Projet Logiciel Transversal

Adou DIALLO – Raphaël DUJARDIN

Table des matières

1 Objectif

1.1 Présentation générale

Le but du projet est la réalisation d'un jeu de stratégie tour par tour permettant la conquête de territoires inspiré du jeu éponyme RISK avec un nombre moindre de règles beaucoup plus simples.

1.2 Règles du jeu

Le but du jeu est de pouvoir s'emparer des territoires adverses tout en défendant le sien et tous ceux précédemment conquis.

- Univers
  L'univers du jeu est une carte 2D en vue de dessus. La carte est décomposée en territoires et en mers. On dénombre 5 types de territoires : prairie, montagne, forêt, désert, et côtier, repérables par une texture et une légende sur laquelle les joueurs se déplaceront. A cela s'ajoute un type particulier de territoire qui est le territoire neutre. Chaque joueur possède un territoire capitale qui a une grande importance. Sur tous ces territoires excepté le neutre sont présentes des troupes.

- Les joueurs
  Les joueurs peuvent être au maximum au nombre de 6. Ils ont chacun une couleur définie qui servira à différencier les territoires possédés par chacun.

- Les personnages
  Les personnages présents sur la carte qui permettront au joueur de jouer seront les troupes qui serviront à défendre ou conquérir un nouveau territoire. Parmi ces troupes existe un héros qui est un soldat particulier avec des caractéristiques différentes des troupes communes.

- Les items
  Les items apparaissent de manière aléatoire au bout d'un certain nombre de tours. Il y en a de plusieurs sortes avec des effets plus ou moins puissants. Ces objets peuvent autant être des malus que des bonus pour le joueur qui devra dès lors bien réfléchir à leur utilisation.

- Les actions
  Il y a deux types d'actions principales dans le jeu : attaquer et défendre. On peut aussi construire des ports. Il y a possibilité de passer son tour mais des mesures adéquates apparaissent à la prise de cette décision.

- Les tours
  Chaque joueur joue chacun son tour, pendant un tour il peut y avoir plusieurs phases qui peuvent être influencées par la présence ou non d'item.

1.3 Conception Logiciel

Packages :

- state
- render

Dépendances :

- SFML 2

2 Description et conception des états

2.1 Description des états

L'état du jeu est caractérisé par :
- le joueur dont c'est actuellement le tour
- la phase actuelle au sein du tour (placement des renforts / action / déplacement)
- l'item qui a été activé pendant le tour, soit automatiquement soit sur action du joueur, le cas échéant
- l'état de chaque joueur et de chaque territoire

L'état d'un joueur est caractérisé par :
- s'il est en vie ou pas
- le territoire qui est sa capitale
- le territoire sur lequel est positionné son héros
- l'item qu'il a en inventaire et le nombre de tours pendant lesquels il l'aura encore avant disparition, le cas échéant

L'état d'un territoire est caractérisé par :
- la liste des territoires adjacents
- ses propriétés géométriques (position et forme), utile pour le rendu
- son type (prairie / forêt / montagne / désert / côte)
- s'il est prenable ou non
- le joueur qui le possède le cas échéant
- l'item présent dessus le cas échéant
- s'il possède des ports ou non
- le nombre de troupes présentes dessus

2.2 Conception logiciel

L'état du jeu sera représenté par trois classes :
- Game pour l'état du jeu global
- Player pour les joueurs
- Land pour les territoires.

Le cycle de vie des objets Player et Land dépend de la classe Game. La classe Game est par ailleurs un singleton car elle représente l'état global de la partie.

Les items ne seront pas représentés par une classe car chaque item a un comportement très différent qui peut intervenir à beaucoup de niveaux différents (dans beaucoup de méthodes différentes), le plus simple, y compris pour ajouter de nouveaux items, est de tester l'item et de faire son action directement dans les méthodes concernées.

