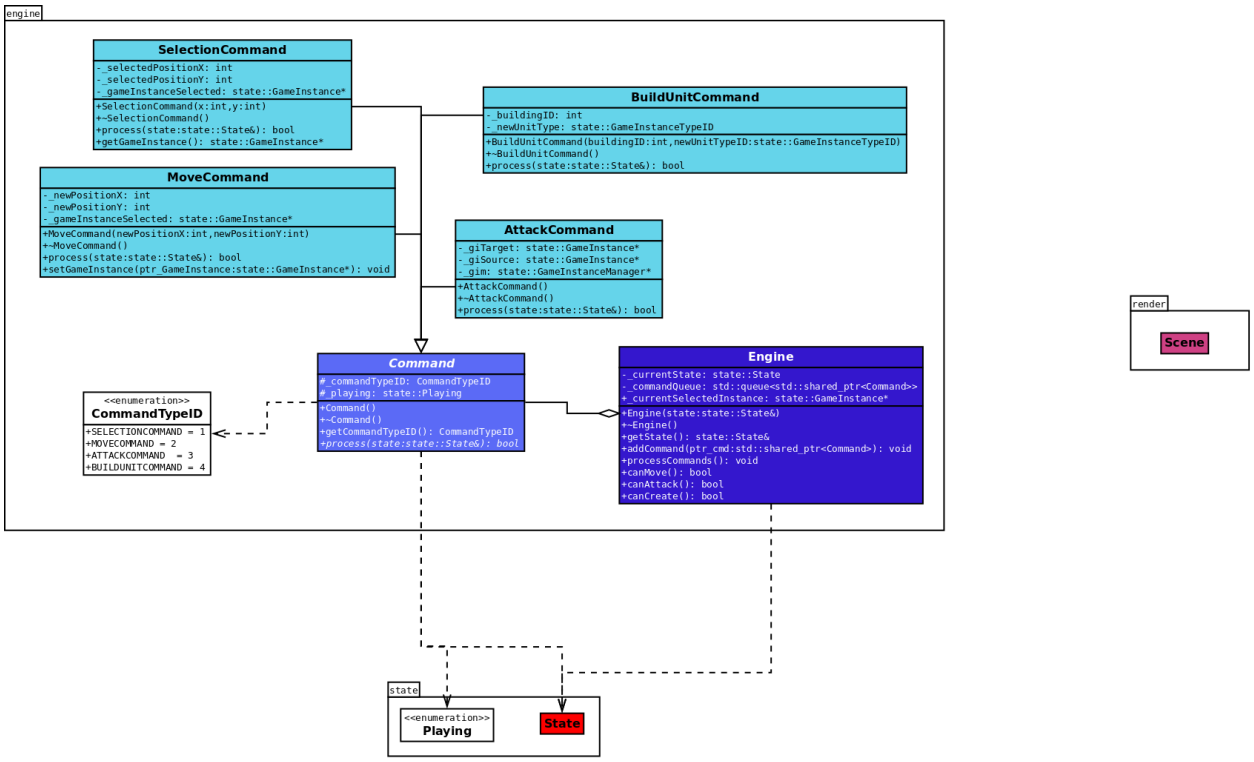


Figure 11: engine.dia

5 . Intelligence Artificielle

Les intelligences artificielles auront pour objectif de pouvoir remplacer (au moins) un joueur dans le jeu.

5.1 Stratégies

Plusieurs stratégies seront développées selon une difficulté et une complexité croissantes.

5.1.1 . IA Aléatoire

L'IA aléatoire est la plus simple et la plus rudimentaire des Ias possibles. Dans notre cas, nous la concevons comme suit : l'IA va dans un premier temps parcourir la liste de toutes ses unités et va à chaque fois choisir aléatoirement si elle va bouger cette unité, puis (si elle le peut) si elle va attaquer avec cette unité (à chaque fois avec une position aléatoire). Ensuite, l'IA va parcourir la liste de ses bâtiments et à chaque fois qu'elle tombe sur un bâtiment pouvant créer des unités (le QG ou les Camps d'Entrainements) elle créera une unité au hasard parmi celles qu'elle peut créer (« aucune unité » étant également un choix possible).

