ECOLE
POLYTECHNIQUE
DE BRUXELLES

Compléments de programmation et d'algorithmique

Compte-rendu de projet
Groupe 15

Romain Denis
Darius Couchard
Stefano Donne

12 décembre 2018

1 Avant propos

Les fonctionnalités de base spécifiées dans le cahier des charges ont toutes été implémentées, ce document se concentre donc sur les ajouts supplémentaires réalisés durant le projet. En l'occurence des méthodes faites pour donner à l'**IA** un comportement beaucoup plus cohérents, et les particularités de l'architecture du code **réseau** et ses avantages ou incovénients par rapport à un système peer-to-peer.

2 Intelligence(s) Artificielle(s)

La résolution de certaines des missions proposées nous ont amenés à la création de cinq IA distinctes :

-**Une IA inactive**: Elle ne fait rien, elle sert à tester les fonctionnalités des autres IA, notamment l'IA Greedy.
-**Une IA Greedy**: L'IA a le contrôle d'une unité d'infanterie. À chaque tour, cette unité se rapproche de la ville neutre ou ennemie la plus proche, et la capture si possible. Cette IA parcoure ainsi toute la carte et capture toutes les villes (contre l'IA inactive). Dans le cas où son unité est détruite, elle en recrée une dans une usine. Dans le cas où l'IA a le contrôle sur plusieurs unités d'infanterie, chaque unité se dirigera vers une ville différente grâce au système d'objectifs (décrit plus bas)
-**Une IA Recon**: L'IA peut créer trois types d'unités distinctes dans les usines : Infantry, Bazooka, Recon. Deux phases de jeux sont introduites: Une phase d'expansion, durant laquelle seules des unités d'infanteries sont créées dans les usines. Ces unités capturent des villes grâce à l'algorithme de l'IA Greedy. La deuxième phase commence lorsqu'un nombre prédéfini de villes ont été capturée par l'IA. La création aléatoire des autres types d'unités est alors lancée.
-**Une IA Anti-Recon**: Cette IA reprend le principe des 2 phases de l'IA Recon mais a la possibilité de créer tous les types d'unités aussi bien dans les usines que dans les aéroports. Par ailleurs, cette IA crée les unités maximales possibles au vu de son argent disponible, en favorisant en priorité les avions.
-**Une IA générique**: Cette IA est une amélioration de l'IA Anti-Recon à laquelle ont été ajoutées des méthodes permettant un comportement plus poussé.

Voici une brève description des systèmes supplémentaires ajoutés au fonctionnement des IA.

Système de contre-unité: Deux méthodes ont été ajoutées, permettant de vérifier la présence ou non d'unité blindé et aériennes chez l'ennemi. Elles renvoient un boolean et sont donc utilisées comme condition dans la fonction génératrice d'unités , par exemple ; si l'ennemi génère une unité aérienne, les probabilités de créer une unité anti-aérienne sont largement augmentées.

Système d'objectifs: Ce système a été conçu pour l'IA générique et implémentée dans toutes. Il assigne un objectif à chaque unité, un objectif peut être soit un batiment capturable soit une unité ennemie.
Dans le cas d'un batiment, l'IA va chercher le batiment capturable le plus proche et vérifier qu'il n'est pas déjà l'objectif d'une unité alliée.
Dans le cas d'une unité, l'IA va assigner comme objectif l'unité ennemie la plus proche qui n'est pas déjà l'objectif de 3 unités alliées et qui remplit les critères de la méthode *niceattack*. Cette dernière vérifie la pertinence de l'attaque vers la cible.Par exemple, elle va empêcher une infanterie de prendre pour cible un neotank ou un bombardier. Enfin, l'unité se déplacera vers la case de déplacement libre la plus proche de l'objectif.

Système d'attaque intelligent: Toutes les IA en sont munies. Si une unité a la possibilité d'attaquer une unité et de la détruire, ou si elle a la possiblité de lui infliger des dégats plus lourd que ce qu'elle recevra lors de la contre-attaque, elle le fait en priorité. La seconde priorité est donnée à la capture pour les Infanteries et Bazooka. La troisième priorité est le déplacement de l'unité vers son objectif.

