Alvarez Josué 2 MIC D
Cichocki Guilhem

COMPTE RENDU DE PROJET : LABOTLANE

Alvarez Josué 2 MIC D
Cichocki Guilhem

SOMMAIRE

Introduction **1**

I. Description sommaire du projet **2**

I.1. Déroulement du jeu 2

I.2. Environnement 2

II. Tests **4**

III. Architecture **6**

IV. Gestion du projet **8**

IV.1. La création du projet 8

IV.2. Gestion des fichiers 8

V. Un problème algorithmique intéressant : l'algorithme de recherche de plus court chemin. **9**

Conclusion **10**

Alvarez Josué 2 MIC D
Cichocki Guilhem

COMPTE RENDU DE PROJET : LABOTLANE

INTRODUCTION

Pour ce projet, nous avons décidé de créer un jeu vidéo car premièrement, nous sommes tous deux attachés au domaine du jeu vidéo, mais aussi, car le développement d'un jeu nécessite la résolution d'un grand nombre et d'une grande variété de problèmes. Notre niveau étant très hétérogène, nous voulions pouvoir être en mesure de répartir les tâches de manière à atteindre tous les deux nos objectifs personnels. Ce choix de projet nous a permis de nous attaquer à différents problèmes de conception, d'algorithmique et de travail d'équipe.

Dans ce rapport, nous n'allons pas détailler toutes les solutions techniques que nous avons mises en place (car elles sont trop nombreuses et nécessitent une connaissance en profondeur de notre architecture). Néanmoins, le code est consultable sur le dépôt SVN suivant : https://www.etud.insa-toulouse.fr/svn/2MIC-Labotlane/.

Remarque : toute ressemblance entre le nom du jeu et un professeur bien connu est totalement fortuite…

I. DESCRIPTION SOMMAIRE DU PROJET

I.1. DÉROULEMENT DU JEU

Deux joueurs jouent tour à tour et contrôlent des unités ainsi qu'une fabrique d'unités. Le jeu se déroule sur une carte en forme de grille, où sont placées les unités, l'environnement, ainsi que divers « bonus ».

FIGURE 1. CAPTURE D'ECRAN DU JEU.

L'objectif principal est la récolte d'un maximum de points en une période donnée au travers de plusieurs objectifs (destructions d'unités, obtention d'objectifs). La réalisation de ces objectifs rapporte également de l'argent permettant de construire des unités.

Le jeu s'effectue en tour par tour durant lesquels les joueurs peuvent contrôler leurs unités individuellement dans un temps imparti de l'ordre de la minute, ainsi que créer de nouvelles unités.

Les joueurs pourront être des humains ou des intelligences artificielles.

I.2. ENVIRONNEMENT

Plusieurs éléments sont présents dans l'environnement :

- Mer (non traversable par les unités de base, mais potentiellement traversables par les unités marines).
- Mines (permettant l'obtention d'argent grâce à des unités de minage).
- Fabriques (permettant la réalisation d'unités moyennant un coût en argent).
- Murs (empêchant les mouvements de toutes les unités).

TYPES D'UNITÉS

- Combattants : unités au sol permettant la destruction de structures et d'autres unités. Il en existent divers types ayant des caractéristiques différentes.
- Mineurs : unités au sol permettant l'obtention d'argent et de points lorsqu'ils minent des structures particulières.
- Heal : unités au sol permettant de soigner d'autres unités.
- Fabriques d'unités : Permet la fabrication d'unités en échange d'argent.

II. TESTS

Etant donné la nature de notre projet, il n'était pas pratique (ni forcément pertinent) de tester les fonctionnalités via des tests unitaires classiques. En effet, les éventuels problèmes ne pouvaient survenir qu'à la suite d'effets de bord dans les fonctions testées, effets de bord qu'il est difficile de simuler.

Nous avons donc décidé de tester chaque fonctionnalité nouvelle en situation réelle, et d'observer le comportement du jeu lors de l'utilisation de cette fonctionnalité, dans les cas « normaux » et « anormaux ». Cependant, cette méthode comportait un inconvénient majeur : la mise en place des cas de test était longue (il fallait reproduire la situation de test dans le jeu).

Nous avons donc décidé de créer un langage de script qui nous permettrait de tester des fonctionnalités de manière automatique, en utilisant le moteur de jeu pour créer les cas de test. Aussi, ce langage nous permet de visualiser des variables en temps réel, afin de connaître leur évolution au cours du temps.

