Projet Logiciel Transversal

Advance Wars

par REN Xiaoyu et WONGWANIT Nicolas

Figure 1 : Un tour de jeu - Advance Wars

Table des matières

1. Présentation Générale

1.1. Archétype

Le projet proposé est un jeu suivant l'archétype du jeu de stratégie en tour par tour "Advance Wars".

Figure 2 : Exemple d'un tour de jeu (Joueur 1 en rouge)

1.2.Règles du jeu

- Le but du jeu est de vaincre l'adversaire, soit en détruisant la totalité de son armée, soit en capturant le Q.G.. Dans ce but, le joueur dispose de nombreuses unités de combat, avec des capacités variées (infanteries, tanks, artilleries, …). Note : dans certains modes de jeu, il n'y a pas de Q.G., et/ou les joueurs commencent la partie sans troupe.
- Le jeu se déroule sur un plateau à maille carré. Il existe des cases ayant des propriétés diverses (plaines, montagnes, villes, usines, …). Les joueurs jouent à tour de rôle. Lors du tour d'un joueur, il est possible de déplacer chaque unités du joueur une seule fois, et ensuite d'attaquer une unité ennemie adjacente. Il n'est pas possible d'attaquer PUIS de se déplacer. Il existe des exceptions, qui sont les unités d'attaque à distance et les unités de transport : unités d'attaque à distance ne peuvent que que se déplacer ou bien attaquer, et les unités de transport ne peuvent tout simplement pas attaquer.
- Le joueur peut également produire de nouvelles unités, tant qu'il possède le capital nécessaire et des usines libres. Chaque usine ne peut produire qu'une unique unité par tour de jeu. De plus, l'unité produite ne peut pas agir durant le tour de production.

- Il est à noter que seul les infanteries peuvent capturer les villes, usines et Q.G. Capturer des villes et usines permet d'acquérir des fonds et de produire plus d'unités.
- Chacune des unités possèdent bien sûr leurs forces et leurs faiblesses. Il est donc nécessaire de déployer ses unités avec soin, en tenant compte de ces avantages et du terrain, afin de triompher de son adversaire.

1.3. Ressources

Des sprites seront utilisés pour réaliser le rendu graphique du jeu. Chaque spite aura la taille d'un carré 16x16 pixels, représentant une case du terrain, une unité, ou un bâtiment.

Figure 3 : Sprites des unités utilisés dans le jeu

Figure 4 : sprites des bâtiments

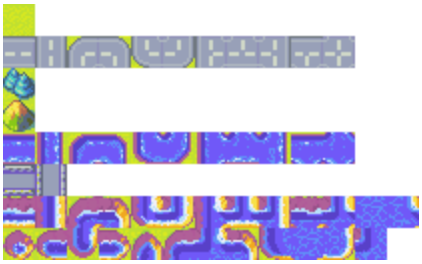

Figure 5 : sprites des cellules terrain.

2. Description et conception des états

2.1. Description des états

Un état du jeu est formé par une matrice de cellules (Cell) 2D représentant le plateau de jeu. Sur ce plateau sont placés des bâtiments (Building), immobiles, et des troupes pouvant appartenir à différents joueurs, qui sont mobiles.

Les bâtiments et les troupes possèdent tous comme propriété leur coordonnées sur le terrain avec les attributs (x,y).

2.1.1. Etat des cellules (Cell)

Chaque cellule possède un attribut énuméré CellType qui décrit un type de terrain, qui offre divers avantages au troupes sur ce terrain.
Les cellules peuvent, de plus, être associées à un objet Unit et/ou Building.

2.1.2. Etat des bâtiment (Building)

Les bâtiments possèdent les attributs énumérés BuildingType et BuildingTeam.

- BuildingTeam indique quel joueur a le contrôle du bâtiment. BuildingType peut avoir les valeurs NEUTRAL, PLAYER1 ou PLAYER2.
- BuildingType indique le type de bâtiment. Parmi ces différent types, on trouve le type BASE, correspondant au Q.G. (quartier général) du joueur, la ville CITY, ainsi que les bâtiment de productions FACTORY, AIRPORT et SEAPORT.
- L'attribut *capturePoint* donne le temps avant capture du bâtiment par un joueur différent du propriétaire.

