Tower Defense

Mise en œuvre d’un graphe interactif

Création d’une variante du célèbre jeu Tower Defense dans le cadre d’un cours d’algorithmique.

ASD2

Lazhar Farjallah / Aurélien Da Campo / Pierre-Dominique Putallaz

Heig-vd

ASD

TD

Table des matières

[1 Analyse préliminaire 3](#_Toc250559539)

[1.1 Introduction [LAZHAR] 3](#_Toc250559540)

[1.2 Organisation [LAZHAR] 3](#_Toc250559541)

[1.3 Objectifs [LAZHAR] 4](#_Toc250559542)

[1.4 Planification initiale [AURELIEN] 5](#_Toc250559543)

[2 Analyse 6](#_Toc250559544)

[2.1 Concepts algorithmiques [LAZHAR] 6](#_Toc250559545)

[2.1.1 Outils algorithmiques 6](#_Toc250559546)

[2.1.2 Complexité 8](#_Toc250559547)

[2.2 Etude de concurrentielle [AURELIEN] 8](#_Toc250559548)

[2.3 Etude de faisabilité [AURELIEN] 8](#_Toc250559549)

[2.4 Planification [AURELIEN] 9](#_Toc250559550)

[3 Conception 10](#_Toc250559551)

[3.1 Dossier de conception 10](#_Toc250559552)

[3.1.1 Systèmes d’exploitation [LAZHAR] 10](#_Toc250559553)

[3.1.2 Outils logiciels [LAZHAR] 10](#_Toc250559554)

[3.1.3 Librairies externes [LAZHAR] 10](#_Toc250559555)

[3.1.4 Architecture de l’application [AURELIEN] 10](#_Toc250559556)

[3.1.5 Interface graphique [AURELIEN] 10](#_Toc250559557)

[3.1.6 Schémas UML [PIERRE-DO] et [AURELIEN] 10](#_Toc250559558)

[3.1.7 Gestion de la concurrence [PIERRE-DO] 11](#_Toc250559559)

[4 Réalisation 12](#_Toc250559560)

[4.1 Dossier de réalisation 12](#_Toc250559561)

[4.1.1 Résultat de l’interface graphique [AURELIEN] 12](#_Toc250559562)

[4.1.2 Résultat de l’implémentation du maillage [PIERRE-DO] 12](#_Toc250559563)

[4.2 Description des tests effectués [LAZHAR] 12](#_Toc250559564)

[4.3 Erreurs restantes [LAZHAR] 12](#_Toc250559565)

[5 Conclusions *[LAZHAR]* 12](#_Toc250559566)

[5.1 Objectifs atteints / non-atteints 12](#_Toc250559567)

[5.2 Points positifs / négatifs 12](#_Toc250559568)

[5.3 Difficultés particulières 12](#_Toc250559569)

[5.4 Avenir du projet 13](#_Toc250559570)

[6 Annexes 14](#_Toc250559571)

[6.1 Sources – Bibliographie [LAZHAR] 14](#_Toc250559572)

[6.2 Journal de bord de chaque participant [LAZHAR] 14](#_Toc250559573)

[6.3 Manuel de TDA maillage [PIERRE-DO] 14](#_Toc250559574)

[6.4 Manuel d'Utilisation [LAZHAR] 14](#_Toc250559575)

[6.5 Archives du projet [LAZHAR] 14](#_Toc250559576)

# Analyse préliminaire

## Introduction [LAZHAR]

Ce projet de fin de semestre consiste à créer une application ludique mettant en œuvre des algorithmes et structures de données étudiées en cours. Nous avons choisi pour cela de créer une variante du célèbre jeu « Tower Defense », dans lequel des personnages se déplacent d’un point A à un point B selon un chemin optimal. En effet, on aura besoin pour cela d’une structure de graphe ainsi que les algorithmes associés, ce qui colle parfaitement avec la contrainte de départ car nous les avons étudiés en cours.

Le but est également ici de créer une application « didacticiel » (il s'agit d'un logiciel interactif destiné à l'apprentissage de savoirs) qui sera présentée dans les futurs cours de cette unité d’enseignement. Ce projet permettra en effet de montrer une application réelle de l’utilisation d’algorithmes associés à des graphes (en particulier celui de recherche du chemin le plus court entre deux nœuds).