Exemple de script de test :

```
# Suppression des variables locales.
@clearlocal soldier
@clearlocal facto1
@clearlocal facto2
@clearlocal pingo
@clearlocal goomba

# On donne aux joueur l'or nécessaire pour faire spawn les entités.
@give playerId=0 amount=400
@give playerId=1 amount=400

# On importe les factory permettant de faire spawn les entités.
@importfactory 0 facto1
@importfactory 1 facto2

# On fait spawn une entité par équipe.
spawn src=facto2 x=20 y=20 type=PkmPingoleonEntity name=pingo
spawn src=facto1 x=21 y=23 type=MGoombaEntity name=goomba

# On demande au goomba d'attaquer.
attack src=goomba x=@20 y=@20

# On demande un mouvement au goomba.
move src=goomba x=1 y=2

# On redemande un mouvement impossible (point inaccessible) au goomba, pour voir
# comment se comporte l'A*.
resetmp goomba
move src=goomba x=1 y=1

# On ajoute la variable Position du goomba dans le tracker.
@track goomba Position
@track goomba RemainingMovePoints
```

En jeu, une console peut être affichée et permet d'exécuter des scripts stockés dans des fichiers (via la commande `@ldscript`), ou des lignes de code.

La console étant un outil de debug, il est important qu'elle puisse faire remonter les erreurs de compilation mais surtout d'exécution des scripts (actions non autorisées, etc…).

FIGURE 2. EXEMPLE D'ERREUR D'EXECUTION D'UN SCRIPT, DONNANT LIEU A UN RAPPORT D'ERREUR SEPARE ET UN RAPPORT SUR LA SORTIE CONSOLE DU JEU

Les erreurs de compilation et d'exécution des commandes (non chargées depuis les scripts) sont uniquement affichées dans la console.

FIGURE 3. EXEMPLE D'ERREUR DE COMPILATION D'UNE LIGNE DE COMMANDE.

