Projet Logiciel Transversal

Hikaru GOTO – Benjamin HYON

Table des matières

1 Objectif

1.1 Présentation générale

Le but ce projet est la création d'un jeu de stratégie tour par tour basé sur Age of Empires : Age of Kings version DS. Le jeu se

concentra sur l'aspect de la bataille et la création d'armée, avec une arborescence des technologies grandement simplifiée par rapport au jeu publié.

1.2 Règles du jeu

Présenter ici une description des principales règles du jeu. Il doit y avoir suffisamment d'éléments pour pouvoir former entièrement le jeu, sans pour autant entrer dans les détails . Notez que c'est une description en « français » qui est demandé, il n'est pas question d'informatique et de programmation dans cette section.

1.2.1 But du jeu

Le principe du jeu est simple : une civilisation à développer, en technologie et en armée, pour but de détruire la civilisation adverse.

1.2.2 Principaux aspects du jeu

- Le joueur doit choisir une civilisation, sous forme d'un choix de **races**. Plusieurs races sont disponibles :

Les elfes, les orcs, les nains ou les humains.

Toutes les races forment les mêmes unités : les villageois, l'unité d'infanterie, de cavalerie et de distance.

Cependant, chaque race a une **affinité particulière** :

Par exemple, les orcs forment une infanterie avec une attaque accrue, pendant que les elfes forment des archers plus habiles.

- Le jeu se déroule sur une carte prédéfinie avec différentes **zones**, telles que le marais, la forêt, la montagne, la plaine, la route ou la mer. Chacune possédant des attributs différents, la connaissance du terrain est un facteur clé pour la victoire.
  - Marais : malus de déplacement, passage impossible pour les unités cavalières.
  - Forêt : malus de vision*, bonus de défense.
  - Montagne : malus de déplacement, bonus de vision* et de défense.
  - Plaine : 0 attribut.
  - Route : bonus de déplacement.
  - Mer : Aucune troupe autorisée.

  **(la vision influx sur la distance de tir des archers)*

- Sur cette carte est présente différentes **ressources**, de type constante et instantanée : leur repérage et leur prise de contrôle sont les aspects les plus importants du jeu.
  - Ressource instantanée : Les ruines, les trésors et les animaux permettent d'obtenir des ressources dès la prise du contrôle de la zone. Cependant, après l'obtention de la ressource, la ressource disparaît.
  - Ressource constante : les champs et les gisements d'ors ; ils nécessitent leur prise de contrôle à l'aide d'une construction dédiée (un moulin pour les champs, une mine pour l'or). Après contrôle de la zone, une quantité constante de ressource sera obtenue chaque tour.

1.2.3 Déroulement du jeu

- Les joueurs jouent un jour chacun leur tour. Chaque unité pourra se déplacer et/ou attaquer une fois par jour.
- Tous les joueurs commencent par une unité d'infanterie, et une unité de villageois.
- Les villageois sont les seules unités capables de construire. Seuls les centres-villes peuvent créer les villageois.
- Le nombre de construction et d'unités autorisé varie en fonction de l'avancée technologique et du nombre de ressources

constantes contrôlées par le joueur.

- La partie se termine lorsque tous les centres-villes adverses sont détruits.

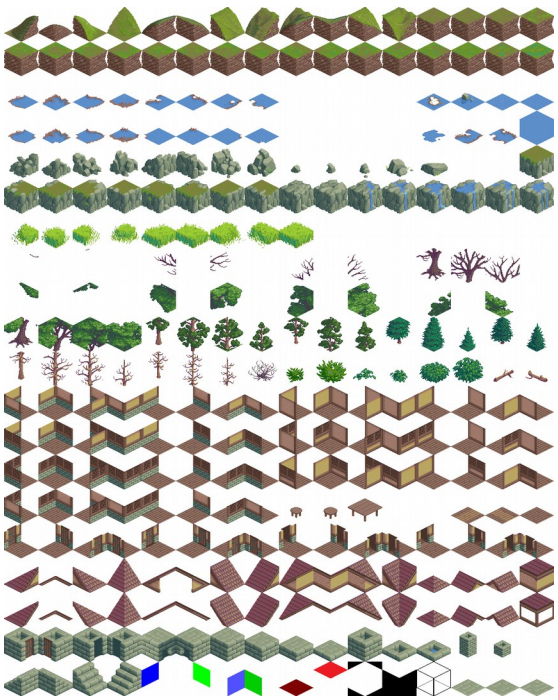
1.3 Conception Logiciel

Présenter ici les packages de votre solution, ainsi que leurs dépendances.

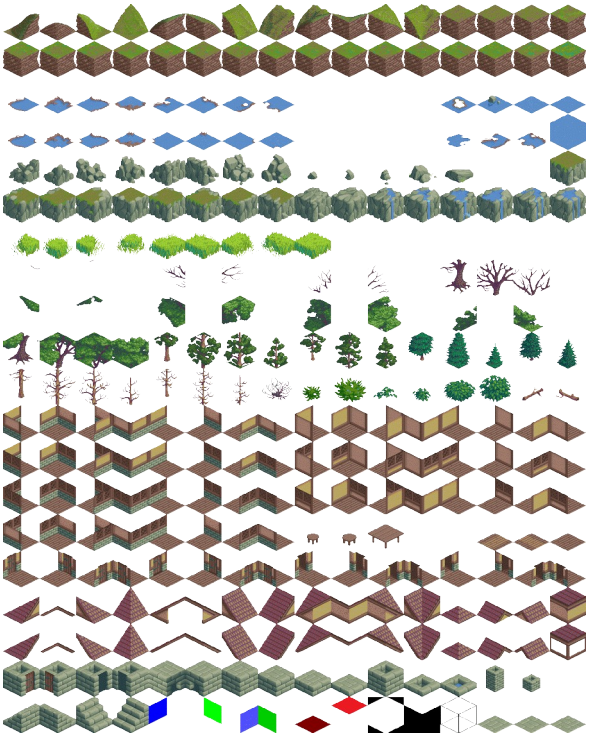
La carte est d'une taille 32x32 pixels, un pixel déterminant une zone.