2.1.3. Etat des troupes (Unit)

Les unités possèdent deux attributs énumérés UnitTeam et UnitType, ainsi que quatre autres attributs de type entier.

- UnitTeam : correspond à l'affiliation des troupes, PLAYER1 ou PLAYER2. Il n'y a pas de troupes neutres.
- UnitType : le type d'unité. Il existe 18 type d'unités (voir figure 4).
- *health* : indique la vitalité de l'unité. Lorsque la vitalité atteint zéro, l'unité est détruite.
- *ammo* : la quantité de munition possédé par la troupe. Lorsque cette valeure atteint zéro, l'unité ne peut plus attaquer.
- *fuel* : représente l'énergie de l'unité. Si cette valeure est nulle, l'unité ne peut plus se déplacer.
- *vision* : le champ de vision de l'unité en mode de jeu avec brouillard de guerre.

2.1.4. Etat des joueurs (Player)

Les joueurs possèdent une liste de leurs unités et bâtiment contrôlés à travers les conteneurs *unitList* et *buildingList* de pointeurs uniques.
Un attribut PlayerID entier est aussi associé à chaque joueurs.

2.1.5. Etat général (State)

L'ensemble des éléments de jeu sont réunis dans une classe état (State) à travers un vector de pointeurs vers Unit et Building, et une matrice de Cell.

2.2. Conception logiciel

Le diagramme des classes d'état est présenté ci-dessous (figure X). Nous pouvons remarquer que les classes Player et State contiennent les classes Unit et Building à travers des conteneurs de pointeurs uniques.

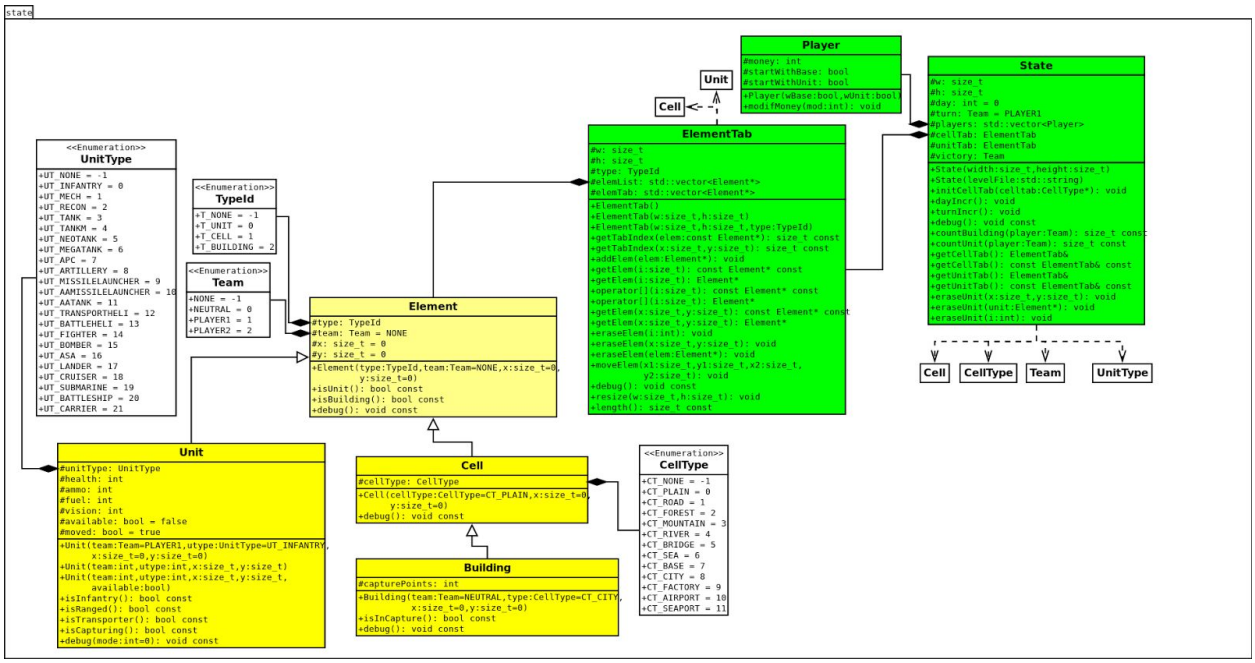

Figure 6 : Diagramme des classes d'état

3. Modèle du rendu

3.1. Description des classes

3.1.1. Tile

La classe Tile décrit un rectangle à travers une abscisse x, une ordonnée y, une largeur w et une hauteur h. Tile définit une tuile qui se trouve dans un fichier ressource, ou une tuile graphique qui s'affiche à l'écran.