Voir diagramme des classes

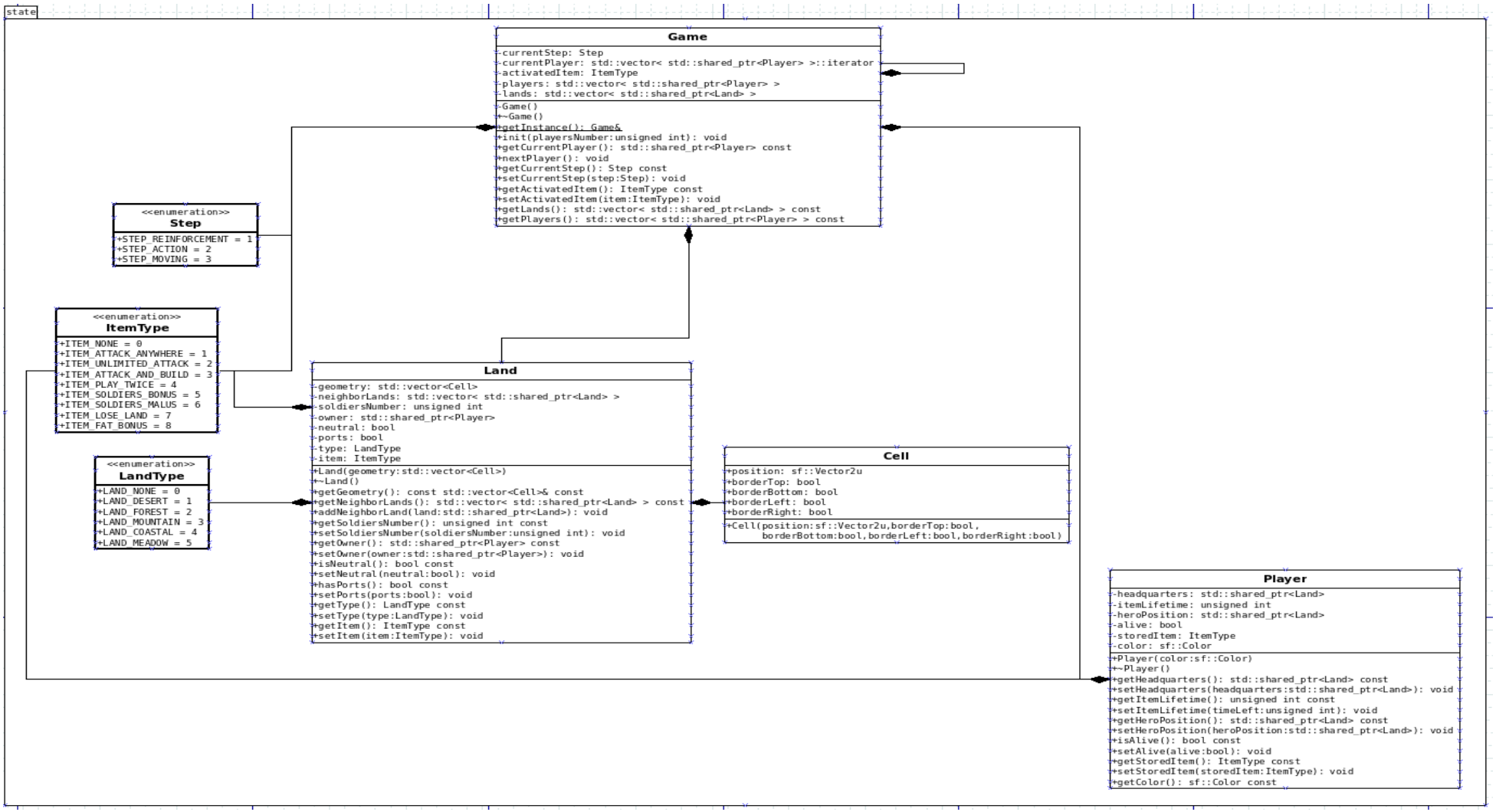
2.3 Conception logiciel : extension pour le rendu

La classe Land possède un attribut *geometry* de type liste de structures *Cell*. Chaque objet *Cell* possède une position x, y dans la grille des cellules de la carte (64x64 cellules), et des informations de bordure sous la forme d'un booléen pour chaque direction (haut, bas, gauche, droite). Les cellules auront une taille à l'écran de 9x9 pixels et les bordures feront 3 pixels d'épaisseur.

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

Tout l'état du jeu est accessible via le singleton Game, qui expose toutes ses données via des accesseurs. On peut donc construire une classe externe pour s'occuper du rendu.

Le rendu est effectué à 30 FPS, il n'a pas besoin d'être synchronisé avec le moteur de jeu puisque ce taux de rafraîchissement est largement supérieur à la vitesse de changement de l'état du jeu, sans consommer de ressources excessives.

Effectuer le rendu avec une classe externe à Game permet de séparer l'état de jeu de son affichage et de la gestion des ressources liées à cet affichage. Cette séparation sera bienvenue quand le jeu sera doté d'une architecture client / serveur.

3.2 Conception logiciel

Le package render est composé d'une unique classe Renderer, un singleton. Celui-ci fait appel au singleton Game du package state pour accéder aux données de l'état du jeu, et s'occupe de l'affichage ainsi que de la gestion des ressources (textures, polices, …).

