Projet Logiciel Transversal

Marlyatou DIALLO – Tian HAOMING

Figure 1 civilisation : conquête et pillage

Table des matières

1 Objectif Civilization : Conquête et pillage

1.1 Présentation générale :

Il s'agit d'un jeu au tour par tour destiné à créer sa propre armée de régime, à développer et à occuper des terres. Chaque joueur doit explorer des terres ou piller le château ennemi pour gagner des pièces d'or qui peuvent être utilisées pour acheter différents types d'unités de combat ou construire de châteaux.
A la fin du nombre de tours fixés, le vainqueur sera celui qui a le grand nombre de châteaux laissés et de pièces gagnés.

1.2 Règles du jeu

-Unité de Combat :
Le jeu comprend cinq unités: **infanterie, ingénieurs, archers, cavalerie et lanciers**. Les ingénieurs sont réprimés par l'infanterie et la cavalerie peut réprimer l'infanterie, tandis que la cavalerie peut être réprimée par les archers et les lances. Le lancier et l'archer sont mutuellement répressifs. Toutes les unités auront un bonus d'attaque lorsqu'elles attaqueront des unités qu'elles suppriment.
Au début de la partie, chaque joueur ne possède qu'une seule équipe d'ingénieurs.

-Occupation château et gain d'or :
Chaque équipe d'ingénieurs explore les terres dont certaines contiennent des pièces d'or. Ces pièces d'or explorées par les ingénieurs seront comptées dans le montant total des pièces. Au début de la partie chaque joueur possède un château initial qui rapporte à chaque tour des pièces tant qu'il n'est pas détruit par l'ennemi. Les pièces d'or pourront servir à la construction de châteaux ou à l'achat d'une unité de combat.
Lors de l'attaque d'un château, on peut simplement l'occuper (avoir des pièces à chaque tour et veiller à sa défense) ou le détruire (obtenir les pièces qui valent ce château).

- Ce que le joueur peut faire à chaque tour :
Les joueurs peuvent effectuer quatre actions par tour: attaque, défense, déplacement et construction. Chaque unité de combat ne peut attaquer que deux fois par tour. Il faudra quatre tours pour construire un nouveau château. Le joueur a le droit de passer ce tour.

-Mouvement et vision
Le jeu sera joué sur une carte en forme de damier, chaque unité se déplaçant d'un certain nombre de carrés par tour, le mouvement de l'unité à chaque tour consomme une certaine quantité de pièces d'or.
Au début de l'ouverture, la majeure partie de la carte est une zone inconnue et chaque unité peut déverrouiller les informations de la carte dans le champ de vision.

	infanterie	**ingénieurs,**	**archers**	**cavalerie**	**lanciers**
Distance de vision	**4**	**8**	**6**	**4**	**4**
Distance de mouvement	**2**	**3**	**2**	**6**	**3**

1.3 Conception Logiciel

Système utilisé : machine virtuelle (Virtual box sous Ubuntu)
Environnement de développement utilisé : Eclipse ou Geany
Compilateurs C++ : g++
CMake : cmake
SFML : package libsfml-dev
Logiciel Tiled Map Editor

2 Description et conception des états

2.1 Description des états

Un état du jeu est formée par un ensemble d'éléments fixes (les terrains, les pièces d'or générées sur les terrains) et un ensemble d'éléments mobiles (les pions). Tous les éléments possèdent des propriétés telles que :

-Une Coordonnées (x,y) sur la carte.
-Un code tuile
-un nom
-une puissance d'attaque et de défense pour certains et aussi un niveau de vie

2.1.1 Etat éléments fixes

La carte de jeu est formée par une grille d'éléments de type **Field**, des éléments **GoldBonus** qui sont générés de façon aléatoires sur la grille et aussi des éléments **Castles** qui seront construits au fur et à mesure. Nous allons choisir une grille de 25 cases par 25 cases.
Les différents types de terrains (Field) sont :
-**Terrains non praticables** que nous avons nommé « NPraticable » : ils sont inaccessibles par les éléments mobiles et n'ont aucune influence sur eux. Ce sont :

Les arbres « Tree »
Les murs « Wall »
L'eau « Water »

-**Terrains Praticables** que nous avons nommés « Praticable » : ceux-ci sont accessibles par les éléments mobiles et peuvent avoir de l'influence sur les statistiques du joueur. Ce sont :

-Terre Ferme « Land »: il peut y avoir de l'or caché sur une terre ferme et on peut également construire des châteaux sur elles.
-Les collines « Hill » : on peut construire des châteaux sur une colline mais il ne peut y avoir de l'or cachés sur une colline.