Nous commencerons par effectuer une analyse du projet, notamment en ce qui concerne l’organisation, les objectifs ainsi que la planification initiale. Puis nous détaillerons les étapes de conception et de réalisation. Enfin, avant de conclure, nous présenterons les fonctionnalités de l’application finale sous forme de mode d’emploi et nous présenterons des captures d’écran.

## Organisation [LAZHAR]

Les membres participant à ce projet sont :

**Etudiant 1** :

Aurélien Da campo, *aurelien.dacampo@heig-vd.ch*

**Etudiant** **2** :

Pierre-Dominique Putallaz, *pierre-dominique.putallaz@heig-vd.ch*

**Etudiant 3 (responsable de projet)** :

Lazhar Farjallah, *lazhar.farjallah@heig-vd.ch*

Nous prévoyons de nous répartir les tâches de manière suivante entre chaque entité du groupe :

1. *Responsable de projet*
   1. Rédactions, administrations
   2. Suivi des rendus (*deadlines*)
   3. Surveillance et coordination
   4. Développement
2. *Etudiant 1*
   1. Création de l’interface graphique
   2. Gestion de l’affichage
   3. Rendu graphique
   4. Interaction avec l’utilisateur
3. *Etudiant 2* 
   1. Algorithmique
   2. Implémentation des algorithmes de graphe
   3. Fournir les briques logicielles pour permettre la construction de la partie fonctionnelle de l’application

## Objectifs [LAZHAR]

Les objectifs de ce projet sont les suivants :

* Illustrer le concept de graphe de manière ludique et interactive.
* Acquérir de l’expérience dans la planification et l’accomplissement d’un projet conséquent.
* Utiliser et découvrir des librairies existantes implémentant le concept de graphe
* Apprendre à mettre en œuvre une interface graphique en Java.
* Séparer le travail en plusieurs niveaux d’abstraction pour faciliter l’élaboration et l’évolutivité de ce projet.
* Comprendre la nécessité d’utiliser des algorithmes complexes dans les applications informatiques.
* Mettre en œuvre un algorithme de recherche de chemin le plus court (ACPC).
* Respecter le design pattern MVC (*Model* – *View* – *Controller*) qui structure un programme en trois couches principales.

## Planification initiale [AURELIEN]

Le projet se déroulera du 18 novembre 2009 au 15 janvier 2010, ce qui représente un total de 25 périodes en classe (par personne). Nous prévoyons également de passer un total d’environ au moins 25 périodes par personne en dehors des heures encadrées.

Au total, c’est environ 150 périodes de travail que nous allons planifier comme suit :

*Ce chapitre montre la planification du projet. Celui-ci peut être découpé en tâches qui seront planifiées. Il s'agit de la première planification du projet, celle-ci devra être revue après l'analyse. Cette planification sera présentée sous la forme d'un diagramme de Gantt et/ou de PERT (l'utilisation de MS project est conseillée).*

*Ces éléments peuvent être repris des spécifications de départ.*

# Analyse

## Concepts algorithmiques [LAZHAR]

Le principal concept algorithmique utilisé dans ce projet est celui de graphes. En effet, ce concept, bien connu dans le domaine algorithmique, constitue un point clé dans la réalisation de notre cahier des charges. Il intervient principalement dans le problème de la recherche d’un chemin (le plus court) entre deux points donnés A et B.

Si on regarde à un niveau plus abstrait, le but premier de notre application est de lancer des objets (créatures) à partir d’un point A pour qu’ils se dirigent vers un point B selon un chemin optimal, calculé à la volée en fonction d’un graphe qui ne cesse de se modifier (dynamique).

Nous avons également besoin d’un algorithme de tri afin de trier les meilleures scores du jeu obtenus et de les afficher dans l’ordre.

Cette simple vision est tout à fait suffisante pour comprendre les outils algorithmiques dont nous devons s’armer.

Nous allons maintenant développer plus en détails quels sont les structures et algorithmes de graphes dont nous avons besoin.