1.3.1 Tiles pour le terrain :

Version original trouvée sur google image :



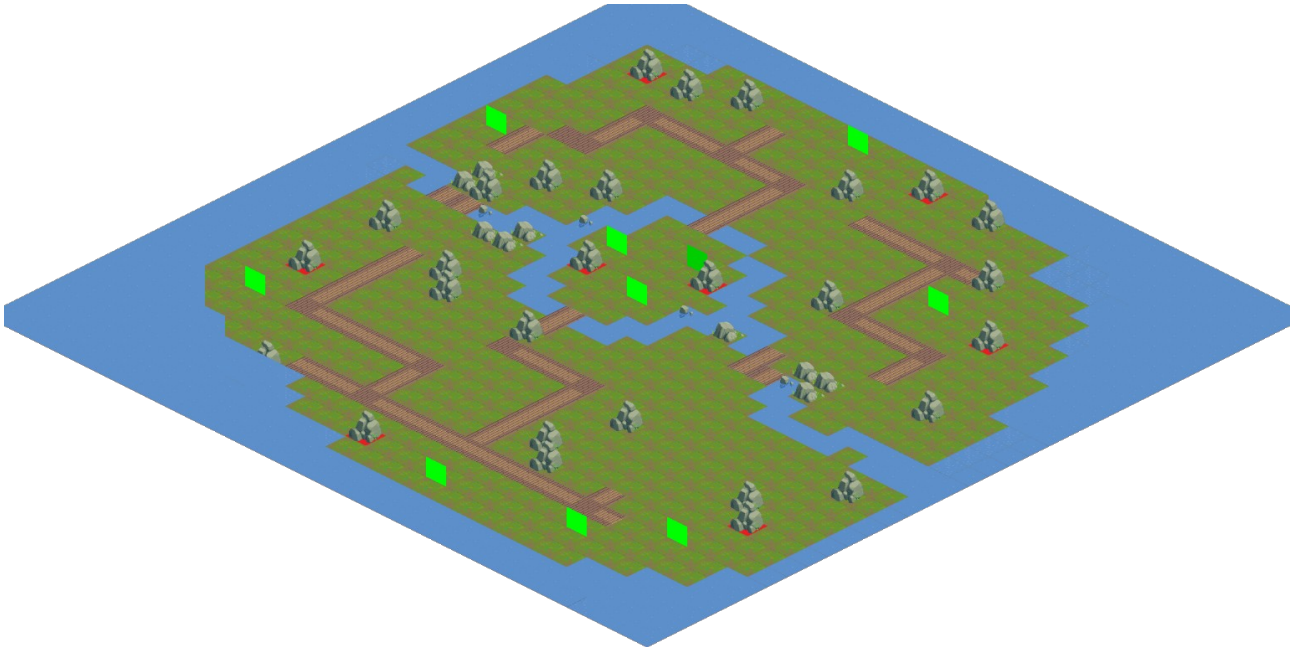
Version transparente utilisée dans le projet :



1.3.2 Tiles pour les ressources :

Création de tiles sur blender (à confirmer)

1.3.3 Tiles pour les unités :
Voici notre prototype de notre première carte : InrushIsland



-Les carrés verts désignent une zone de champ.
-Les carrés rouges désignent une zone de gisement d’or.

2 Description et conception des états

L'objectif de cette section est une description très fine des états dans le projet. Plusieurs niveaux de descriptions sont attendus. Le premier doit être général, afin que le lecteur puisse comprendre les éléments et principes en jeux. Le niveau suivant est celui de la

conception logiciel. Pour ce faire, on présente à la fois un diagramme des classes, ainsi qu'un commentaire détaillé de ce diagramme. Indiquer l'utilisation de patron de conception sera très appriécé. Notez bien que les règles de changement d'état ne sont pas attendues dans cette section, même s'il n'est pas interdit d'illustrer de temps à autre des états par leur possibles changements.

2.1 Description des états

L'état du jeu est composé par un ensemble d'éléments d'environnements et d'éléments d'unités propre aux joueurs. Chaque élément possède une coordonnée (X, Y) et une valeur entière pour les différencier.

Pour le moment, les éléments d'environnements comportent les terrains, et les éléments d'unités propre aux joueurs les constructions et unités.

2.1.1 Etat : éléments d'environnements

a) La carte

La carte du jeu est pour l'instant une grille de taille 64x64, où tous les éléments des états seront stockés. Nous avons opté pour une structure de type liste pour la facilité de mise en place. Voici comment sera organisé notre gestion de la carte :
Les 64 premières valeurs de la liste seront les positions de l'abscisse X allant de 0 à 63, et la position Y=0.
Puis, de la 65ème valeur de notre liste sera la position (X=0, Y=1).
Ainsi, sans passer par un vecteur de vecteur, nous pourrons identifier les positions de nos éléments sur la carte, tout en gardant la syntaxe (X, Y) pour les éléments pour une meilleure visualisation.

La carte comporte les attributs suivants :

- Une liste de type « terrain », permettant de stocker en mémoire tous les terrains de la map (l'instanciation de cette liste sera lors du démarrage d'une partie)
- Une liste de pointeurs pointant sur des « GameObject », permettant la mise en mémoire de l'état actuel des objets présents sur la carte
- Une liste de « Property », qui sera une liste à double champ permettant d'identifier un élément statique avec ses propriétés,
- Une liste de « Property », qui sera une liste à double champ permettant d'identifier un élément dynamique, ici les unités, avec ses propriétés.
- La taille X et Y de la map (pour le moment, elle est fixe de taille X = 64 et Y = 64)

a) Le Terrain

Les terrains sont les « zones » mentionnées dans la première partie du rapport, avec les « montagnes, marais, forets » etc.
Un terrain possède les attributs suivants :

- Un ID d'objet et une position, permettant sa disposition sur la carte du jeu,
- Un « TerrainType », permettant d'identifier le type de zone qu'on a affaire,
- Un mouvement cost, permettant de déterminer l'existence d'un chemin sur le terrain

Sa taille est fixe : (1,1), et sont indépendants des autres terrains voisins.

2.1.2 Etat : éléments propres au joueur

a) GameObject

Un gameobject est un ensemble d'objets qu'un joueur peut posséder.
Chaque GameObject possède les attributs suivants :

- Un ID d'objet et d'une position, permettant l'insertion de l'objet sur la map,
- Un booléen is_static pour déterminer si c'est un objet statique ou dynamique, la valeur qui nous permettra ensuite d'identifier dans quelle liste de propriété il faut aller chercher pour obtenir ces caractéristiques.
- Un booléen is_destroyed, permettant de savoir si l'unité a été détruite ou pas (sa barre de vie descendant à 0 ou inférieur à 0 lors d'un « overkill »)
- Un player_id, permettant l'identification de l'objet,
- Un health_bar, montrant la vie actuelle de l'objet,
- Un attribut « Property » qui lui détermine ses caractéristiques.