5.1.2 . IA Heuristique

L'IA heuristique, contrairement à l'IA aléatoire ne peut pas se baser sur le hasard pour établir une stratégie. En effet, cette IA doit utiliser les informations fournies par l'état de jeu afin de prendre une décision sur la démarche à suivre dans le but de gagner la partie. Dans notre cas l'IA heuristique se dirigera vers l'unité ou le bâtiment ennemi le plus proche. Selon les cas de figures, l'IA heuristique devra attaquer si sa porter d'attaque lui permet ou se déplacer vers l'ennemi. Pour tester l'efficacité de notre IA nous l'avons fait jouer 100 parties contre l'IA heuristique :

```
int winCounter_H = 0;
int winCounter_R = 0;
int nb_turn = 100;
```

Figure 12: nombre de partie de tests random vs heuristic

Nous constatons que l'IA heuristique gagne à 98 % contre l'IA random :

```
heuristic win rate : 98%
random win rate : 2%
```

Figure 13: résultat des affrontements heuristique contre random

L’IA finale peut gagner car la map est petite et il peut se diriger aléatoirement vers le QG adverse et le prendre. Parfois des seg fault peuvent survenir car il y a des sûrement des cas que nous n’avons pas traité.

5.1.3 . IA Avancé

Pour l’IA avancée, l’idée est de permettre à l’ia de pré-calculer des states complet et de lui fournir une méthode de notation de ces states selon différents critères (position des entités de jeu, de leur valeur, leur type, etc). Pour ce faire nous pouvons utiliser la méthode précédente mais en calculant les scores des positions et en les sommant. Les scores des states seront sauvegardés et comparés de manière à garder la valeur maximum. Cette ia nécessite l’implémentation d’une méthode de rollback permettant de revenir en arrière et d’annuler une action réalisée (ce qui implique une modification de l’engine), afin de parcourir entièrement l’arbre des états. De manière générale, la profondeur de parcours des arbres d’états est la plus grande contrainte pour ce type d’ia car très coûteuse en mémoire vive et donc en performances. Cela est particulièrement vrai pour notre jeu qui possède un certains nombre de règles et d’instances dans un état, ce qui démultiplie les calculs pour un état et diminue les performances.

5.1.4 . Conception Logicielle

Le diagramme des classes des intelligences artificielles est présentée en figure 12.

La classe principale est la classe AI qui est la classe abstraite des IA et qui possède les méthodes nécessaires.

La classe fille RandomAI représente la classe permettant d’instancier les IAs aléatoires et implémentent les méthodes de sa classe mère. Elle contient une méthode et des attribues supplémentaire qui permettent de gérer l’aspect aléatoire.

La classe fille HeuristicAI représente la classe de l’IA heuristique. Elle implémente les méthodes virtuelles de la classe mère AI. Afin de faire ses calculs de chemin nous luis avons associé une classe PathMap qui a comme objectif de calculer la future position d’une unité et de choisir la meilleure possibilité. Nous utilisons une classe Position qui contient un score et une position (x, y). Le score varie avec les méthode de la classe PathMap.