### Outils algorithmiques

Partant du constat du paragraphe 2.1, on comprend assez facilement que nous aurons besoin des structures suivantes :

* un graphe pondéré non orienté
* des nœuds composant le graphe
* des arcs pour relier les différents nœuds du graphe
* un algorithme de recherche d’un chemin optimal entre deux nœuds (connu sous l’appellation « algorithme de Dijkstra ») appliqué à un graphe pondéré non orienté
* un algorithme de tri afin de trier les meilleures scores des joueurs

#### Graphe pondéré non orienté

Nous avons décidé d’utiliser un tel graphe car il correspond parfaitement à ce dont nous avons besoin. En effet, chaque arc possède un poids (pondéré), car une créature doit savoir à partir de n’importe quel nœud du graphe quel est la longueur du chemin jusqu’au prochain nœud. De plus, nous avons choisi d’utiliser un graphe non orienté, c’est-à-dire que les arcs n’ont pas de sens (si un nœud A est voisin d’un nœud B alors B est aussi un voisin de A). Les nœuds sont donc traversables en double sens, ce qui se prête bien à nos besoins car il se peut très bien qu’une créature doive tout d’un coup reculer et revenir sur ces pas.

#### Nœuds composants le graphe

Les nœuds de notre graphe contiennent des informations telles que la position x et y (en fonction du repère cartésien du jeu) afin de connaître le parcours à suivre en coordonnées cartésiennes, mais aussi un « drapeau » (*flag* en anglais) permettant de savoir si un tel nœud est actif ou non.

#### Arcs pour relier les nœuds du graphe

Les arcs reliant les nœuds du graphe sont pondérés, c’est-à-dire qu’ils possèdent une valeur associée (un nombre) strictement positif déterminant la longueur (tout simplement) de l’arc en question. Cette longueur est calculée selon la position cartésienne des extrémités de l’arc (autrement dit à partir des 2 nœuds reliés par l’arc) grâce au théorème de Pythagore. Ces arcs sont également non orientés, c’est-à-dire qu’ils ne possèdent pas de sens (chaque arc peut être parcouru dans n’importe quel sens).

#### Algorithme de dijkstra (sur un graphe pondéré non orienté)

Voici, en pseudo-code, l’algorithme de recherche de tous les chemins les plus courts à partir d’un sommet initial vers tout autre sommet du graphe (un sommet est un synonyme de nœud) que nous utilisons:

*(partie générale)*

marquer tous les sommets comme non visités

visiter le sommet initial

*(partie spécifique à un sommet : visiter le sommet)*

**si** le sommet est non visité **alors**

déposer le sommet avec priorité nulle dans la queue de

priorité

**tant que** la queue de priorité n’est pas vide **boucler**

prélever le sommet de tête (il devient le sommet courant)

marquer le sommet courant comme visité

traiter le sommet courant

**pour** chaque sommet voisin du sommet courant **boucler**

**si** le sommet voisin est non visité **alors**

calculer la priorité du sommet voisin (c’est-à-dire

prendre la somme de l’attribut de l’arc, entre le

sommet courant et lui-même, et de la priorité du

sommet courant)

déposer le sommet voisin et sa priorité dans la

queue de priorité

**fin si**

**fin boucler**

**fin boucler**

**fin si**

Algorithme 2.1 : recherche des chemins les plus courts dans un graphe (Dijkstra).

#### Algorithme de tri pour le tri des meilleurs scores des joueurs

Pour trier les meilleurs scores obtenus par le joueurs, nous utilisons la méthode sort() disponible dans l’API Java, applicable sur un objet de type ArrayList (une collection). En effet, Java utilise pour ce faire le tri Merge Sort (tri fusion) dont voici le pseudo-code :

procedure triFusionI(entier[] tab)

entier[] tmp <- tableau de taille N;

entier i <- 1;

entier debut <- 1;

entier fin <- debut + i + i - 1;

tant que (i < N) faire

debut <- 1;

tant que (debut + i -1 < N) faire

fin <- debut + i + i - 1;

si (fin > N) alors

fin <- N;

fusion(tab, tmp, debut, debut + i - 1, fin);

debut <- debut + i + i;

fin tant que

i <- i + i;

fin tant que

fin procedure

Algorithme 2.2 : tri fusion sur un tableau d’entiers.

### Complexité

#### Algorithme de Dijkstra

Si le graphe possède *m* arcs et *n* nœuds, en supposant que les comparaisons des poids d'arcs soient à temps constant, et que le tas soit binomial, alors la complexité de l'algorithme est .

#### Tri fusion

On voit que la procédure de fusion nécessite un tableau intermédiaire aussi grand que le nombre d'éléments à trier. C'est en effet le principal inconvénient du tri fusion, car sa complexité est dans tous les cas en, (prix du tableau annexe aussi grand que le tableau initial).