3.1.2. TileSet

La classe TileSet permet de trouver la tuile correspondant à un Element, à travers ses classes filles : StateTileSet, CellTileSet, BuildingTileSet et UnitTileSet.

3.1.3. Surface

Surface est la classe permettant de réaliser l'interface entre les classes SFML et les classes que nous avons créés. Cette classe possède la méthode draw, qui permet d'afficher graphiquement le contenu de la classe.

3.1.4. Layer

Cette classe contient une classe Surface et une classe Tileset qui la correspond. Deux classes filles en héritent : BoardLayer, dont le rôle est de gérer des éléments du jeu, et StatusLayer, dont le rôle est d'afficher des informations par dessus les BoardLayer, et ainsi réaliser une interface utilisateur.

3.1.5. Render

C'est la classe principale du moteur de rendu. Render contient une référence vers State, un StatusLayer, et trois BoardLayer, correspondant aux terrain, aux bâtiments et aux unités.

3.2. Conception logicielle

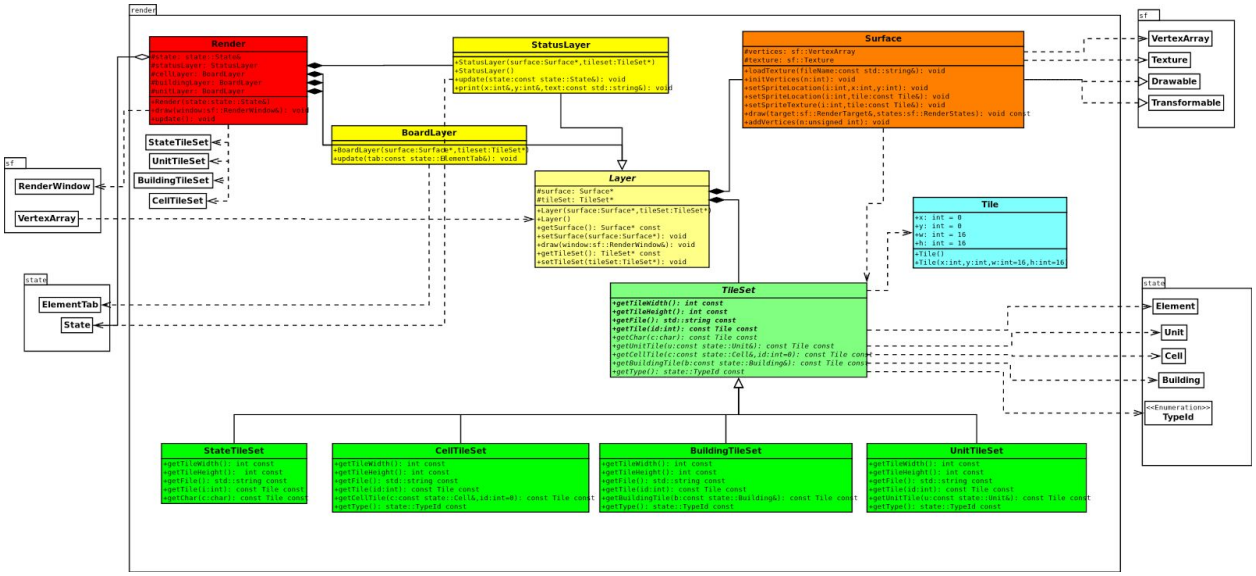

Figure 7 : Diagramme des classes de rendu

4. Conception du moteur de jeu

4.1. Description des classes

Le moteur du jeu est formée par les différentes commandes qui contrôlent des opérations nécessaires.

4.1.1. Commande Capture

Nous pouvons contrôler notre unité à capturer un bâtiment. Soit ce bâtiment est la base d'un joueur, ce joueur perdra le jeu. Soit ce bâtiment est capturé, le propriétaire de unit possédera ce bâtiment.