La « Property » d'un objet comporte :

1. Un champ permettant d'identifier l'objet créé
2. Un champ de défense
3. Un champ d'attaque
4. Un champ de vie max

Ensuite, un GameObject se diffère en 2 sous ensemble d'objets, les objets statiques, et les objets dynamiques.

- Objets statiques

Les objets statiques sont les différentes constructions que le joueur peut créer, tels que les centres-villes ou les tours de gardes.
Ils possèdent un attribut « can_attack », qui détermine si la construction possède le pouvoir d'attaquer ou non (exemple : tour de

garde peut tirer à distance pour se protéger contre les unités ennemis).
Le BuildingType détermine le type bâtiment qu'il s'agit, ce qui influencera les propriétés de la construction.

- Objets dynamiques

Les objets dynamiques sont les unités que peuvent créer le joueur, tels que les villageois, ou les archers.
Ils possèdent un attribut mouvement_range, qui détermine le nombre de cases que l'unité peut se déplacer en un tour.
Le UnitType détermine le type d'unité formé, ce qui influencera ses propriétés.

Sa taille est de (1,1) fixe. Lors du rajout des châteaux dans le jeu, il sera constitué de 4 « building» différents disposés côte à côte.

2.2 Conception logiciel

Voici le diagramme de classes d'état, mettant en évidence les liens entre les différents états du jeu. (Voir Illustration 1 page 8)
En vert sont les classes pour l'état de la carte, avec sa classe IngameState permettant l'initialisation et sa mise à jour.
En bleu est la classe pour l'état d'environnement, avec sa classe « énumération » pour les différents types de terrains.
En jaune est la classe pour l'état des objets, avec ses classes filles en jaune foncé pour les objets statiques/dynamiques. Chaque classe fille possède une classe « énumération » permettant l'identification des objets créés.

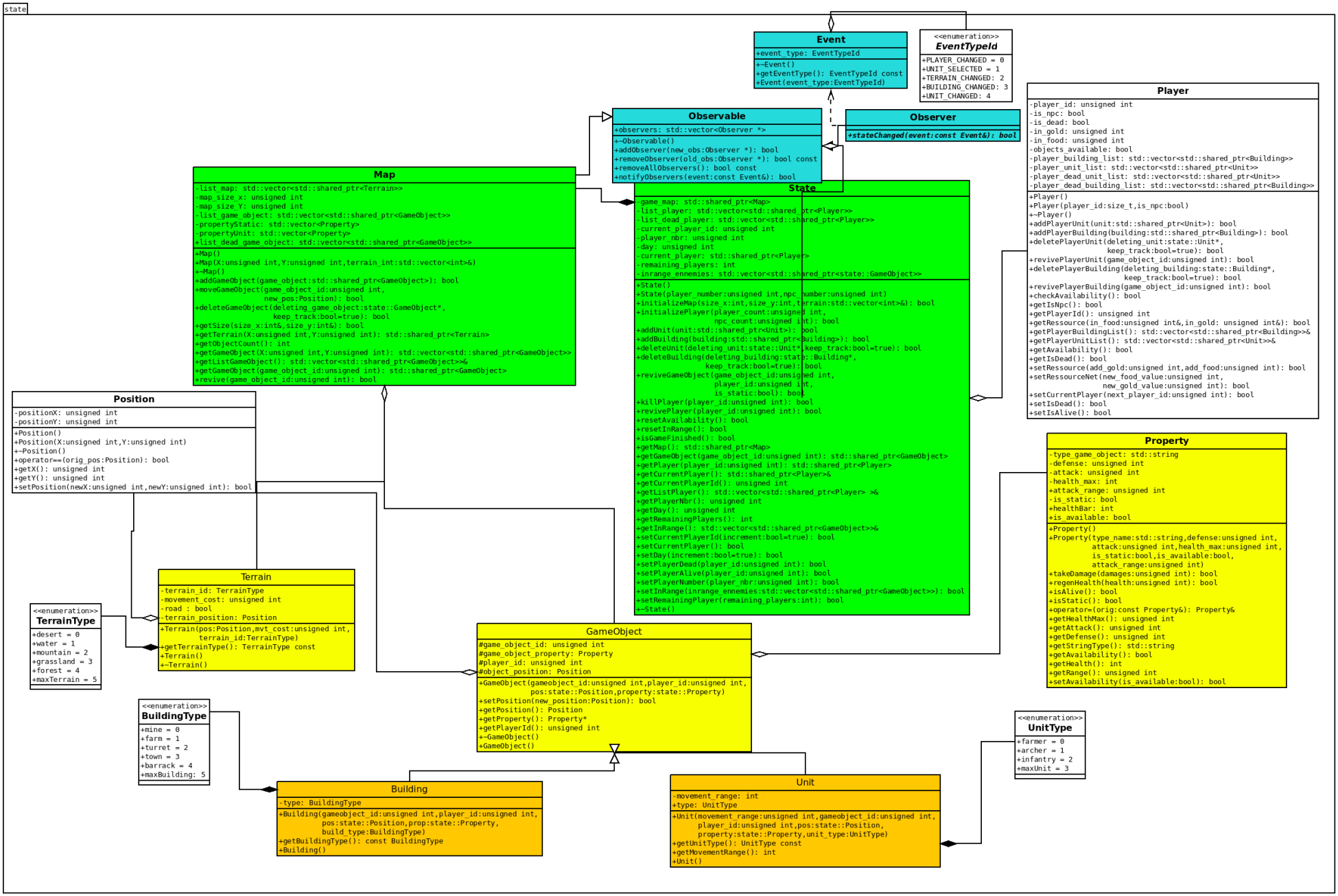
Chaque élément possède donc une position, et chaque objet possède une propriété propre à lui.

2.3 Conception logiciel : extension pour le rendu

2.4 Conception logiciel : extension pour le moteur de jeu

2.5 Ressources

Illustration 1: Diagramme des classes d'état



3 Rendu : Stratégie et Conception

Présentez ici la stratégie générale que vous comptez suivre pour rendre un état. Cela doit tenir compte des problématiques de synchronisation entre les changements d'états et la vitesse d'affichage à l'écran. Puis, lorsque vous serez rendu à la partie client/serveur, expliquez comment vous aller gérer les problèmes liés à la latence. Après cette description, présentez la conception logicielle. Pour celle-ci, il est fortement recommandé de former une première partie indépendante de toute librairie graphique, puis de présenter d'autres parties qui l'implémente pour une librairie particulière. Enfin, toutes les classes de la première partie doivent avoir pour unique dépendance les classes d'état de la section précédente.