## Etude de concurrentielle [AURELIEN]

## Etude de faisabilité [AURELIEN]

*Détailler les 3 a****s****pects de l'étude de faisabilité:*

* *risques techniques (complexité, manque de compétences, …).*
* *risques concernant le planning & les ressources humaines.*
* *risques concernant le budget.*

*Décrire aussi quelles solutions ont été appliquées pour réduire les risques (priorités, formation, actions, …).*

## Planification [AURELIEN]

*Révision de la planification initiale (Gantt et/ou PERT) du projet :*

* *planning indiquant les dates de début et de fin du projet ainsi que le découpage connu des diverses phases.*
* *partage des tâches en cas de travail à plusieurs.*

*Il s’agit en principe de la planification* ***définitive du projet****. Elle peut être ensuite affinée (découpage des tâches). Si les délais doivent être ensuite modifiés, le responsable de projet doit être avisé, et les raisons doivent être expliquées dans l’historique.*

# Conception

## Dossier de conception

### Systèmes d’exploitation [LAZHAR]

Le système d’exploitation sur lequel nous travaillons est Windows (*XP* et *Seven*). Cependant, grâce au choix qui a été fait d’utiliser un encodage de type UTF-8 ainsi que celui du langage portable Java, nous pouvons sans soucis travailler sur un environnement Linux ou Mac par exemple.

### Outils logiciels [LAZHAR]

Nous développons avec l’IDE Eclipse (version 3.4 et supérieure) intégrant tous les outils nécessaires au développement d’applications Java. Ce logiciel est très largement répandu dans le monde des développeurs et est très utilisé. Il possède de nombreuses fonctions spécialement conçues pour augmenter la productivité des développeurs et leur simplifier la vie (comme le *refactoring* par exemple). De plus, cette plateforme nous permet d’ajouter toute une série de plugins qui peuvent ajouter des fonctionnalités, telles que SVN (logiciel de gestion des versions du code). En ce qui concerne la génération des diagrammes de classe UML, nous utilisons le plugin ***eUML 2.0*** de chez ***Soyatec***.

### Librairies externes [LAZHAR]

#### JGraphT

Nous utilisons la librairie (libre) externe ***JGraphT*** codée en Java. En effet, cette librairie possède toutes les briques logicielles nécessaires à la création de graphes (de tout type). Nous l’utilisons comme une boîte noire sans se soucier de son implémentation. Le choix de cette librairie n’a pas été sans réflexion. En effet, après quelques recherches effectuées sur Internet ainsi qu’une discussion avec le professeur, cette librairie nous parait correcte.

#### JLayer

Nous utilisons la librairie (libre) externe ***JLayer*** codée en Java. Cette librairie nous permet de jouer des musiques codées en divers formats tel que le mp3 par exemple (ou le wav). Elle sert uniquement à jouer de la musique, ce qui apporte un petit plus au projet.

### Architecture de l’application [AURELIEN]

Nous avons choisi comme architecture du programme le fameux design pattern MVC (Modèle – Vue – Contrôleur).

### Interface graphique [AURELIEN]

### Schémas UML [PIERRE-DO] et [AURELIEN]

#### Maillage [PIERRE-DO]

#### Interface graphique (vue et controleur) [AURELIEN]

#### Gestionnaire du jeu (model) [AURELIEN]

### Gestion de la concurrence [PIERRE-DO]

*Fournir tous les document de conception:*

* *le choix du matériel HW*
* *le choix des systèmes d'exploitation pour la réalisation et l'utilisation*
* *le choix des outils logiciels pour la réalisation et l'utilisation*
* *site web: réaliser les maquettes avec un logiciel, décrire toutes les animations sur papier, définir les mots-clés, choisir une formule d'hébergement, définir la méthode de mise à jour, …*
* *bases de données: décrire le modèle relationnel, le contenu détaillé des tables (caractéristiques de chaque champs) et les requêtes.*
* *programmation et scripts: organigramme, architecture du programme, découpage modulaire, entrées-sorties des modules, pseudo-code / structogramme…*

***Le dossier de conception devrait permettre de sous-traiter la réalisation du projet !***

# Réalisation

## Dossier de réalisation

### Résultat de l’interface graphique [AURELIEN]

### Résultat de l’implémentation du maillage [PIERRE-DO]

*Décrire la réalisation "physique" de votre projet*

* *les répertoires où le logiciel est installé*
* *la liste de tous les fichiers et une rapide description de leur contenu (des noms qui parlent !)*
* *les versions des systèmes d'exploitation et des outils logiciels*
* *la description exacte du matériel*
* *le numéro de version de votre produit !*
* *programmation et scripts: librairies externes, dictionnaire des données, reconstruction du logiciel - cible à partir des sources.*