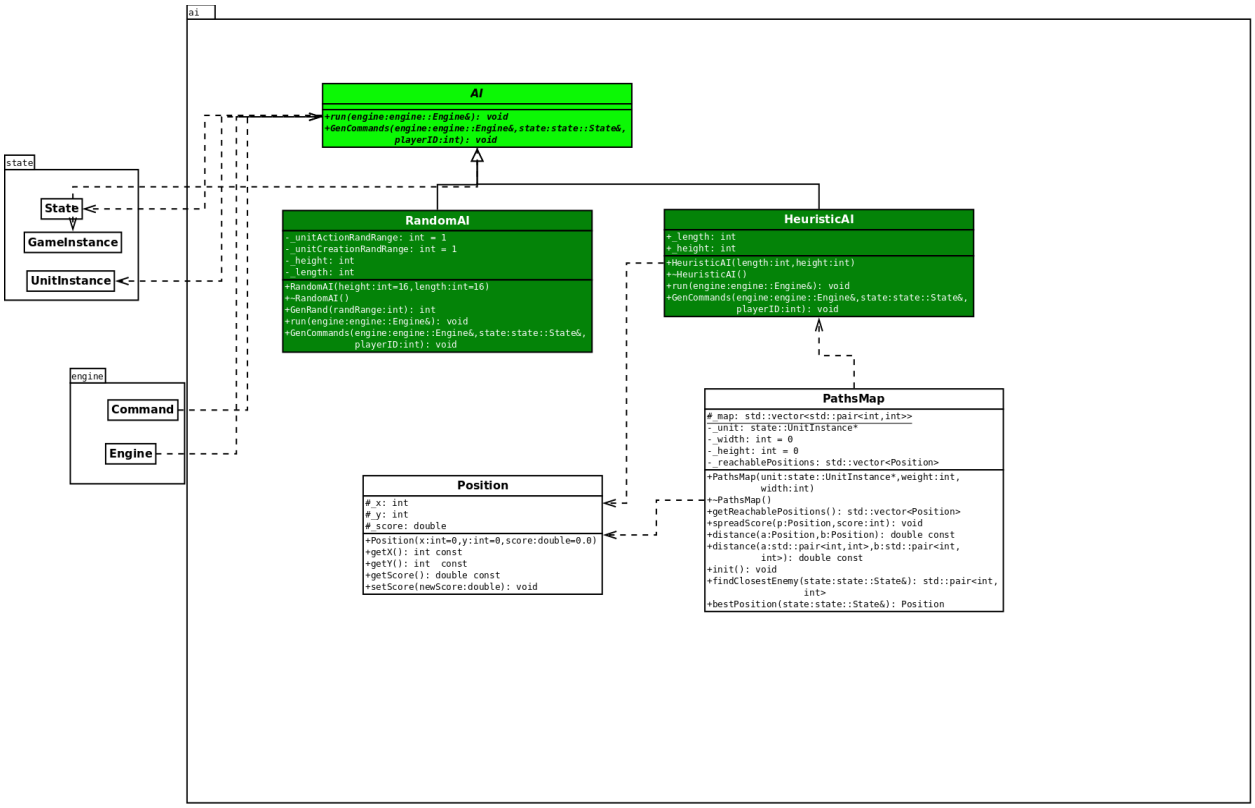


Figure 14: ai.dia

6 . Modularisation

Notre objectif ici est de placer le moteur de jeu sur un thread, puis le moteur de rendu sur un autre thread. Le moteur de rendu est nécessairement sur le thread principale (contraire matérielle), et le moteur du jeu est sur un thread secondaire. Nous avons deux type d'information qui transite d'un module à l'autre : les commandes et les notifications de rendu.

6.1 Organisation des modules

Plusieurs stratégies seront développées selon une difficulté et une complexité croissantes.

6.1.1 . Répartition sur différents threads

Notifications de rendu. Ce cas est problématique, car il n'est pas raisonnable d'adopter une approche par double tampon. En effet, cela implique un doublement de l'état du jeu (un en cours de rendu, l'autre en cours de mise à jour), ce qui augmente de manière significative l'utilisation mémoire et processeurs, ainsi que la latence du jeu. Nous nous sommes donc tournés donc vers une solution avec recouvrement entre les deux modules, en proposant une approche qui le minimise autant que possible. Pour ce faire, nous ajoutons un cache qui va « absorber » toutes les notifications de changement qu'émet une mise à jour de l'état du jeu. De façon continue, le moteur graphique actualisera le contenu du state et afficher ce qui est dedans.