3.1 Stratégie de rendu d'un état

Pour le rendu d'un état, nous avons opté pour un rendu très facile à implémenter, avec le minimum requis :
Il faut prendre en compte les conditions suivantes :

- Le terrain, après être initialisé, reste constante pendant toute la partie,
- Les unités peuvent se mettre sur un bâtiment,
- Deux unités ne peuvent pas se positionner sur la même tuile.

Pour cela, on compte d'abord séparer la création des terrains et des objets. Les terrains sont donc initialisés au début, en tant que 1 er layer de la carte, à l'aide d'une matrice de niveau prédéfinie contenant les tuiles à utiliser.
Ensuite, lors d'une création d'un objet de jeu, on actualise la carte en rajoutant cette objet, grâce à ses positions X,Y, obtenues depuis le state.
L'ordre de création d'objet est : objets statiques d'abord, puis ensuite les unités. Ainsi, en gardant des tuiles transparentes, on peut simplement superposer les tuiles des unités pour pouvoir les placer sur une construction.
D'autre part, nous avons choisi de créer une carte en 2D isométrique, pour rendre un aspect visuel de 3D.
Pour cela, un changement de repère a été nécessaire :

$$Xiso = Xcarté - Ycarté$$

En ce qui concerne l'horloge, l'actualisation dépend de la décision du joueur. Ainsi, le rafraîchissement de la carte s'effectue après chaque modification de l'état du jeu.

3.2 Conception logiciel

Le diagramme des classes pour le rendu général est présenté en figure 6.

La classe DrawManager récupère et traite les informations de l'état en cours. Elle contient un objet DrawLayer qui sert d'interface avec la SFML.
DrawLayer est parente de OnMapLayer qui n'affiche que des tiles, sur différents layer : le terrain, puis les bâtiments, puis les unités, et est assez modulaire pour que l'on puisse rajouter d'autres layer puisque tout est transformé en vector de DrawElement par DrawManager, et OnMapLayer ne sait afficher que des vectors de DrawElements.

Lorsque l'état du jeu est modifié et qu'il y a une répercussion visuelle, celui-ci notifie DrawManager.

DrawManager récupère l'état courant, et pour chaque layer va déduire la position et le numéro de tile à afficher. Pour chaque élément ainsi réduit à une position et un numéro de tile, il fait appels à l'objet OnMapLayer correspondant.

La classe PlayerAction gere les actions du joueur, notamment lorsque c'est son tour de jouer. C'est pour cela qu'il se trouve dans le render.
Le joueur peut lancer des actions dans le moteur. Pour que le résultat soit rapide, on fait un engine.execute, mais cela c'est par la suite retourné contre nous lors du multi-threading. En effet avant lorsque l'on pushait une commande et l'exécutais, le moteur mettais a jour le state et donc grâce a l'observer le rendu. Par la suite cela à été supprimé. De plus tant que le joueur n'a pas fini son tour on est bloqué dans le playerAction et donc on bloque le thread en multi-threading.

3.3 Conception logiciel : extension

pour les animations

3.4 Ressources

Comme notre jeu est en 2D isométrique, et que notre implémentation des tuiles nécessite des textures de mêmes formes que les vertices créées, nous ne pouvons plus utiliser les ressources précédemment trouvées.
Voici les nouvelles ressources provisoires pour le rendu :

Les 5 terrains présents dans le jeu : (de la gauche) le désert, la mer, la montagne, la plaine et la forêt.

- Vielle version :

- nouvelle version :

Les 5 constructions disponibles dans le jeu : (de la gauche) la mine, le moulin, la tour, la ville et le centre-ville.

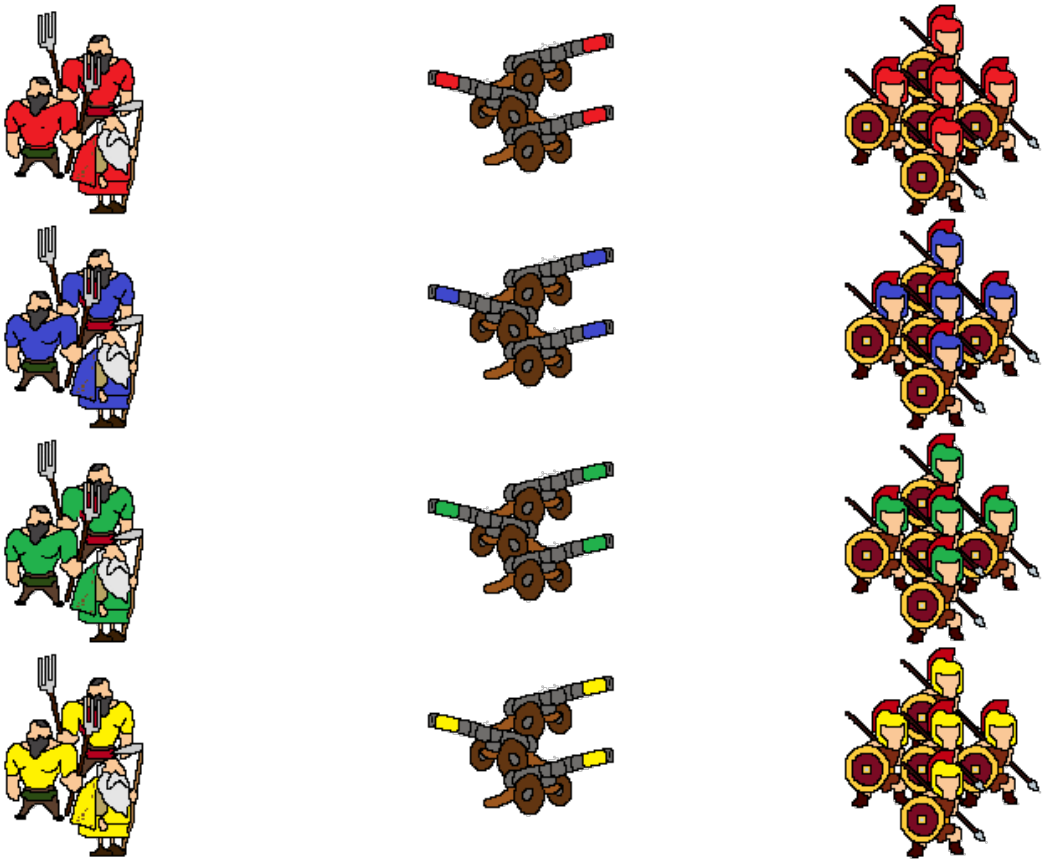
- Vielle version :

- nouvelle version :

Les 3 unités du jeu : (de la gauche) les villageois, l'archer et l'infanterie.