La boucle événementielle est externe à cette classe Renderer. La classe Renderer a pour seul rôle de dessiner l'état du jeu. La gestion des événements ne lui reviendra pas, elle se fait donc pour le moment dans la fonction *main*. Renderer expose une méthode **void *render* (sf::RenderWindow&)** appelée dans la boucle événementielle pour le dessin du jeu.

3.3 Conception logiciel : extension pour les animations

3.4 Ressources

Numéro du joueur	Couleur distinctive	Son distinctif
1	Rouge (#ff0000)	res/sounds/player1.wav
2	Cyan (#00ffff)	res/sounds/player2.wav
3	Jaune (#ffff00)	res/sounds/player3.wav
4	Violet (#8000ff)	res/sounds/player4.wav

5	Noir (#222222)	res/sounds/player5.wav
6	Rose (#ff00ff)	res/sounds/player6.wav

Type de territoire	**Texture**
Prairie	Pas de texture
Forêt	res/textures/forest.jpg
Montagne	res/textures/mountain.jpg
Désert	res/textures/desert.jpg
Côtier	res/textures/coastal.jpg

Chaque territoire sera rendu avec une texture selon son type, filtrée par une couleur selon le joueur qui possède le territoire. Les territoires neutres auront la couleur blanche.

Les étendues d'eau n'auront pas de texture mais seront de couleur bleu (#0000ff).

Les ports seront matérialisés par une série de carrés noirs (représentant des pontons) disposés le long du littoral du territoire concerné, côté eau.