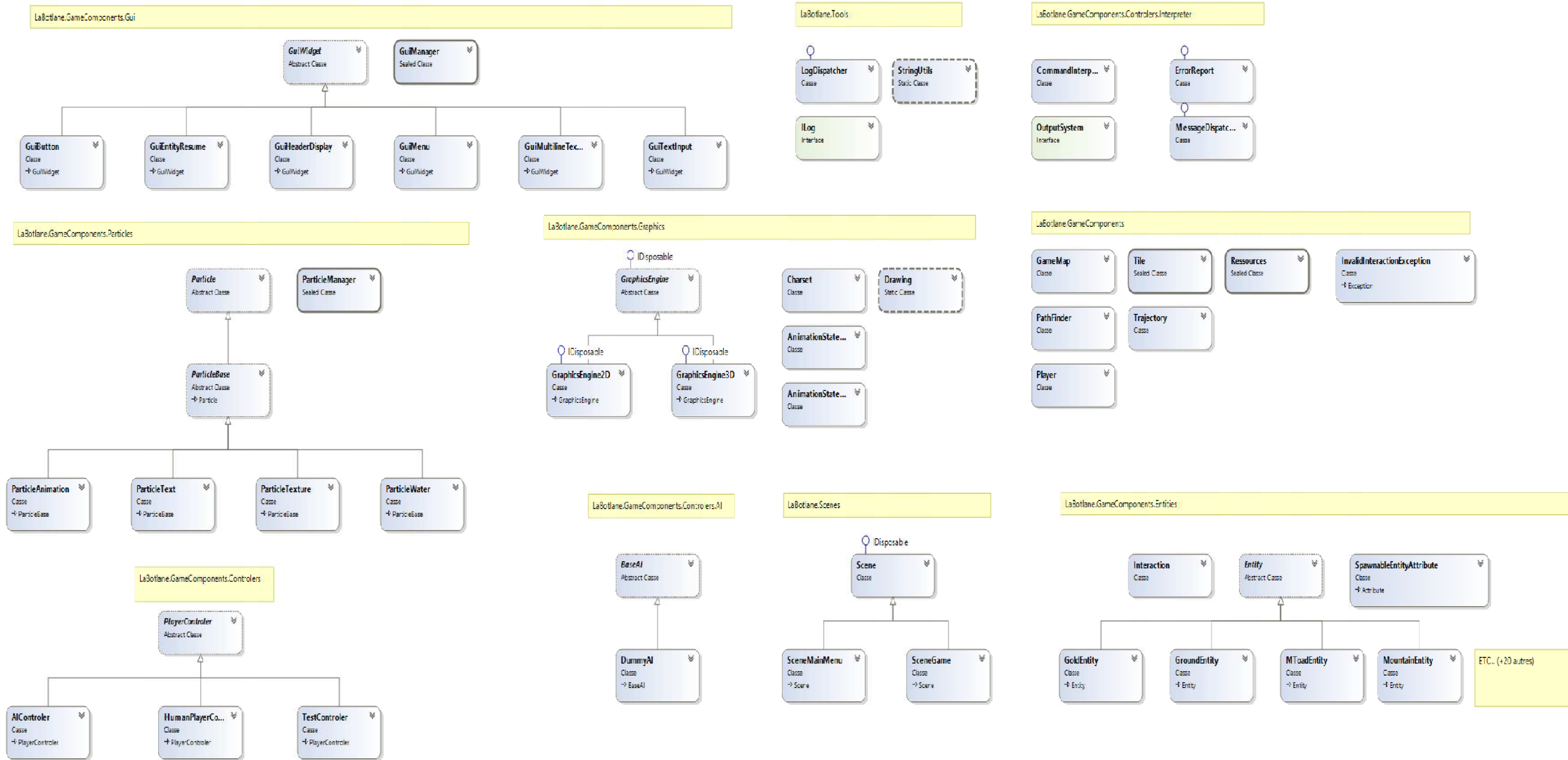
III. ARCHITECTURE

Notre architecture logicielle tire pleinement parti du paradigme de programmation Orienté-Objet. Lors de la conception de notre projet, nous nous sommes imposés les contraintes suivantes :

- La partie graphique doit être indépendante de la logique de jeu. La logique de jeu peut contrôler certains indicateurs graphiques (ex : choisir de mettre en surbrillance certains objets), mais ne connait pas le code associé à la mise en place de ces indicateurs.
- La logique de jeu doit être le plus centralisée possible.

Nous avons donc découpé notre projet en plusieurs sous parties, regroupées dans des Namespaces :

- **LaBotlane.GameComponents** : cette partie contient tous les composants de jeu, dont la plupart répartis dans des sous-espaces de noms détaillés ci-après. Elle contient entre autres la classe gérant la carte (Map), l'algorithme de recherche de plus court chemin (A*), le système de gestion de ressources (qui s'occupe de charger les images, shaders, ou autres ressources).
- **LaBotlane.GameComponents.Controlers** : cette partie s'occupe de la logique de jeu. Elle contient une classe de base « PlayerControler » qui contient le code de traitement et de validation des « intéractions » des entités de jeu. D'autres classes héritent de cette classe et contiennent aussi le code qui permet aux différents types de joueurs (Humains, Intelligences artificielles, ou voir même – pas encore implémenté- Réseau (système Client/Serveur possible) d'interagir avec le moteur de jeu).
- **LaBotlane.GameComponents.Controlers.Interpreter** : contient les classes responsables de l'interprétation du code de script.
- **LaBotlane**.**GameComponents.Graphics** : contient les classes qui permettent l'affichage des différents éléments de jeu. Une classe de base « GraphicsEngine » existe et contient des méthodes qui encapsulent le code de dessin des éléments de jeu. Les composants qui souhaitent afficher quelque chose à l'écran passent par GraphicsEngine pour cela.
- **LaBotlane**.**GameComponents.Gui** : contient les classes qui permettent l'affichage et la gestion d'éléments de l'interface Homme Machine du jeu (boutons, menus, textboxes etc…).
- **LaBotlane .GameComponents.Particles** : contient un ensemble de classes qui permettent l'affichage et la gestion des effets de particules visibles en jeu. Par exemple, l'affichage des animations d'attaque, des effets indiquant des la perte de points de vie, sont gérées par des classes héritant d'une classe de base Particle.
- **LaBotlane.GameComponents.Entities** : contient une classe de base Entity, et des sous classes contenant des données concernant chaque entité (nombre de points de vie max, restants, type etc…), mais aussi des fonctions de base permettant au moteur de jeu d'interagir avec ces entités.
- **LaBotlane.GameComponents.AI** : contient une classe de base de d'intelligence artificielle dont héritent les différentes intelligences artificielles.
- **LaBotlane.Tools** : contient divers outils.



2 MIC D

Alvarez Josué
Cichocki Guilhem

Diagramme de classe partiel représentant les différentes classes rangées dans leur namespace et leurs liens de parentés (héritage seulement, agrégation non comprise)

IV. Gestion du projet

Afin de pouvoir travailler facilement sur le projet chacun chez soi, et ne pas avoir à éviter les conflits manuellement, nous nous sommes tournés vers des logiciels nous permettant de gérer facilement un dossier commun, et dans dans cas, un projet de groupe.