- Vielle version :

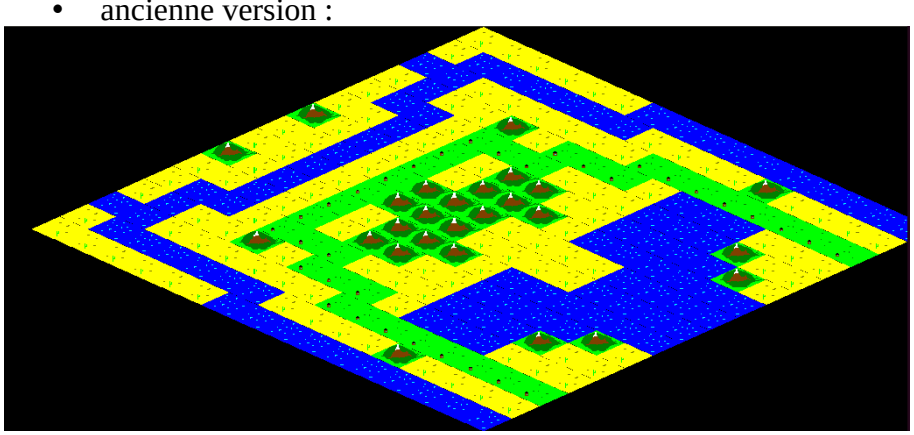
- nouvelle version :



3.5 Exemple de rendu

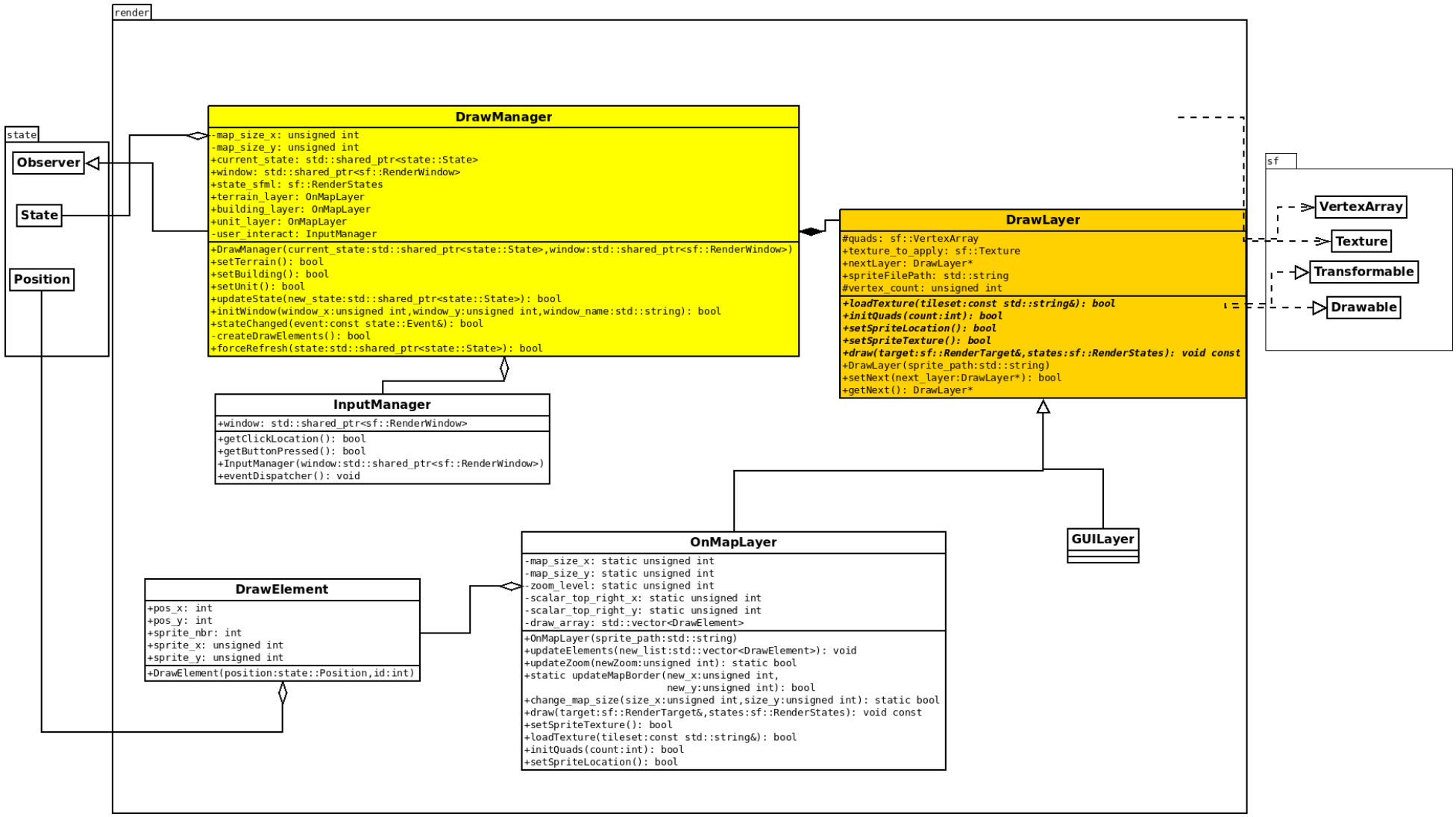
Rendu d'une carte comportant un terrain :

- ancienne version :



- nouvelle version :

Illustration 2: Diagramme de classes pour le rendu



4 Règles de changement d'états et moteur de jeu

Dans cette section, il faut présenter les événements qui peuvent faire passer d'un état à un autre. Il faut également décrire les aspects lié au temps, comme la chronologie des événements et les aspects de synchronisation. Une fois ceci présenté, on propose une conception logiciel pour pouvoir mettre en œuvre ces règles, autrement dit le moteur de jeu.

4.1 Horloge globale

De manière générale, l'état n'est pas modifié de manière autonome : il dépend toujours d'une commande extérieure.
Notre jeu se base sur la gestion d' unités et de bâtiments. Leur créations, leur déplacements leur agencement aux combats. Ainsi, la synchronisation automatique entre les états et les rendus ne sont pas tant importants. Pour le moment, seuls les commandes extérieurs déclenchent une mise à jour du rendu. L'horloge globale n'a donc pas de fréquence déterminée.

En multi-threads, le rendu est actualisée 60 fois par seconde (60fps).

4.2 Changements extérieurs

Le joueur peut effectuer plusieurs commandes sur les objets :

- La création d'un objet, un bâtiment ou une unité,
- Le déplacement d'une unité,
- L'attaque vers un objet ennemi à l'aide de son unité.