4.1.2. Commande Attack

Nous pouvons contrôler notre unité à attaquer l'unité d'autre joueur. Le command va changer la vitalité de l'unité par calculer des valeurs de capacité attaque d'unité et armure du but.

4.1.3. Commande Supply

Nous pouvons recharger une unité pour avoir la capacité de mouvement.

4.1.4. Commande Repair

Nous pouvons réparer une unité pour regarnir la vitalité de l'unité.

4.1.5. Commande Load

Nous pouvons utiliser le véhicule à transporter notre unités.

4.1.6. Commande Select

Nous pouvons choisir notre unité pour les opérations suivantes.

4.1.7. Commande Move

Nous pouvons bouger notre unité.

4.1.8. Commande Destroy

Soit la vitalité de l'unité est inférieur ou égale à zéro, ce unité va être mort.

4.1.9. Engine

C'est le coeur du monteur. Elle stocke les commandes dans une std::map avec clé entière. Ce mode de stockage permet d'introduire une notion de priorité : on traite les commande dans l'ordre de leur clés, de la plus petite à la plus grande. Lorsqu'une nouvelle époque démarre, ie lorsqu'on a appelé la méthode update() après un temps suffisant, le moteur appelle la méthode execute() de chaque commande, incrémente l'époque, puis supprime toutes les commandes.

4.2. Conception logicielle

Le diagramme des classes de commande est présenté ci-dessous (figure X). Nous pouvons remarquer que les classes commande.

.