IV.1. La création du projet

Avant de commencer à coder notre projet, nous avons utilisé un outil présent sur Visual Studio 2013, permettant de créer facilement des diagrammes UML (Annexe 1). Grâce à cet outil, nous avons pu définir précisément les classes que nous allions devoir créer, afin d'avoir une structure de données intelligente et efficace. Nous avons ensuite créé un un calendrier, indiquant des dates limites avant lesquelles il fallait avoir atteint certains objectifs. Il a donc fallut évaluer le temps nécessaire au développement de certains algorithmes. Comme l'algorithme de recherche de plus court chemin par exemple.

IV.2. Gestion des fichiers

Dropbox : un choix pour commencer, pas forcement pertinent.

Dans un premier temps, nous nous sommes tournés vers Dropbox, car nous avions tout les deux déjà utilisé ce système pour à d'autres occasions. De plus, il nous avait semblé rapidement nécessaire d'utiliser un logiciel de ce type car notre projet était ambitieux, et l'un de nous deux ne pouvait être présent après les cours sur le campus régulièrement. Nous avons en revanche, vite rencontré quelques problèmes avec Dropbox car des conflits pouvaient très facilement être générés dès que nous étions deux à modifier un même fichier en même temps. Le logiciel copiait donc le fichier, présent ensuite en deux exemplaires. De plus, il pouvait arriver que Dropbox partage un fichier sauvegardé qui ne compilait pas puisque qu'il met à jour un fichier dès qu'une personne du groupe demande à enregistrer la moindre différence. Nous avons donc rapidement décidé après avoir reçu quelques conseils de nos professeurs de nous tourner vers un vrai gestionnaire de versions.

SVN

Nous avons, à la suite des problèmes rencontrés avec Dropbox, décidé de migrer vers le gestionnaire de versions SVN. Nous avons utilisé le service SVN disponible sur le serveur etud, et avons choisi d'utiliser TortoiseSVN comme client SVN, couplé au plug-in Ankh SVN pour Visual Studio.
L'outil nous permettait de gérer efficacement les conflits lorsque des modifications étaient faites sur le même fichier, grâce à une vue comparée des versions en conflit.
Après un court moment d'utilisation, nous avons décidé d'instaurer certaines règles qui nous ont évité plusieurs situations désagréables :

- On ne commit une modification que si elle compile.
- On ne commit une modification que si elle a été correctement testée. Si cela n'est pas le cas, le commentaire : TODO TESTME doit être ajouté afin qu'on n'oublie pas de tester la fonctionnalité affectée.

V. UN PROBLÈME ALGORITHMIQUE INTÉRESSANT : L'ALGORITHME DE RECHERCHE DE PLUS COURT CHEMIN.

Afin que les déplacements puissent se faire simplement pour les joueurs, nous devions être en mesure de trouver le chemin le plus court entre la position d'une unité, et la position à laquelle le joueur veut la déplacer. Nous avons donc choisis de nous tourner vers un algorithme de recherche de plus cours chemin : dans notre cas, un A*. Le principe est simple puisque le but est de rechercher tout simplement quelle est la trajectoire la plus optimisée pour aller d'un point A à un point B (appelée nodes) tout en évitant les obstacles.

L'algorithme a besoin d'une heuristique, c'est-à-dire d'une fonction qui estime le coût du trajet d'un point jusqu'au point d'arrivée. Cette heuristique ne doit pas surestimer le coût pour être « admissible ». Dans notre cas, le coût heuristique est une fonction simple : la distance entre le point donné et le point d'arrivée. Cette heuristique sous-estime le coût du trajet dans la plupart des cas, mais ne le surestime jamais.
De plus, l'algorithme doit pouvoir s'exécuter sur un graphe contenant les nœuds accessibles. Ce graphe est généré à la volée : pour un nœud (une case de la map), les nœuds associés sont ses voisins directs, si ceux-ci sont traversables.

La structure de données que nous avons utilisée pour le stockage de la carte (un tableau à 2 dimensions) était particulièrement bien adaptée au développement de cet algorithme, l'accès à un nœud quelconque étant en O(1) et l'accès aux voisins disponibles également.

CONCLUSION

Au final, le temps de conception et le choix que nous avons fait nous ont permis d'atteindre nos objectifs sans avoir à repenser l'architecture du projet à chaque itération. Nous avons fait en sorte qu'une fois le moteur terminé, nous puissions ajouter des fonctionnalités facilement et rapidement. Notamment au niveau de la gestion des joueurs, où il est même possible de créer des contrôleurs permettant la gestion du jeu en réseau (non implémenté faute de temps). Notre méthode de test a aussi été très efficace, et nous a permis de localiser rapidement les bugs et de les corriger.

Notre seul regret concerne le manque de temps que nous avons eu pour finaliser le projet (équilibrage, ajout d'éléments de jeu), que nous n'avions pas prévu.