*NOTE : Evitez d’inclure les listings des sources, à moins que vous ne désiriez en expliquer une partie vous paraissant importante. Dans ce cas n’incluez que cette partie…*

## Description des tests effectués [LAZHAR]

### Génération du maillage (graphe associé) avec *JGraphT*

Pour ce test, nous avons mesuré le temps de génération du graphe associé à un terrain (une « map ») du jeu. Nous avons décidé d’effectuer ce test avec un graphe doté d’un nœud tous les 2 pixels, ainsi que 8 arcs par nœuds, répartis uniformément. Les nœuds sont tous reliés les un aux autres.

Voici un schéma du graphe généré pour un terrain de 4x4 pixels :

x [pixels]

y [pixels]

(0,0)

(2,0)

(4,0)

(0,2)

(0,4)

(4,4)

noeuds

arcs

Figure 4.2.1 : graphe généré avec un nœud tous les 2 pixels.

Le résultat, après avoir pris les mesures correspondantes, figurent sous deux formes. Dans la première, on mesure le temps de génération du maillage en fonction d’une taille de terrain en pixels (par exemple 800x600). Dans la seconde, on mesure l’espace mémoire nécessaire en fonction toujours de la taille du terrain en pixels.

Voici les résultats obtenus, sous forme de graphiques :

Figure 4.2.1.1 : temps de génération d’un maillage (graphe).

Figure 4.2.1.2 : mémoire utilisée lors de la génération d’un maillage (graphe).

En conclusion de ce premier test, on peut voir plusieurs choses intéressantes. Tout d’abord, rappelons que nous utilisons pour générer les maillages (graphes) une librairie externe, d’où l’utilité de ces tests, car nous ne sommes que des utilisateurs de cette librairie.

Premièrement, on voit sur le premier graphique que le temps de génération d’un maillage est en , grâce à la fonction y d’approximation. Il en va de même pour le second graphique, cette fois pour la mémoire. Cette première conclusion n’est pas surprenante, puisque pour créer le maillage, il faut créer n nœuds puis ensuite pour chaque nœud, il faut créer m arcs.

Deuxièmement, on voit qu’au niveau du temps de génération ainsi que de la mémoire, il va falloir choisir une valeur « correcte », c’est-à-dire une valeur optimale de la taille en pixels du terrain pour optimiser le temps et la mémoire lors de la génération du maillage, qui se fait au lancement de notre application. Nous avons donc choisi d’utiliser un terrain de jeu composé d’au maximum 500x500 pixels. En effet, d’après les graphiques, cela signifie que le temps de génération est d’environ 1 seconde et la mémoire utilisée d’environ 30 mégaoctets, ce qui est relativement raisonnable.

Cependant, pour baisser encore ces valeurs, nous avons décidé de créer finalement un nœud tous les 10 pixels, et non plus un nœud tous les 2 pixels. De ce fait, nous aurons un graphe qui aura 5 fois moins de nœuds et d’arcs que le graphe que nous générons dans ce test (car pour une zone de 10x10 pixels, on avait avant 5 nœuds, alors que maintenant on en a plus qu’un). On peut alors s’attendre à un temps de génération d’environ 0.2 seconde ainsi qu’une mémoire utilisée d’environ 6 mégaoctets, ce qui est cette fois parfaitement raisonnable.

### Recherche du chemin le plus court

Dans ce test, nous nous bornerons à faire des captures d’écran de l’application et à démontrer le bon fonctionnement de l’algorithme de recherche de chemin le plus court (*Dijkstra*).

Voici une série de trois captures d’écran illustrant les trois situations les plus intéressantes, suivies d’explications :