Pour mettre en place ces actions, le moteur de jeu nécessite un agencement de différentes commandes.

- En ce qui concerne la création d'objets, elle ne peut pas créer un bâtiment sur un autre bâtiment, ou encore elle ne peut pas créer un bâtiment que le jeu ne reconnaît pas. De même, la création d'objet coûte, ainsi il faut gérer les ressources détenues par les joueurs.
- Pour le déplacement, il faut prendre en compte les différentes mobilités des unités, et s'assurer qu'il n'y a pas de blocage en terme de terrain (eau impénétrable) ou d'autres unités.
- Enfin, pour l'agencement de l'attaque, elle se base sur la distance d'attaque des unités, et bien sûr du joueur au quel il est affecté. (le friendly fire est désactivé).

Pour cela, le moteur de jeu possède les agencements suivants :

1. Agencement des ressources : permet de mettre à jour les ressources détenues par les joueurs. Les revenues sont reçus au début du tour, et les dépenses sont mis à jour à chaque dépense.
2. Agencement des déplacements : permet de déplacer une unité vers une autre position. Elle vérifie si la distance est atteignable par rapport à la mobilité de l'objet.
3. Agencement des attaques : permet de mettre en évidence les cibles attaquables par l'unité considérée.
4. Agencement des dégâts : permet de mettre à jour les dégâts infligées par l'unité à la cible choisie au préalable.
5. Agencement de la création d'objet : permet de créer des bâtiments et des unités en fonction de leur position et des ressources disponibles du joueurs.

4.3 Changements autonomes

Pour le moment, il n'existe pas de changements autonomes.

4.4 Conception logiciel

Le diagramme des classes pour le moteur du jeu est présenté en Figure 7. 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.

Classe Engine :
But : gérer toutes les commandes.
Son travail : grande boucle qui attend une commande extérieure.

Classe Command : (abstraite)
Ces classes permettent de faire le lien entre la commande extérieure et la prise en compte dans l'état et l'affichage.

HandleStartGame :
But : gérer l'initialisation du jeu
Son travail : instancie les joueurs, la carte et les unités « starters ».

HandleTurn :
But : gérer le système de tour/jour
Son travail : mettre à jour le joueur courant du state.

HandleGrowth :
But : prendre en compte l'avancement économique du joueur

Son travail :

- mettre à jour les ressources détenues par les joueurs. Les revenues sont mis à jour au début de la journée, tandis que les dépenses sont mis à jour à pour chaque dépense.

HandleMovement :
But : prendre en compte du déplacement de l'unité considérée.
Son travail:

- Il effectue le calcul de la mobilité en fonction de ses propriétés intrinsèque et des propriétés du terrain.
- Il vérifie la collision entre objets et les obstacles potentiels au cours de son cheminement.

HandleCanAttack :
But : met en évidence les cibles attaquables par l'unité.
Son travail :

- regarde la distance d'attaque de l'unité, et vérifie si des cibles ennemies se trouve sur le champ de tir.
- Renvoie la liste des objets (unités/batiments) attaquables.

HandleDamage :
But : met à jour les points de vies de la cible attaquée par l'unité.
Son travail :

- calcul les points de vie restants de la cible attaquée par l'unité.
- Si points de vie = 0, alors détruit l'objet
- Si l'objet détruit est un centre-ville, alors met à mort le joueur

HandleCreation :
but : prendre en compte l'instanciation d'un bâtiment ou d'une unité.
Son travail :

- vérifie si le bâtiment/l'unité voulant être instancié existe bien,
- vérifie si le terrain est conçu pour les constructions,
- vérifie les collisions possibles des objets pré-définis.

4.5 Conception logiciel : extension pour l'IA

Il n'y a pas eu de modifications notables pour le moteur concernant les IA random/Heuristic.

Pour la Deep AI, nous avons rajouté un système de undo des commandes, permettant ainsi de revenir en arrière et annuler une action.
Pour cela, les commandes possèdent maintenant différents attributs, pour permettre de garder en mémoire les informations utiles pour effectuer une commande, même après que l'état soit changé. Par exemple, pour undo un mouvement d'une unité, undo a besoin de l'unité considérée, et des anciens coordonnées.

Nouvelles méthodes implémentées pour l'extension Deep IA :

- undo : permet d'annuler une commande exécutée auparavant.
- cleanExecuted : permet de mettre à zéro la liste des commandes exécutées. (Est appelé à chaque nouveau jour, pour être sûr que la Deep AI ne undo que ses actions pendant son tour).