3.5 Exemple de rendu

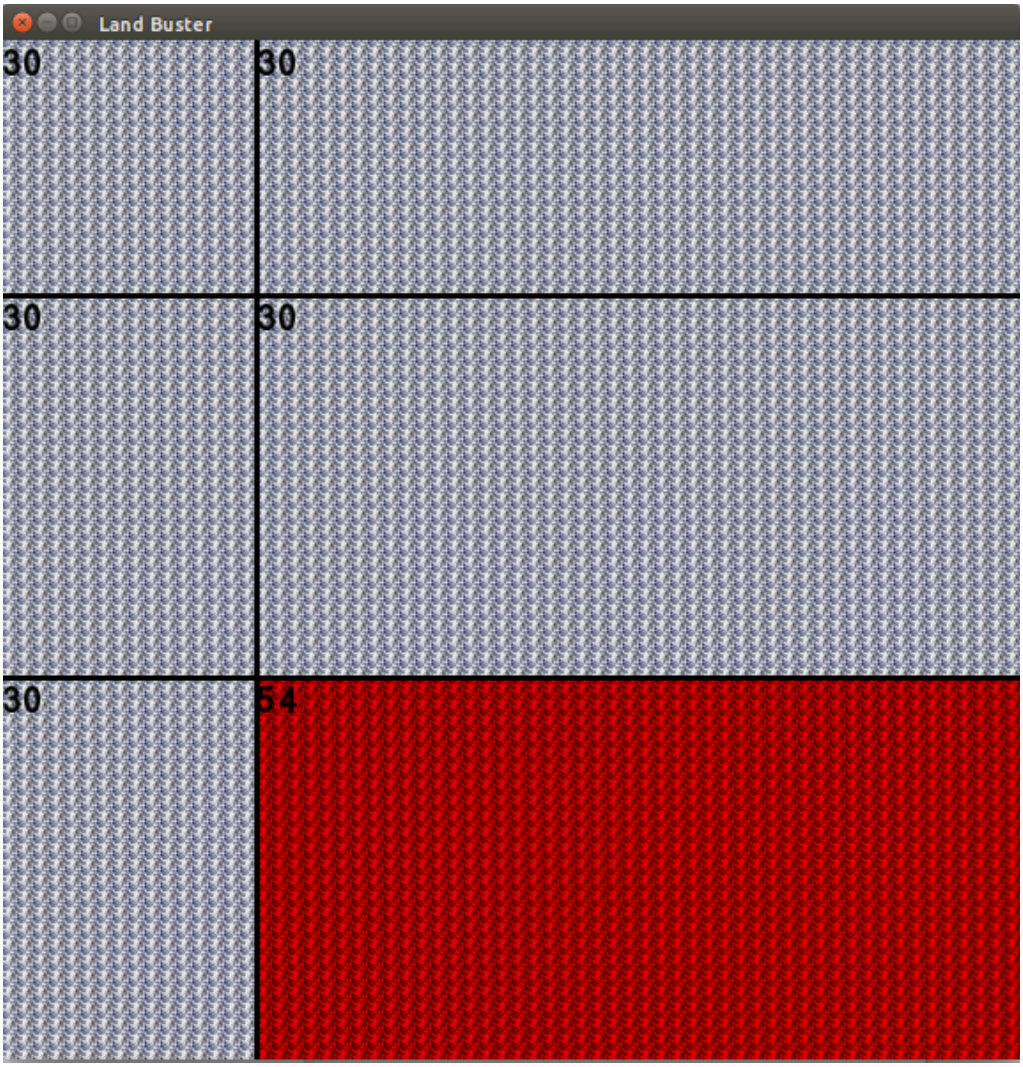
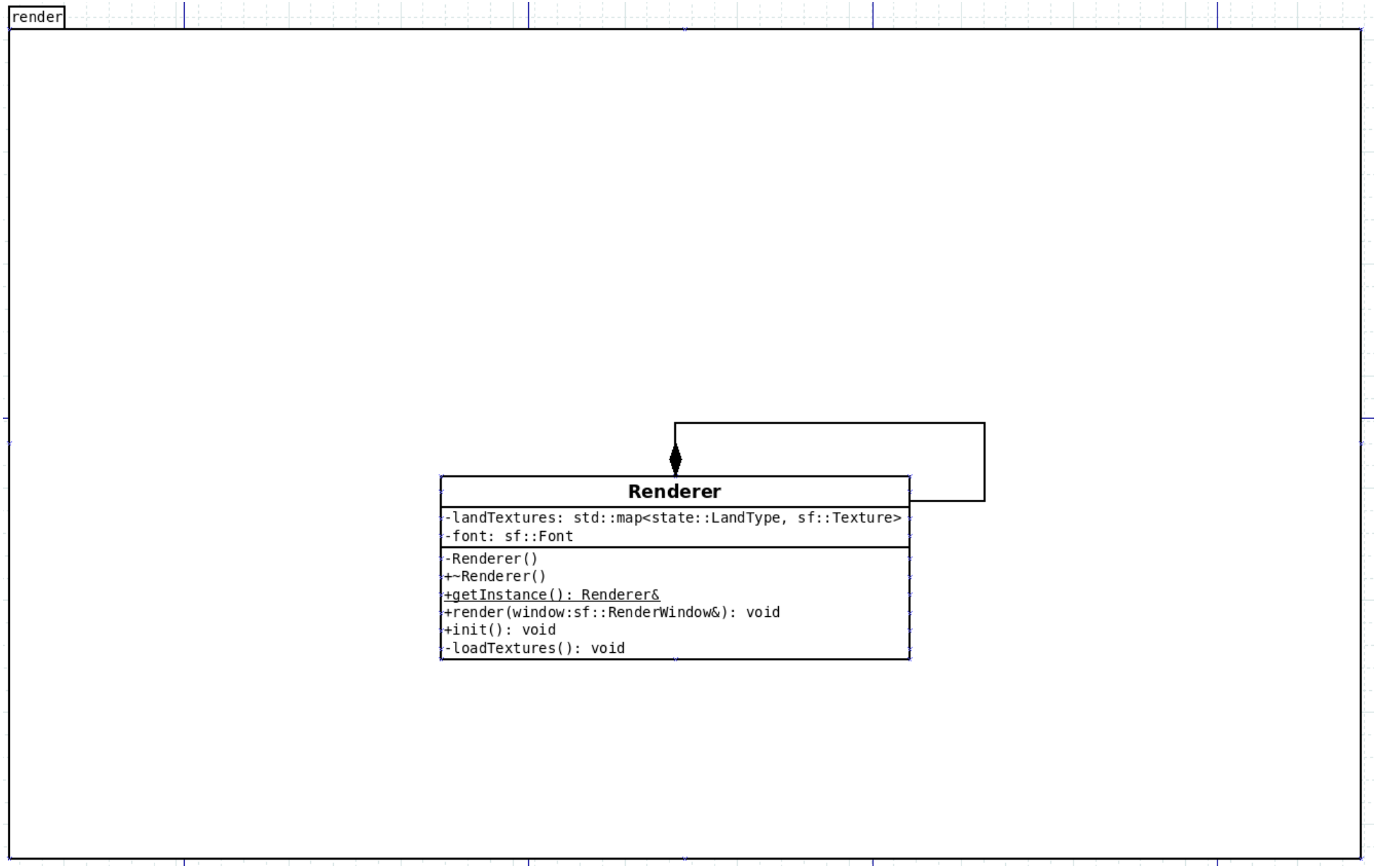


Illustration 2: Diagramme de classes pour le rendu



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

4.2 Changements extérieurs

4.3 Changements autonomes

4.4 Conception logiciel

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