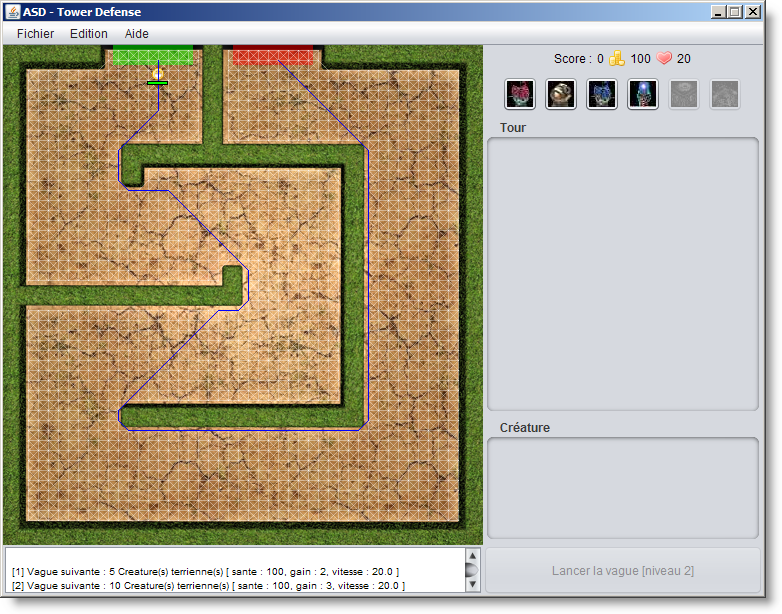


Figure 4.2.2.1 : terrain de jeu avec le chemin le plus court en bleu, situation 1.

Dans cette première situation, aucun obstacle n’est placé. Le chemin optimal est celui tracé en bleu. On voit nettement que ce chemin est le plus court. Cette situation est la plus triviale et ne nécessite pas de remarques particulières, si ce n’est que l’algorithme de recherche du chemin optimal fonctionne parfaitement.

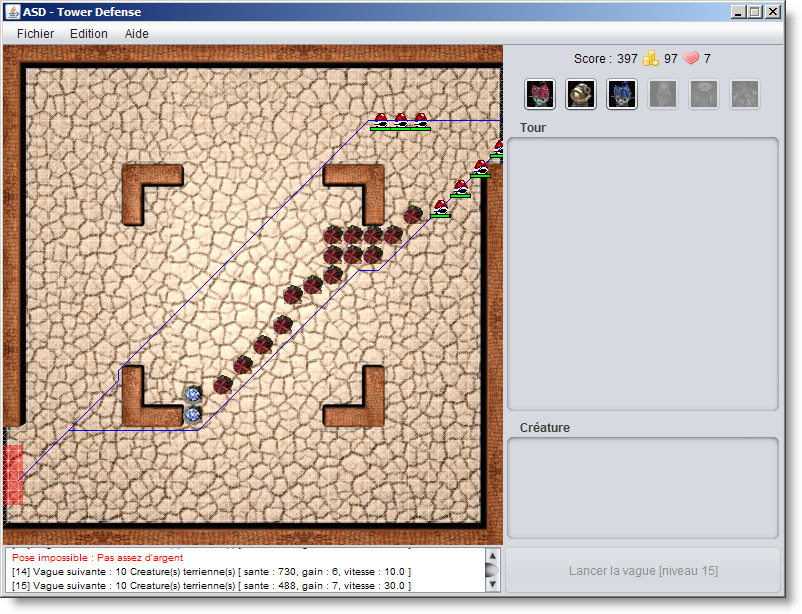


Figure 4.2.2.2 : terrain de jeu avec le chemin le plus court en bleu, situation 2.

Dans cette deuxième situation, il est intéressant de constater qu’il peut y avoir plusieurs chemins. En fait, c’est parce qu’on peut bouger dynamiquement les tours faisant office d’obstacle pendant le jeu. Ici par exemple, les trois premières créatures en haut avaient à la base un autre chemin optimal, en fonction des tours qui étaient placées à ce moment-là. Puis, après avoir supprimé/ajouté des tours, les quatre créatures suivantes se sont attribuées un nouveau chemin. Il est donc intéressant de voir que la recherche du chemin optimal est bel et bien dynamique : le chemin peut varier en tout temps. De plus, on constate que chaque créature possède son propre chemin optimal, toutes les créatures n’étant pas au même endroit (c’est-à-dire au même nœud en terme de graphe) à un temps donné *t*.