2.1.2 Etat éléments mobiles

Ces éléments sont dirigés par les joueurs que nous avons défini par une classe **Player**. Chaque Joueur a un nom, manipulera des pions et châteaux et peut gagner des espèces d'or en bonus.
Chaque pion que nous avons nommé **UnityArmy** possède un niveau de vie (currentlife), un camp (camp), un champ d'attaque (attack_field), un champ de mouvement (move_field), un statut (status), un identifiant (type_id).
Les statut represente l'etat d'un pion à un instant donné :

-Selectionné « SELECTED» : dans le cas ou le pion est selectionné pour faire une action
-Disponible « AVAILABLE » : dans le cas ou il peut etre selectionné mais n'a pas encore agi.
-Attente « WAITING » : dans le cas ou le pion ne peut etre selectionné car c'est le tour d'un autre joueur ou le pion a effectué toutes les actions qu'il peut faire en un tour.
-Mort « DEAD » : dans le cas ou son niveau de vie est à 0.

Les Pions sont de plusieurs catégories et sont identifiés par un type_id :

-Cavalerie « CAVLRY »
-Archer « ARCHER »
-Ingénieur « INGINEER »
-Lancer « LANCER »

2.1.3 Etat général

Sur l'ensemble des éléments mobiles et statiques, nous avons rajouté :

-Nombre de tours
-Numéro de tours
-Maximum de tours
-Fin : pour indiquer la fin de la partie.

2.2 Conception logiciel

Le diagramme d'état est constitué de plusieurs classes dont :
-Classe Eléments : toutes les classes filles(en bleu marine) issue de la classe mère **Elements** représentent les différentes catégories des éléments mobiles et statiques.
-Classe Conteneur d'éléments : il y'a les classe **State** et **Map2D qui contiennent l'ensemble des éléments.** Map2D est un tableau à deux dimensions d'éléments, par exemple pour contenir la grille des éléments du niveau.
Enfin, la classe State est le conteneur principal, à partir duquel on peut accéder à toutes les données de l'état et contient aussi la liste des joueurs.
-Classe Position : classe qui décrit la position de chaque élément sur la grille.
-Classe Correspondence : classe qui regroupe des sets de double composé d'un code tuile et d'un ID d'éléments.
-**Classe Observable** : la conception de ses classes suit un design patterr Observer. la classe **Observable** dont hérite **State** enregistre des observateurs et les notifie à chaque changement d'etat.
Classe Correspondances : Classe qui regroupe des sets de doublets composés d'un code de tuile et d'un ID d'élément

2.3 Conception logiciel : extension pour le rendu

Les Classes State, Observable et Position seront des classes qui seront inclues dans le rendu et l'entension est render.dia

