UNIVERSITÉ DE SHERBROOKE

Projet IFT630

Présenté à M. Gabriel Girard

Dominic Vincent 10 201 319

30-Apr-13

INTRODUCTION **2**

PROBLÉMATIQUE **2**

MISE EN CONTEXTE 2
Travaux précédents *2*
Énoncé *2*
NOTIONS PARALLÈLES 3
Client / serveur *3*
Serveur multi-thread *3*
Maître / travailleur *3*
INTÉRÊTS 3

IMPLÉMENTATION **4**

CONFIGURATION 4
Client: HTML5 et jQuery *4*
Serveur : Jetty *4*
Programmes et plug-ins utilisés *4*
Autres configurations *4*
CONCEPTION 5
Client *5*
Serveur *5*
DIFFICULTÉS RENCONTRÉES 5

RÉSULTAT **6**

CAPACITÉ DU SERVEUR 6
CAPACITÉ DU CLIENT 6
LIMITES DES OPTIONS IMPLANTÉS 6

CONCLUSION **6**

FONCTIONNALITÉS À IMPLANTER 6
OPTIMISATIONS POSSIBLES 6

INTRODUCTION

Le document qui suit présente et documente les aspects importants de la réalisation du projet de fin de session dans le cadre du cours IFT630. J'y décris entre autre les raisons de mon choix de projet, de type d'implémentation de des programmes utilisés, les problèmes rencontrés et le résultat final.

PROBLÉMATIQUE

MISE EN CONTEXTE

TRAVAUX PRÉCÉDENTS

Pendant la session d'automne 2013, j'avais comme projet personnel d'implanter un algorithme qui pouvait simuler un combat entre un nombre N de légions. Ce programme prenait en entrée la liste des légions et produisait en sortie un rapport contenant la liste des unités qui tombaient au combat. La simulation se déroulait tour par tour, et un certain nombre d'unités tombaient à chaque tour. Cet algorithme prend aussi en considération que plusieurs légions peuvent être alliées et donc s'ignoreront lors de l'assignation du dégât prit par chaque unité. Finalement, on prend aussi en considération le fait que chaque unité a un attribut *weight* qui lui est associé, dans le but de mettre dans une seule variable l'impact que peut avoir une unité dans le combat. Une unité va alors distribuer son dégât de façon proportionnel au *weight* de l'unité adverse par rapport au *weight* total des unités adverse. Plus une unité a un gros *weight*, plus elle sera la cible des autres unités. Cela a pour but d'éviter que, si une unité est extrêmement plus puissante que les autre, elle survie à coup sûr et gagne automatiquement le combat pour sa légion.

Après avoir testé en long et en large cet algorithme, j'ai trouvé intéressant la possibilité d'ajouter une notion de parallélisme dans celui-ci. Étant donné que l'algorithme était déjà programmé et testé, il ne restait qu'à ajouter le parallélisme et l'implanter dans un langage supportant ceci, puisque je l'avais d'abord implanté en PHP.

ÉNONCÉ

Le projet consiste à l'implémentation d'une portion d'un jeu de simulation de combat. Dans ce jeu, chaque joueur peut créer des unités et les regrouper pour former une légion et ainsi combattre ses adversaires. Le jeu se déroule tour par tour, c'est-à-dire qu'un joueur choisi ses actions pendant son tour, puis ses actions s'exécutent, et enfin on passe au tour du prochain joueur. Dans la portion du jeu qui est implémentée, seule l'action de déplacer une légion sera disponible. Si deux légions ennemies se croisent, il y a alors combat. Il faudra à ce moment exécuter un seul tour de l'algorithme de combat, et mettre à jour le contenu des légions. Un joueur ne possédant plus d'unités perd la partie. Si une autre légion ennemie croise le combat, elle s'y joint.

NOTIONS PARALLÈLES

CLIENT / SERVEUR

Ce projet nécessitera une architecture client / serveur. Chaque joueur est représenté par un client, et le serveur traite les requêtes des différents clients en parallèle. Le serveur doit pouvoir supporter plusieurs parties à la fois.