4.6 Conception logiciel : extension pour la parallélisation

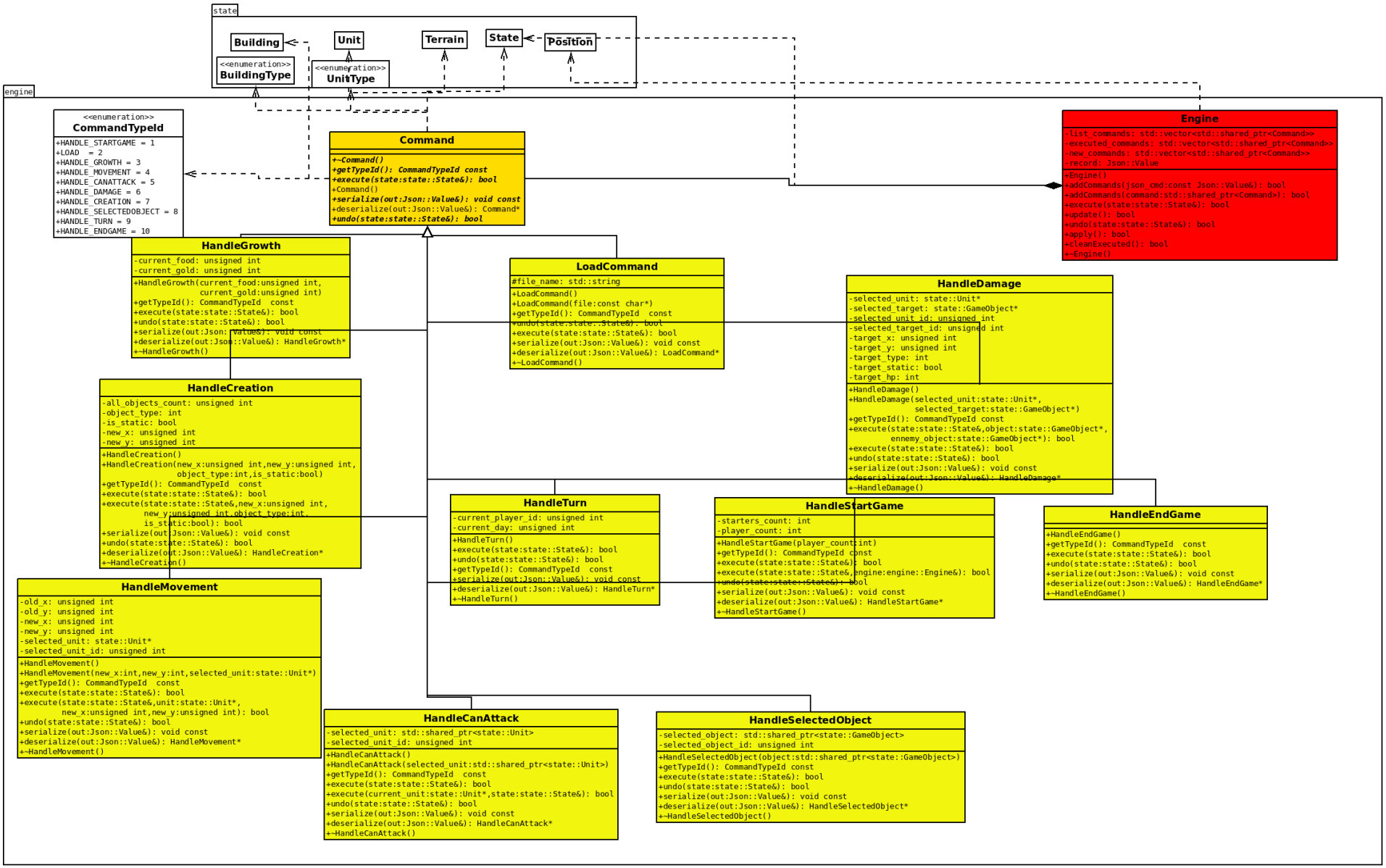
Partie parallélisation :

Une parallélisation complète du moteur a été compliqué, car l'état du jeu devient une source critique, ce qui rendait impossible la parallélisation entre le moteur et les AI/joueurs.
En effet, les AI prennent compte de l'état actuelle pour prendre ses décisions, hors, le moteur modifie l'état directement après exécution d'une commande. Ainsi, on se retrouve dans des situations telles que l'IA joue à la place du joueur, ou encore le moteur execute le HandleTurn alors que le joueur venait de commencer son jour.

Partie Json :

- chaque commande a maintenant une méthode serialize et deserialize.
  Serialize permet de transformer un objet de type Command à un objet Json, et Deserialize permet de faire l'inverse, d'un objet Json à un objet Command.
- Engine possède lui une méthode execute(Json) qui prend un objet Json en tant que commande, pour l'executer.

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

Notre première IA, la IA « bête » consiste à laisser choisir à l'ordinateur des choix aléatoires d'actions possibles.
Il existe quelques contraintes que l'on a dû fixer, pour le bon fonctionnement du jeu :

- une unité attaque si il existe une unité adverse se trouve à coté de lui.
- Une unité construit un bâtiment si elle en a les moyens, idem pour les bâtiments pour construire des unités.

5.1.2 Intelligence basée sur des heuristiques

Ensuite, nous sommes passés à l'IA heuristique, où l'ordinateur va prendre des décisions en fonction de sa situation dans le jeu.
Il est un petit plus robuste et intéressant que l'IA « bête », mais elle reste tout de même loin d'être intelligente.
Voici les conditions qu'elle prend en compte :

- toute création se fait maintenant en fonction des ressources disponibles : plus l'armée possède de l'argent et de la nourriture, plus elle crée des unités/bâtiments.
- L'attaque des archers est maintenant « cohérente », elle attaque à distance si elle le peut.
- Les déplacements des unités en général sont orientés vers le centre-ville ennemi. (sauf les villageois qui restent sur du déplacement aléatoire).

5.1.3 Intelligence basée sur les arbres de recherche

Enfin, nous sommes passés à l'IA Deep. Il se base sur l'heuristic et le random AI, où l'on rajoute la dimension du « pouvoir de simuler les états futurs».

Explication des différents aspects du DeepAi :

- déplacements

Les déplacements sont aléatoires pour les farmers.
Pour les unités offensives, les déplacements seront aléatoires mais contrôlés => l'unité fera 5 mouvements aléatoires que l'on quantifiera en terme de point, selon sa pertinence par rapport à l'état du jeu.
Système de point de déplacement :

+1 si il se rapproche d'une unité ennemie (-1 si il s' éloigne)
+1 si il se rapproche d'un centre ville ennemi (-1 si il s' éloigne)

Étant donnée que les déplacements se font case par case jusqu'à atteindre la distance parcourable maximale de l'unité, les points ci-dessus sont attribués après un déplacement d'une case.
(Un déplacement total donne donc un écart conséquent sur les points, ce qui facilitera le choix du « meilleur » mouvement)

- attaques

Idem pour les attaques, selon si l'unité parvient à attaquer, des points d'attaques seront attribués :
+2 si il parvient à attaquer une unité;
+5 si il parvient à attaquer un batiment ;
+10 si il attaque le centre-ville ;

D'autre part, des points supplémentaires seront attribués selon la situation/l'outcome de l'attaque :

+5 si il parvient à attaquer en ayant une seule cible dans son champ de combat. (limite les regroupements).

+10 si il parvient à détruire l'objet qu'il a attaqué
+50 si il parvient à détruire un centre ville

- créations

Pour les créations, ça se complique un peu.
Un vecteur de points sera mis en place, pour permettre de comparer l'armée du joueur à une armée adverse à la fois.
Le système de points est le suivant :

- +3 si la taille totale de l'armée est supérieure à l'adversaire (-3 si inférieur)
- +1 si nombre d'archers supérieur (-1 si inférieur)
- +1 si nombre d'infanteries supérieur (-1 si inférieur)

Les armées de puissances inférieures à celle du joueur n'étant pas intéressantes, on ne prendra que en compte celles supérieures, auxquelles on sommera leurs points, pour en donnant un nombre négatif, qui refléterai le « retard » du joueur par rapport aux autres joueurs.
Le but du joueur sera alors de faire tendre ce nombre négatif à 0.

Pour le moment, ces 3 critères seront évalués simultanément, ce qui limite les possibilités à 5 « futurs » différents, le meilleur étant celui qui aura le maximum de point sur l'ensemble des 3 critères (somme des 3).
Cependant, on compte améliorer cette AI pour faire en sorte qu'il puisse séparer ces 3 critères, pour pouvoir choisir le meilleur « futur » pour chaque critère séparé, pour en faire un ensemble de « meilleurs » choix.