3 Réseau

L'architecture du réseau initiallement demandée dans l'énoncé est une architecture de connexion peer to peer directe entre les deux joueurs, dont la communication est assurée par un Socket qui envoye les informations sous format Json en protocole TCP. Tous les outils utilisés viennent du *Framework Qt*, notamment la connection entre les *SIGNALS* et les *SLOTS* qui nous épargne de la manipulation des *Threads*.

Bien que quelques adaptations ont été faites afin de respecter le style de communication énoncé dans le cahier de charge, l'Architecture à été poussée plus loins dans l'idée de laisser par la suite la possibilité de jouer à plus de 2 joueurs en utilisant le modèle Client/Server. Il y a donc deux modes de lancement pour le jeu en ce qui concerne le réseau: *Client & Server,* ou bien *Client only*.

Partie Server: Un objet *QTcpServer* du Framework crée un nouvel objet *QTcpSocket* pour chaque nouvelle connexion entrante et l'enregistre dans un objet *Session*, qui contiendra en plus du socket un pointeur vers le modèle du jeu, le joueur dont il appartient (un joueur par client) et toutes les méthodes nécessaire pour réagir aux instructions donnés par les clients non-locaux et renvoyer les instructions aux autres clients. Le client local, c'est à dire le client de la même instance de programme que celle du server, fait directement les modifications sur le modèle, dans ce cas le server ne fait donc que renvoyer aux autres joueurs l'instruction.

Partie Client: La classe *NetworkClient* contient toutes les méthodes nécessaires pour envoyer et réagir aux instructions du server. Un *QTcpSocket* est initialisé et connecté au server dans le constructeur du *NetworkClient.*

Client/Server *vs* peer-to-peer.

Bien que l'Architecture Peer-to-Peer est nettement plus simple que l'architecture Client/Server quand il s'agit de directement partager les instructions entre deux programmes, elle devient rapidement beaucoup plus compliquée lorsque l'on rajoute des joueurs supplémentaires. Notamment si il y a un problème de désynchronisations des modèles du jeu entre les différents joueurs. En plus de cela, l'Architecture Client/Server est beacoup moins susceptible au cheat: le Server peut vérifier les instructions données avant d'apporter une modification au modèle du Jeu, en particulier dans les jeux tour par tour où la fréquence d'instructions envoyées est relativement basse (dans un *FPS,* il est bien plus difficile de tout vérifier car beaucoup plus d'actions sont effectuées à la seconde, par exemple: le jeu serait beacoup trop lent si le server devait vérifier puis confirmer chaque rotation de caméra par les joueurs, d'où la possibilité d'utiliser facilement un *aimbot* même dans les plus grand titres).

4 Annexe

`Problème de socket rencontré durant la fusion des deux parties.`

Un problème dans la partie *Network* a été rencontré lorsque la partie *Intelligence Artificielle* et la partie *Network* ont été fusionnées. Jouant beaucoup de coups à la seconde, les instructions de l'IA envoyées par le Client ne sont pas toutes reçues par le Server. Le Signal *readyRead()* n'est appelé qu'une seule fois sur tout le tour de l'IA.

Nous avons alors essayé d'ajouter un intervalle de temps entre chaque instruction grâce à la méthode *std::this_thread::sleep_for()* en attendant jusqu'à une seconde par instruction, mais cela n'à rien changé. Encore plus curieux en sachant que les joueurs jouent souvent plus rapidement qu'à une instruction à la seconde et que pourtant le réseau fonctionne parfaitement en JcJ.

En conclusion, toutes les fonctionnalités, à l'exception des IAs (sauf inactive), fonctionnent parfaitement en JcJ avec le réseau. Le code se trouve dans la branch "networking" de github. Par contre, pour tester les fonctionnalités des IAs, il faut reprendre le code avant l'implémentation du réseau, situé dans la branch "master" de github.