SERVEUR *MULTI-THREAD*

Le serveur doit être capable de gérer plusieurs requêtes à la fois. Le client ne doit pas bloquer s'il envoie une requête dû au fait que le serveur est en train de traiter une autre requête. Chaque légion doit aussi être implémentée comme un fil d'exécution sur le serveur. Même si ce n'est potentiellement pas la meilleure technique, j'aimerai faire l'implémentation de cette façon pour des raisons pédagogiques.

MAÎTRE / TRAVAILLEUR

Tel que mentionné précédemment, le calcul du combat sera en partie parallélisé. Étant donné que chaque unité est unique et assigne du dégât individuellement à chaque unité ennemie, la quantité de petit calcul est énorme. Il y aura donc une implémentation de la technique maître / travailleur pour accélérer le calcul. Le maître créera des travailleurs qui feront un calcul et qui sauvegarderont le résultat dans une table passée en paramètre. Les accès à cette table se feront de façon synchronisée, pour éviter qu'elle soit modifiée en même temps qu'elle soit lue.

INTÉRÊTS

J'ai choisi de faire ce travail car je suis un passionné des mondes fantaisistes, et je rêve un jour de développer un mini-jeu comme celui que ce projet décrit où les unités sont des créatures imaginaires. J'ai eu beaucoup de plaisir à implémenter l'algorithme de calcul de combat, et je crois que je vais en avoir tout autant pour ce projet.

IMPLÉMENTATION

CONFIGURATION

Je désirais faire un programme qui fonctionne directement à partir d'un navigateur web, puisqu'il s'agissait d'une technique qui, selon moi, est assez simple, et pour laquelle il existe énormément de documentation. De plus, le *cloud computing* est encore à la mode, alors de cette façon, je peux utiliser les outils les plus à jour.

CLIENT: HTML5 ET JQUERY

Côté client, je tenais à utiliser du HTML5, pour commencer à toucher un peu à cette « presque » nouveauté. HTML5 offre les web socket, qui sont une façon de faire en sorte qu'un client et un serveur peuvent communiquer par message en tout temps. Un serveur peut envoyer un message à un client et ce dernier peut se mettre à jour en temps réel sans avoir à rafraîchir la page, et sans avoir à utiliser des techniques particulières comme le *poiling*. J'ai également utilisé jQuery pour exécuter de simples scripts qui modifient le comportement et l'apparence des objets DOM de la page.

SERVEUR : JETTY

Je suis à l'aise dans la programmation en Java, alors j'ai décidé d'avoir un serveur qui fonctionne dans ce langage. Jetty offre un support pour les web socket, et est un logiciel qui est mis à jour régulièrement. Pour ce projet, j'ai utilisé Jetty 8, car l'outil de développement que j'ai utilisé ne me permettait pas d'utiliser la version la plus récente de Jetty, c'est-à-dire la version 9.

PROGRAMMES ET PLUG-INS UTILISÉS

La page index.html a été codée sur Notepad++, et exécuté sur Google Chrome. Le serveur a été codé sur éclipse, et est exécuté à l'aide de Maven et Run-jetty-run. La configuration est entièrement dans Éclipse, rien n'a besoin d'être téléchargé à l'extérieur de la fonction « Install new software » du menu « help » d'Éclipse.

AUTRES CONFIGURATIONS

Le programme s'exécute par défaut sur le port 8080. Dans index.html, la variable wsURL désigne l'endroit où aller chercher le serveur, alors il faut changer sa valeur si le serveur est exécuté sur un autre ordinateur que celui sur lequel il a été testé.

CONCEPTION

CLIENT

Le client peut voir le jeu de façon graphique et concrète. Un tableau 10x10 représente le champ de bataille, un carré rouge la position de son adversaire, et des boutons lui permettant de se déplacer sur les cases légales. Il y a aussi un bouton qui lui permet d'entrer dans une partie et un autre qui lui permet de quitter.

Chaque bouton est associé à une connexion avec un web socket. Les web socket envoient des messages sous forme de chaîne de caractères, où le premier mot est un code pour spécifier une action que le serveur a à accomplir, et les autres sont les paramètres. Plusieurs web socket peuvent s'ouvrir à la fois, mais ils sont tous refermé à la fin de chaque action à l'exception de l'un d'entre eux, qui lui est utilisé par le serveur pour envoyer des mises à jour au client, en particulier la nouvelle position de son adversaire.