2.4 Conception logiciel : extension pour le moteur de jeu

Engine.dia

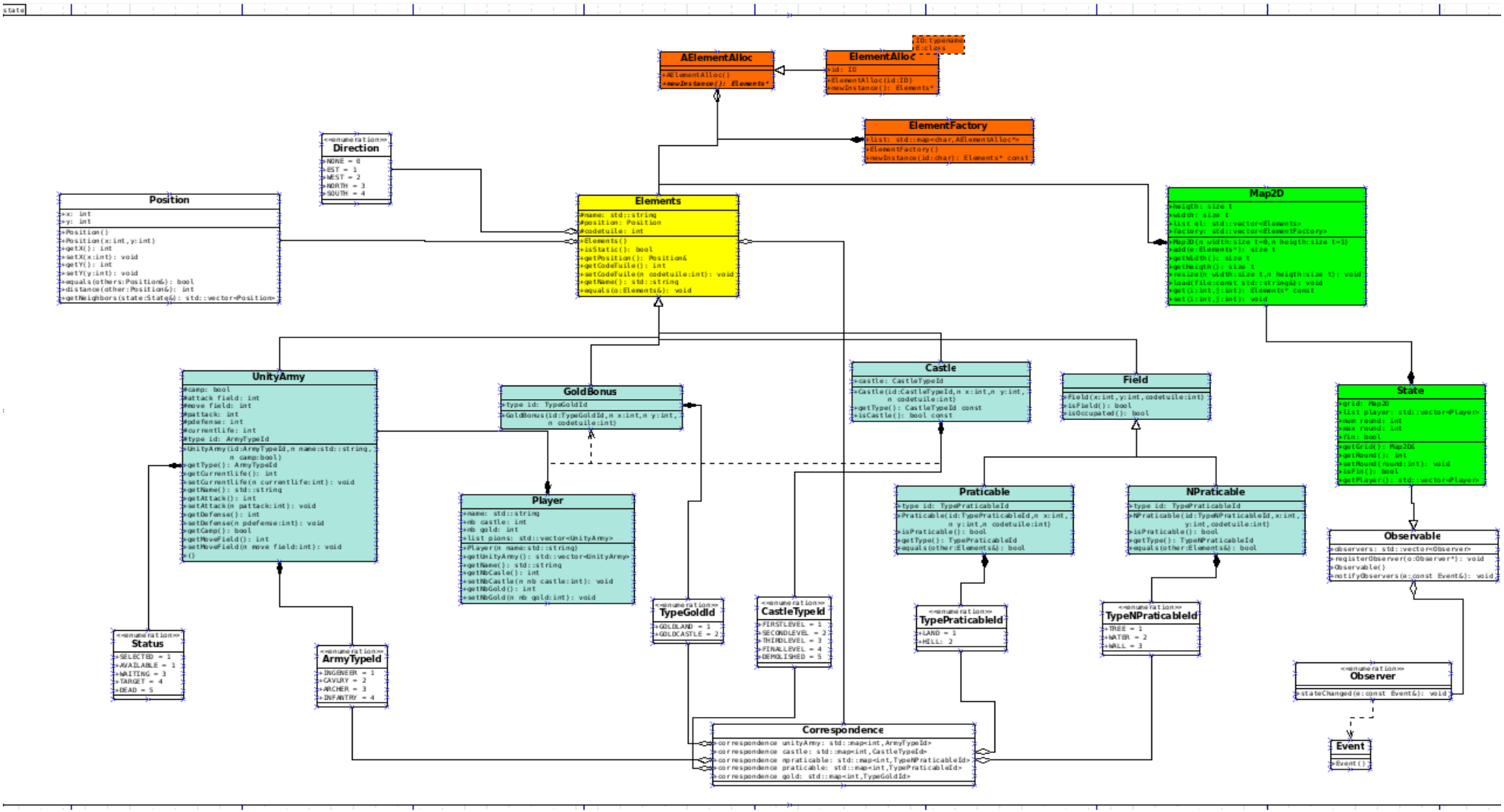
2.5 Ressources

Figure 2: tuile UnityArmy

Figure 3:tuile niveau château

Figure 4:tuile Tree

Illustration 1: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

NB : nous avons apporté beaucoup de modification sur le diagramme d'etat et par conséquent des modifications ont été apporté sur ce qui a été redigé précédement.

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons fait le choix d'un rendu par tuile à l'aide de la bibliothèque SFML.
Nous divisons la scène à afficher en trois surfaces : une surface contenant les éléments statiques Field, une surface pour les éléments mobiles UnityArmy et une surface pour les informations de jeu (statistiques et nom d'un personnage, tour de jeu…).
La grille de Field est créée à partir du fichier texte « map.txt » composé de 25x25 codes de tuiles. La classe Correspondences (dans le package state) possède un tableau de correspondance pour les Praticables Field et un autre pour les NonPraticables Field. Ces std::map associent à chaque code de tuile un ID de terrain spécifique. Par exemple, la première tuile du tileset pour la grille porte le numéro 0 et représente un ensemble d'arbre, et cette association est donc répertoriée dans l'attribut correspondance_P de la classe Correspondences. Pour modifier la grille il suffit donc demodifier les entiers dans le fichier « map.txt » (qui se trouve dans res/). Pour le moment nous avons juste utilisé un tableau contenant des codes tuiles et declarer directement dans le main pour aller plus vite.
La méthode initGrille de la classe State (package state) nous permet de fabriquer pour chaque code tuile du fichier « map.txt » le terrain correspondant, de créer un pointeur unique vers cet objet et de l'ajouter à la grille (tableau à deux dimensions contenant des pointeurs de Field).
Les unityarmy eux ne sont pas créés à partir d'un fichier, pour l'instant ils sont créés par la méthode initUnityArmy de la classe State.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général est présenté sur la figure ci dessous.