5.2 Conception logiciel

Le diagramme de classe pour l'AI se trouve à la page suivante :

Classe AI :
Classe mère de toutes les AI que l'on va implementer.

RandomAI :
Première AI qui se repose sur des décisions aléatoires.

5.3 Conception logiciel : extension pour l'IA composée

HeuristicAI :
Deuxième AI qui se base sur des conditions pour choisir les actions les plus « pertinentes ».

5.4 Conception logiciel : extension pour IA avancée

DeepAI :
Dernière AI qui se base sur un système de points pour choisir ses actions.

<u>Ses attributs :</u>

#depth : le nombre de fois que l'IA execute son tour ;
- best state : l'id de l'état futur optimal ;
-update_count : l'id de l'état actuel ; (0 à depth -1)

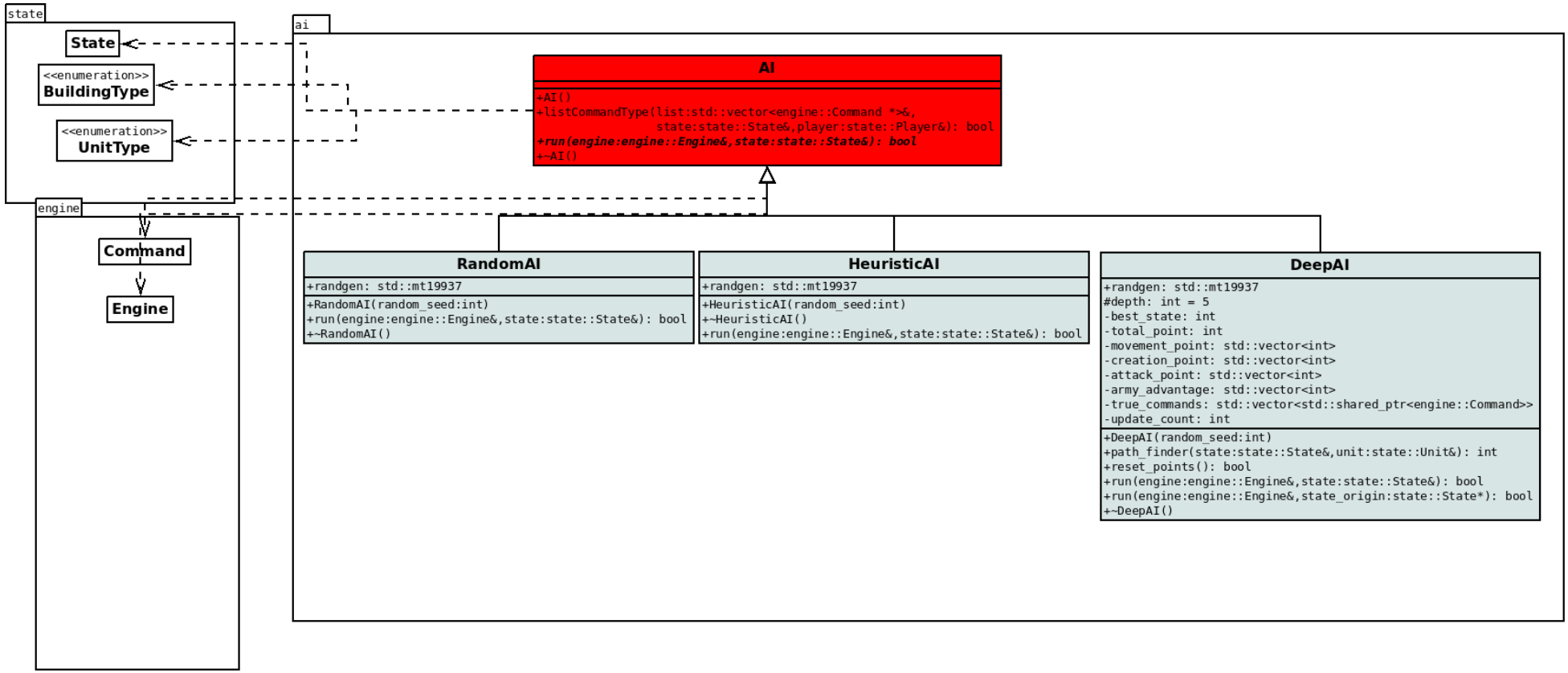
-total_point : la valeur finale obtenu par un état après calcul de points ;
-les différents points :
vecteur de points qui permettent de mettre en évidence les points obtenus par état ;

-true_commands : la liste de commande qui sera finalement exécuté ;

Ses méthodes :

reset_points()
permet de remettre à 0 les vecteurs de points ;

5.5 Conception logiciel : extension pour la parallélisation


state
State
<<enumeration>>
BuildingType
<<enumeration>>
UnitType
engine
Command
Engine
ai
AI
+AI()
+listCommandType(list:std::vector<engine::Command *>&,
state:state::State&,player:state::Player&): bool
+run(engine:engine::Engine&,state:state::State&): bool
+~AI()
RandomAI
+randgen: std::mt19937
+RandomAI(random_seed:int)
+run(engine:engine::Engine&,state:state::State&): bool
+~RandomAI()
HeuristicAI
+randgen: std::mt19937
+HeuristicAI(random_seed:int)
+~HeuristicAI()
+run(engine:engine::Engine&,state:state::State&): bool
DeepAI
+randgen: std::mt19937
#depth: int = 5
-best_state: int
-total_point: int
-movement_point: std::vector<int>
-creation_point: std::vector<int>
-attack_point: std::vector<int>
-army_advantage: std::vector<int>
-true_commands: std::vector<std::shared_ptr<engine::Command>>
-update_count: int
+DeepAI(random_seed:int)
+path_finder(state:state::State&,unit:state::Unit&): int
+reset_points(): bool
+run(engine:engine::Engine&,state:state::State&): bool
+run(engine:engine::Engine&,state_origin:state::State*): bool
+~DeepAI()

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

Nous avons choisi de mettre le moteur, le rendu et les IA sur différents threads.
L'état devient une variable globale, accessible à tous, comme le rendu.
Nous avons cependant rajouté des mutex pour prendre en compte l'état qui est une ressource critique.
Ainsi, notre thread du moteur ne peut s'exécuter en parallèle avec les threads des AI, qui sont elles mêmes non parallélisables entre-elles.
Au final, seulement notre rendu est multi-threadé avec les autres 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