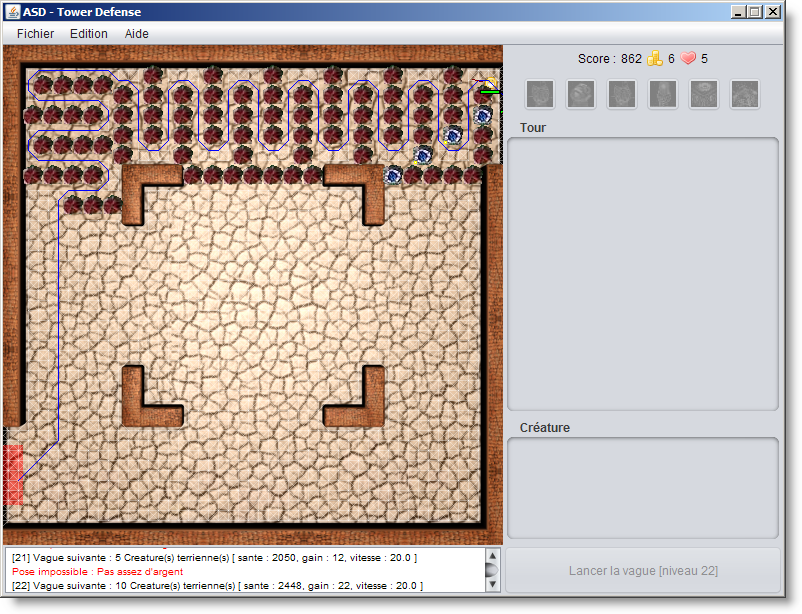


Figure 4.2.2.3 : terrain de jeu avec le chemin le plus court en bleu, situation 3.

Dans cette troisième et dernière situation, on voit que le joueur a pris le soin de créer une espèce de labyrinthe. Comme les créatures doivent se rendre du point de départ (en haut à gauche) jusqu’à l’arrivée (en bas à droite) coûte que coûte, elles prendront le meilleur chemin existant. En effet, ici elles seront obligées de parcourir tout le labyrinthe, car le chemin résultant, malgré qu’il soit long, devient un chemin optimal. De plus, on voit nettement qu’à la sortie du labyrinthe créer par le joueur, le chemin est à nouveau « livré à lui-même » car il n’y a plus aucun obstacle (d’où la ligne droite allant vers l’arrivée qu’on voit après la sortie du labyrinthe).

### Enregistrement des scores et leur tri

Lorsqu’un joueur termine une partie, son score est sauvegardé dans un fichier « *sérializé »* sur le disque dur dans le dossier /donnees. Nous allons ici simplement tester cette fonctionnalité et fournir des captures d’écran démontrant son bon fonctionnement.

Voici un schéma de jeu classique :

1. Le joueur lance la partie et joue jusqu’à ce qu’il perde.
2. Le joueur a perdu ; il est invité à inscrire son nom.
3. Le joueur est ensuite invité à consulter la liste des dix derniers meilleurs scores classés par ordre décroissant (le meilleur score en premier), le score qu’il vient d’obtenir étant lui aussi inclus.
4. Sans le voir directement, les scores sont automatiquement sauvegardés dans un fichier sur le disque dur pour que le joueur puisse à tout moment revoir ses anciens scores.

Voici à présent trois captures d’écran destinées à démontrer les étapes 2, 3, et 4 décrites ci-dessus :

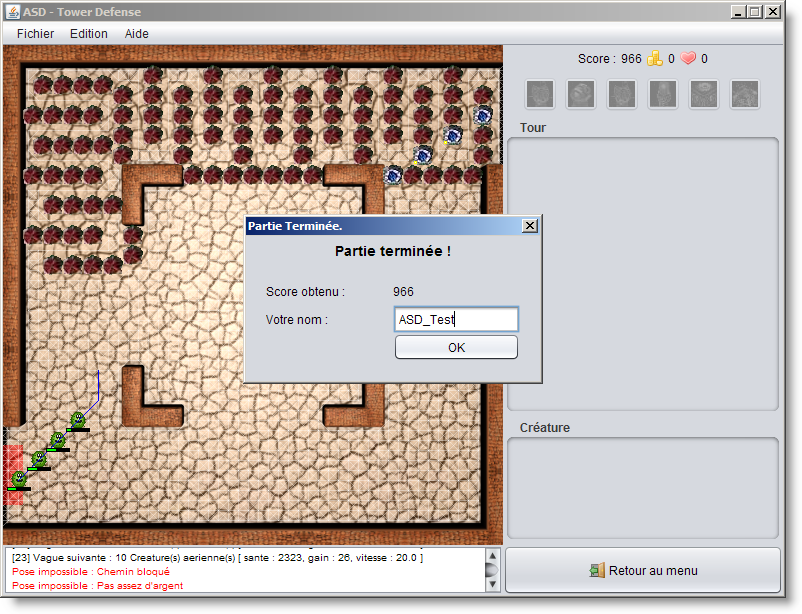


Figure 4.2.3.1 : le joueur a perdu ; il est invité à inscrire son nom.