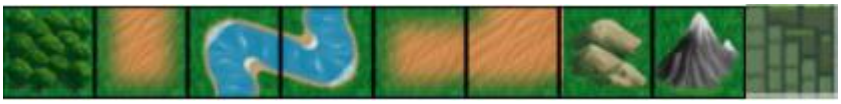
Classe Surface : cette classe possède deux attributs : une texture et un tableau de Vertex (quads) contenant la position des éléments et leurs coordonnées dans la texture. Elle possède les méthodes loadGrille, loadPersonnage et loadCurseur lui permettant d'initialiser ses attributs à partir d'un tableau de Field, d'une liste de UnityArmy et d'un fichier « spritee.png », « unityArmy.png » . La méthode Draw a pour but de dessiner une texture pour ensuite permettre son affichage dans une fenêtre.
Classe TileSet : Cette classe possède plusieurs attributs : un id de type TileSetID, des entiers cellWidth et cellHeight (qui représentent respectivement la largeur et la longueur en pixel d'une tuile) et une chaîne de caractères imageFile (chemin vers un « fichier.png »). L'ID peut prendre différentes valeurs :
• GRIDTILESET
• UNITYARMYTILESET
• INFOSTILESET
Suivant l'ID passé en argument du constructeur de TileSet, les attributs cellWidth, cellHeight et imageFile sont initialisés différemment. De plus, les textures sont chargées depuis le fichier correspondant lors de l'appel au constructeur, et donc une seule fois par TileSet. Les méthodes getCellWidth, getCellHeight et getImageFile permettent de récupérer ces attributs.
Classe StateLayer : Cette classe possède un attribut state qui est une référence à une instance De State. Elle possède également un tableau de pointeurs de TileSet et un autre de pointeurs de Surface. Le but de cette classe est de créer trois surfaces grâce à la méthode initSurface et d'initialiser leur texture. Cette classe est un observateur, elle implémente l'interface Observer pour

être avertie des changements d'état. Elle réagit ensuite en actualisant les textures.
Projet