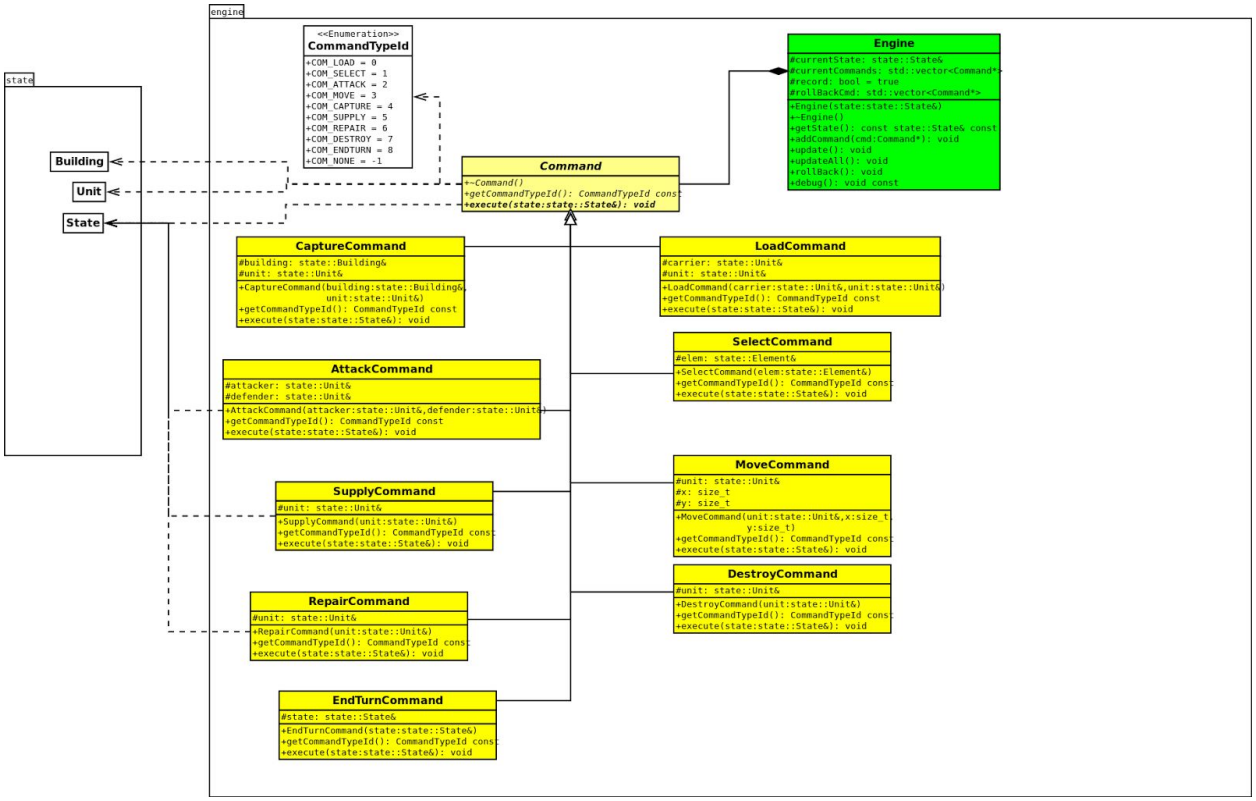

Figure 8 : Diagramme des classes du moteur de jeu

5.Intelligence Artificielle

5.1.Stratégies

5.1.1.Intelligence aléatoire

Cette stratégie est la même pour tous les personnages : à chaque fois, on choisit au hasard un élément sur la carte(soit une bâtiment ou un unité) et faire les actions au hasard.

5.1.2.Intelligence basée sur des heuristiques

Nous proposons ensuite un ensemble d'heuristiques pour offrir un comportement meilleur que le hasard, et avec une chance de résoudre le problème complet(ie, bouger l'élément ou attaquer l'ennemi).

D'après choisissons un élément au hasard, nous vont calculer les distances par rapport aux éléments ailleurs. Rappelons que dans la conception de notre moteur de jeu, l'élément va vérifier le groupe d'autre élément et faire l'action que nous avons choisi:

- S'ils sont dans différent groupe, l'élément choisi va attaquer l'élément détecté(s'il est un unité) ou capturer l'élément(s'il est un bâtiment) .
- S'ils sont dans le même groupe, l'élément choisi va vérifier son état et fait repérer lui même(si dans mauvais état) ou fait transporter l'élément détecté.

La plupart des heuristiques proposées sont mis en oeuvre en utilisant des cartes de distances vers un ou plusieurs objectifs.

5.2. Conception logiciel

Le diagramme des classes pour l'intelligence artificielle est présenté en Figure 9.

Classes AI. Les classes filles de la classe AI implémente différentes stratégies d'IA, que l'on peut appliquer pour un personnage :

- RandomAI : Intelligence aléatoire
- HeuristicAI : Intelligence heuristique

PathMap. La classe PathMap permet de calculer une carte des distances à un ou plusieurs objectifs. Dans un groupe, chaque élément va détecter des élément ailleurs le plus proche de lui et vérifier son groupe et sa sorte(un unité ou un bâtiment). Puis, chaque élément va faire des différentes action par rapport à cette distance.