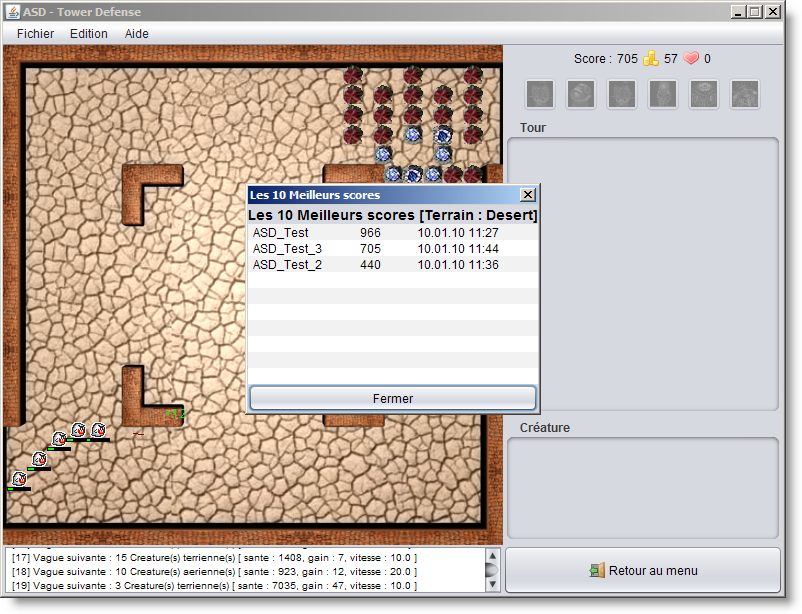


Figure 4.2.3.2 : le joueur voit la liste des 10 derniers meilleurs scores triés.

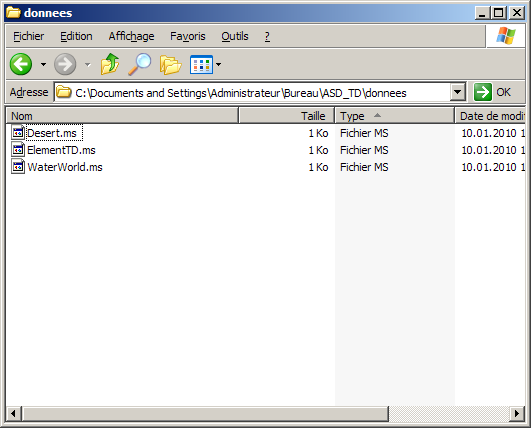


Figure 4.2.3.3 : Les scores sont sauvegardés dans le dossier /donnees.

### Test global de l’application

Nous avons décidé, peu de temps avant le rendu de ce laboratoire, de diffuser, en « *beta-test »*, une version beta de notre jeu sur Internet, à l’adresse http://code.google.com/p/asd-tower-defense/. Ainsi, après avoir envoyé quelques e-mails, plusieurs personnes ont joué à notre jeu et nous ont fait des petits feedbacks.

Voici par exemple la capture d’écran qu’un contact nous a envoyé, pendant qu’il testait notre application :



Figure 4.2.4.1 : Capture d’écran fournie par un de nos « beta-testeurs ».

*Pour chaque partie testée de votre projet, il faut décrire:*

* *les conditions exactes de chaque test*
* *les preuves de test (papier ou fichier)*
* *tests sans preuve: fournir au moins une description*

## Erreurs restantes [LAZHAR]

*S'il reste encore des erreurs:*

* *Description détaillée*
* *Conséquences sur l'utilisation du produit*
* *Actions envisagées ou possibles*

# Conclusions *[LAZHAR]*

## Objectifs atteints / non-atteints

## Points positifs / négatifs

## Difficultés particulières

## Avenir du projet

# Annexes

## Sources – Bibliographie [LAZHAR]

*Liste des livres utilisé (Titre, auteur, date), des sites Internet (URL) consultés, des articles (Revue, date, titre, auteur)… Et de toutes les aides externes (noms)*

## Journal de bord de chaque participant [LAZHAR]

## Manuel de TDA maillage [PIERRE-DO]

## Manuel d'Utilisation [LAZHAR]

## Archives du projet [LAZHAR]

*CD, … dans une fourre en plastique*