3.3 Conception logiciel : extension pour les animations

Le diagramme pour les animations sera engine.dia

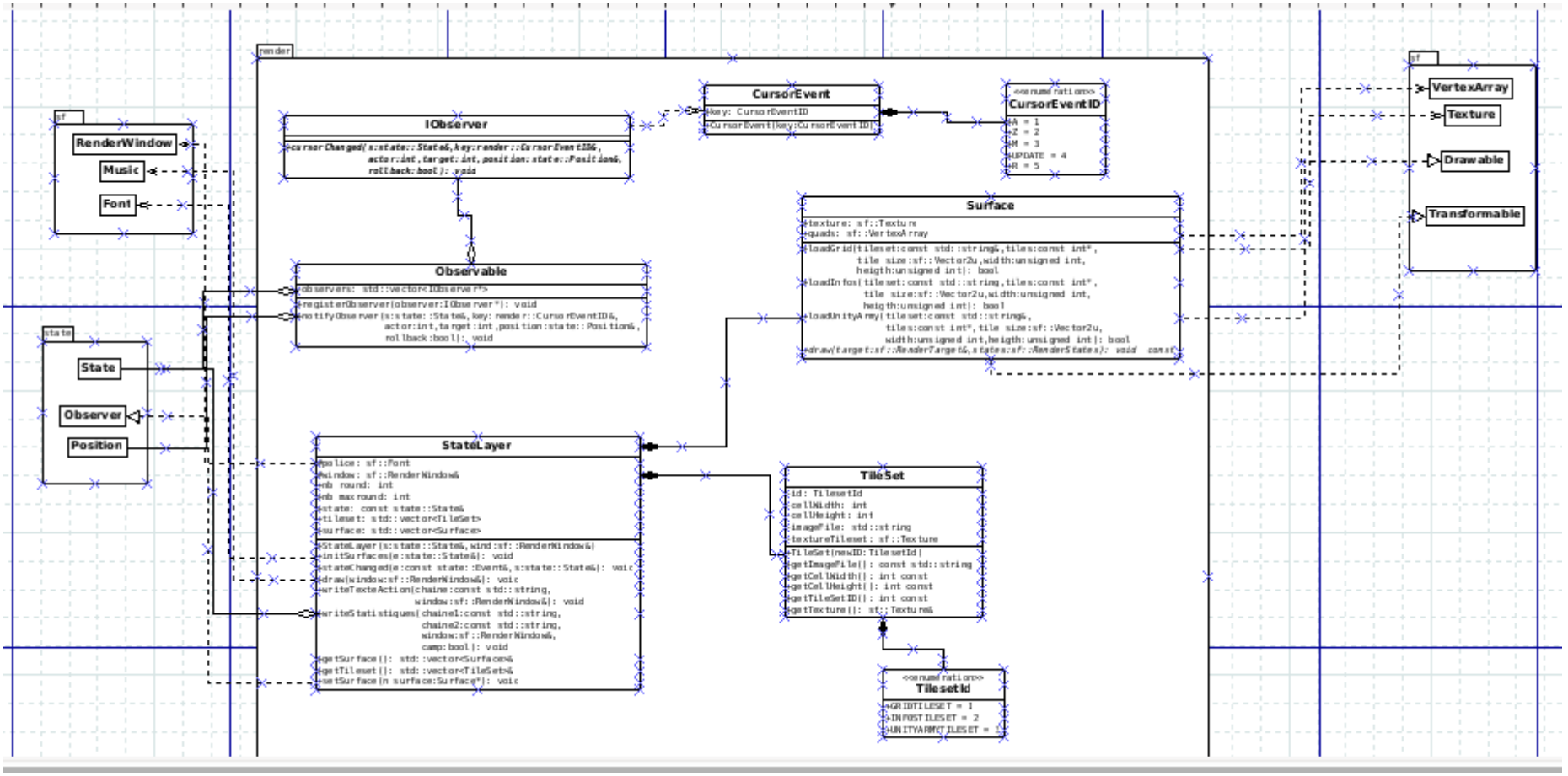
3.4 Ressources



3.5 Exemple de rendu

Nous avons encore des choses à rajouté sur ce rendu mais etant pris par le temps nous allons devoir fournir en attendant cela.

Illustration 2: Diagramme de classes pour le rendu



4 Règles de changement d'états et moteur de jeu

4.1 Horloge globale

Les changements d'état suivent une horloge globale : de manière régulière, on passe directement d'un état à un autre. Il n'y a pas de notion d'état intermédiaire. Ces changements sont calibrés sur le temps qu'il faut pour un élément mobile à vitesse maximale pour passer d'une case à une autre. En conséquence, tous les mouvements auront une vitesse fonction de cet élément temporel unitaire. Notons bien que cela est décorrélé de la vitesse d'affichage : l'utilisateur doit avoir l'impression que tout est continu, bien que dans les faits les choses évoluent de manière discrète. Par exemple, s'il y a 60 images par seconde, et que le temps unitaire est de 200ms, alors pendant un temps unitaire, 12 étapes d'animation seront affichées pour les éléments en mouvement.

4.2 Changements extérieurs

Chaque action effectuée modifie l'état de jeu

. Le choix de l'unityarmy sélectionné et des actions effectuées est défini par des commandes.
Les commandes du joueur sont provoquées par une pression sur une touche.
. Deplacer le curseur : flèches « haut/bas/gauche/droite ».
. Selectionner une unity : « Enter » (lorsque le curseur encadre le personnage).
. Deplacer une unity : flèche « haut/bas/droite/gauche » (l'unityarmy doit avoir été sélectionné au préalable).
. Attaquer : « A » puis déplacer le curseur avec les flèches sur la case d'un adversaire puis « Enter » pour confirmer l'attaque (le personnage doit avoir été sélectionne au préalable).
. Annuler une attaque : Appuyer sur « N ».
. Terminer le tour d'actions d'un personnage : « Z » (le personnage doit avoir été sélectionné au prealable).

4.3 Changements autonomes

En début de tour, toutes les unityArmy de type INGINEER d'un joueur possèdent le statut DISPONIBLE. Ils peuvent donc tous être SÉLECTIONNÉS. Lorsqu'une unityArmy est SÉLECTIONNÉ par un joueur il peut effectuer selon les cas au moins une des 3 actions listées ci-dessous :

- Se déplacer
- Attaquer
- Construire
- Terminer ses actions (son tour)

Une unityarmy ne peut se déplacer que d'une case par une case (sur des terrains praticables et inoccupés) . Il possède des points de mouvements qui correspondent au nombre de déplacement maximum qu'il lui est possible de faire. Lorsque tous ses points de mouvement ont été utilisés, il lui est impossible de continuer son déplacement.
Une unityarmy ne peut attaquer une autre que lorsque celle-ci se trouve dans son champ d'attaque et appartient au camp adverse (les attaques alliées ne sont pas autorisées).
Lorsqu'une unityarmy termine ses actions, son statut devient WAITING. Cela signifie qu'il ne pourra plus être SÉLECTIONNÉ (et ne pourra donc plus effectuer d'actions) avant le prochain tour du joueur.