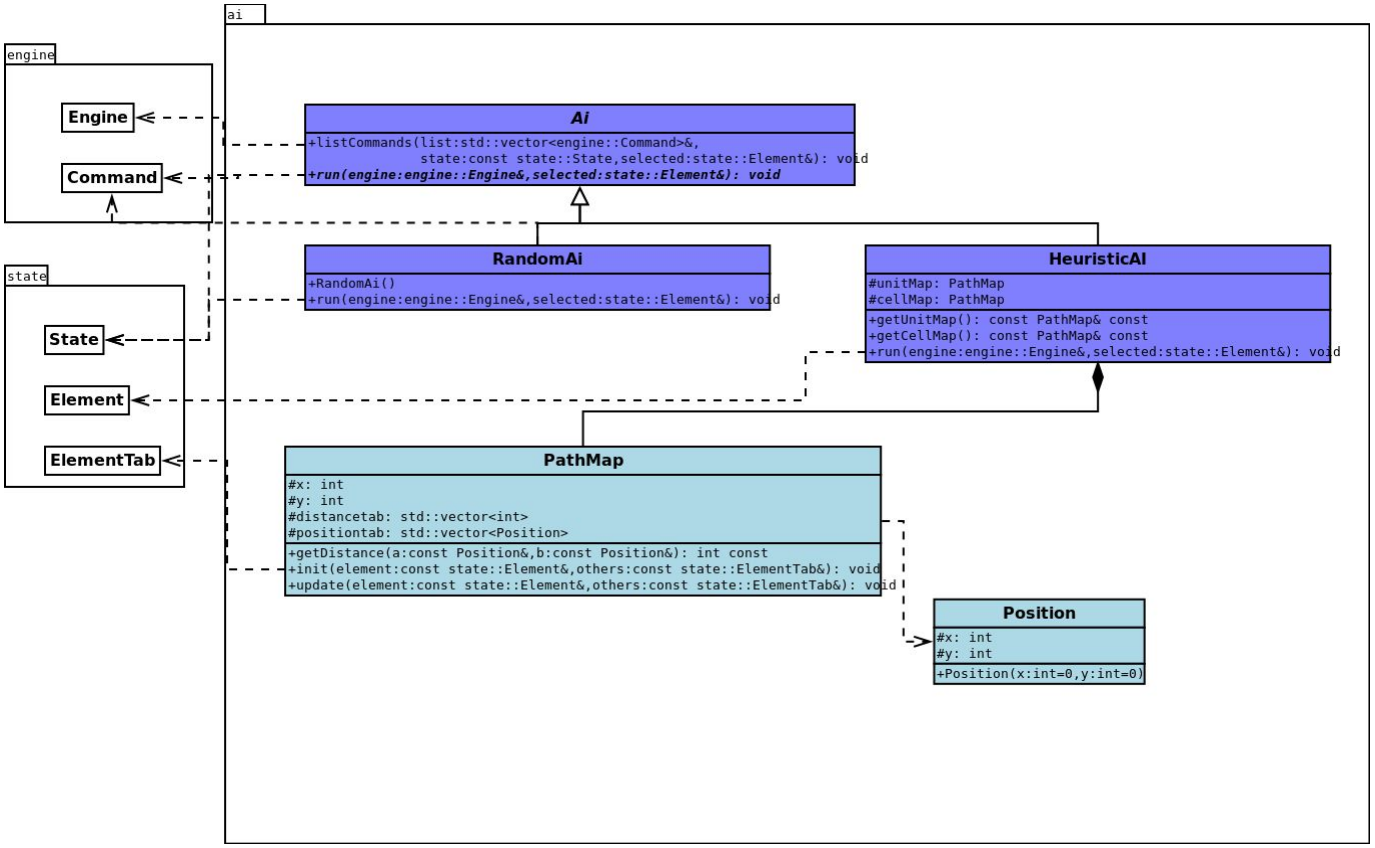

Figure 9 : Diagramme des classes d'intelligence artificielle

6. Service Web

6.1. API Requêtes

Lancer le serveur :
.../server/listen *(8080)*

6.1.1. Rassemblement des joueurs

GET : obtenir le nom d'un joueur / la liste des joueurs
curl http://localhost:8080/player/<id>

Entrée		Pas de données d'entrée

Sortie	Cas <id> existant	Status OK type:"object" properties:{ "name":{type:string}, }, required:["name"]

	Cas <id> négatif	Statut OK
		
		type: "array",
		items:{
		type: "object",
		properties: {
		"name": { type:string },
		},
		},
		required: ["name"]

	Cas <id> inexistant	Statut NOT_FOUND Pas de données de sortie

PUT : ajouter un joueur
curl -X PUT -d ‘{“name”:”<playername>”}’ http://localhost:8080/player

Entrée		type: "object", properties: { "name": { type:string }, }, required: ["name"]

Sortie	Cas il reste de la place	Statut CREATED type: "object", properties: { "id": { type:number,minimum:0,maximum:4 }, }, required: ["id"]
	Cas plus de place	Statut OUT_OF_RESOURCES Pas de données de sortie

POST : modifier un joueur
curl -X POST -d ‘{“name”:”<playername>”}’ http://localhost:8080/player/<id>

Entrée		type: "object", properties: { "name": { type:string }, }, required: ["name"]

Sortie	Cas joueur <id> existe	Statut NO_CONTENT Pas de données de sortie
	Cas joueur <id> n’existe pas	Statut NOT_FOUND Pas de données de sortie

DELETE : supprimer un joueur
curl -X DELETE http://localhost:8080/player/<id>

Entrée		Pas de données d'entrée

Sortie	Cas joueur <id> existe	Statut NO_CONTENT Pas de données de sortie
	Cas joueur <id> n'existe pas	Statut NOT_FOUND Pas de données de sortie

6.2. Conception