SERVEUR

Le serveur implémente un objet par entité. Lorsqu'il reçoit un message, il envoie l'action à faire à la bonne instance de la classe « Game ». Chaque partie, joueur, unité, légion et même classe et race ont leur propre identificateur (« id »), qui permet de les retrouver dans chaque « Map ».

DIFFICULTÉS RENCONTRÉES

La plus grande difficulté fut sans aucun doute le choix initial de la configuration que j'allais utiliser. Je tenais vraiment à utiliser les web socket et Java, et les options disponibles étaient Jetty, GlassFish et jWebSocket. J'ai essayé les trois et eu beaucoup de misère à les faire fonctionner. J'ai finalement décidé de me concentrer à faire fonctionner Jetty, puisque c'était le serveur qui était le plus souvent conseillé selon mes recherches. C'est là que j'ai appris ce qu'était Maven et son plug-in dans Éclipse, ce qui m'a beaucoup aidé. J'ai finalement trouvé run-Jetty-run, à partir duquel je lance tous mes tests. Il s'agit d'une configuration qui est extrêmement simple à utiliser, mais quelque peu complexe à installer.

Par la suite, j'ai eu un problème avec les web socket. La « webSocketFactory » de jetty semblait toujours produire un web socket null. Cela m'a pris une journée pour réaliser que je n'avais simplement pas spécifié la bonne version de Jetty dans pom.xml, la bonne version étant 8.0.0.RC0.

Le dernier problème majeur que j'ai eu fut lorsque j'ai paralléliser le calcul du combat. J'avais une erreur que je n'avais jamais vue auparavant qui spécifiait un accès concurrent illégal à la table des résultats. Il a fallu que je fasse quelques recherches pour savoir que le mot clé « synchronized » pour une fonction en java spécifie son accès comme si elle était utilisée par un moniteur.

RÉSULTAT

CAPACITÉ DU SERVEUR

Le serveur peut gérer une quasi-infinité de joueurs et de parties en même temps. Il traite également en parallèle toutes les requêtes reçues, ce qui est implanté dans Jetty.

CAPACITÉ DU CLIENT

Le client peut joindre une partie, la quitter, utiliser l'option déplacer, combattre un adversaire sur la même case, gagner et perdre une partie.

LIMITES DES OPTIONS IMPLANTÉS

Présentement, chaque joueur ne peut posséder qu'une seule légion, et chaque partie ne peut avoir que deux joueurs. Cependant, il ne serait pas si complexe d'ajouter ces fonctionnalités, étant donné que tout a été implanté en vision d'ajouter ces fonctionnalités. Il ne resterait surtout qu'à les tester, ce qui peut être assez long.

CONCLUSION

FONCTIONNALITÉS À IMPLANTER

Mon objectif est de créer un mini-jeu basé sur le code que j'ai présentement, alors il est évident que dans un avenir proche, certaines fonctionnalités seront ajoutées, à commencer par celles déjà mentionnées, voire la possibilité d'être plus de joueurs et avoir plus de légions. Ensuite, il faudrait implémenter une page qui fait la liste de toutes parties en cours, et pouvoir joindre une partie ou encore en visionner une. Un chat serait aussi très simple à implanter.

Ensuite, ajouter la possibilité de configurer les options d'une partie: taille du champ de bataille, nombre maximum de joueur, couleur des légions et utiliser des hexagones au lieu des cases.

Finalement, il s'agirait d'ajouter toutes les options du joueur, comme par exemple, créer des unités, amasser des ressources, construire des bâtiments et plusieurs autres.

OPTIMISATIONS POSSIBLES

Il existe un problème de redondance inutile dans mon code coté client présentement. Si mon serveur supportait PHP, je n'aurais pas à avoir 100 <th></th>. Ajouter le support de PHP deviendra sans doute une priorité si le code coté client s'agrandi de plus en plus. Enfin, le code coté serveur n'est pas entièrement parallélisé. Il est possible d'ajouter d'autre fil d'exécutions. Cependant, la rapidité de mon serveur n'est pas un problème pour l'instant, alors il s'agit ici simplement d'une piste d'optimisation possible non prioritaire.