L'action « Terminer son tour d'action » doit obligatoirement être effectuée par chaque unityarmy encore actif à chaque tour du joueur (même lors du dernier tour lorsque tous les adversaires ennemis sont morts). Il existe 4 enchaînements d'actions possibles :

· Attaquer directement (lorsque cela est possible) ce qui termine automatiquement le tour d'un personnage
· Effectuer un ou plusieurs déplacements , attaquer puis terminer son tour automatiquement
· Effectuer un ou plusieurs déplacements, construire puis terminer son tour automatiquement
· Effectuer un ou plusieurs déplacements puis terminer son tour manuellement
· Terminer son tour directement sans avoir effectuer aucune autre action

Le tour de jeu d'un joueur est terminé lorsque tous ses personnages sont en WAITING.
C'est alors le tour du joueur adverse.
Lorsqu'une unityarmy est attaqué par un ennemi, il tente toujours une contre-attaque (à moins d'avoir été achevé durant la première attaque).

Si une unityarmy perd tous ses points de vie, son statut évolue et prend la valeur DEAD.

Si toutes les unityArmy d'un joueur meurent avant qu'on n'atteigne le nombre de tour maximum fixés. et qu'il ne possède ni de châteaux, ni d'espèces d'or pour acheter des unityarmy la partie est terminée a la fin du tour adverse.

Si les unityArmy des deux joueurs sont encore vivants à la fin de la partie, le joueur qui gagne est celui qui a construit le plus de châteaux et qui a une quantité plus grande de pièces d'or.

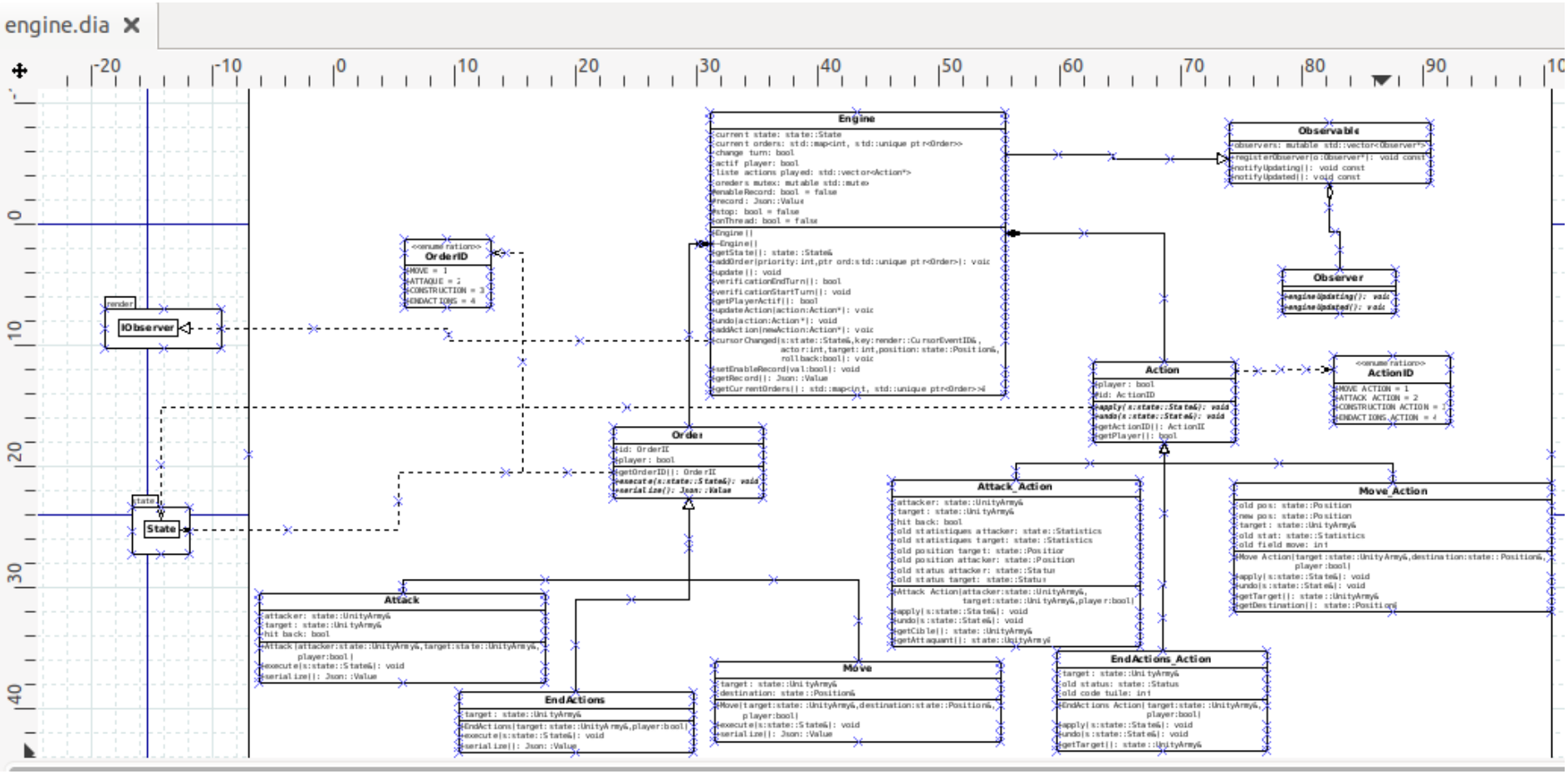
4.4 Conception logiciel

Le diagramme des classes pour le moteur de jeu (« engine.dia ») est présenté sur la figure ci-dessous.
Le moteur de jeu repose sur le Design Pattern Order.
Classe Engine : C'est le cœur du moteur. Elle permet de stocker les commandes dans une std::map avec clef entière (avec « addOrder »). Ce mode de stockage permet d'introduire une notion de priorité : on traite les commandes dans l'ordre de leur clef (de la plus petite à la plus grande). Lorsque la méthode « update » est appelée, le moteur appelle la méthode « execute » de chaque commande puis supprime toutes les commandes une fois executées. La méthode « verificationStartTurn » change le joueur actif, réinitialise les points de mouvement de tous les unityarmy du joueur et procède à la récupération de la quantité d'or des unityarmy ayant construit des châteaux. La méthode « verificationEndTurn » incrémente le nombre de tours si tous les unityarmy restants du joueur actif sont en WAITING et déclare la partie terminée si tous les personnages du joueur adverse sont MORTS avant que le nombre de tours fixés soit atteint.
La classe Engine a été étendue pour permettre une exécution dans un thread séparé.
Classes Commandes : Les classes Attack, Move et EndAction, Construction héritant de la classe Order, possèdent chacune une méthode « execute » qui fait effectuer à un personnage l'action correspondante.
Classes Actions : Les classes Attack_Action, Move_Action et EndActions_Action, ConstructionAction_Actionn héritant de la classe Action, possèdent chacune une méthode « apply » qui change l'état de la même manière que le ferait une Commande, mais possèdent également un méthode « undo » qui remodifie l'état mais en annulant les changements opérés par l'action en question, grâce aux attributs de chaque Action qui correspondent aux changements que provoque l'action.

4.5 Conception logiciel : extension pour l'IA

4.6 Conception logiciel : extension pour la parallélisation

Illustration 3: Diagrammes des classes pour le moteur de jeu



5 Intelligence Artificielle

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

5.1 Stratégies

5.1.1 Intelligence minimale

5.1.2 Intelligence basée sur des heuristiques

5.1.3 Intelligence basée sur les arbres de recherche

5.2 Conception logiciel

5.3 Conception logiciel : extension pour l'IA composée

5.4 Conception logiciel : extension pour IA avancée

5.5 Conception logiciel : extension pour la parallélisation

6 Modularisation

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

6.1 Organisation des modules

6.1.1 Répartition sur différents threads

6.1.2 Répartition sur différentes machines

6.2 Conception logiciel

6.3 Conception logiciel : extension réseau

6.4 Conception logiciel : client Android

Illustration 4: Diagramme de classes pour la modularisation