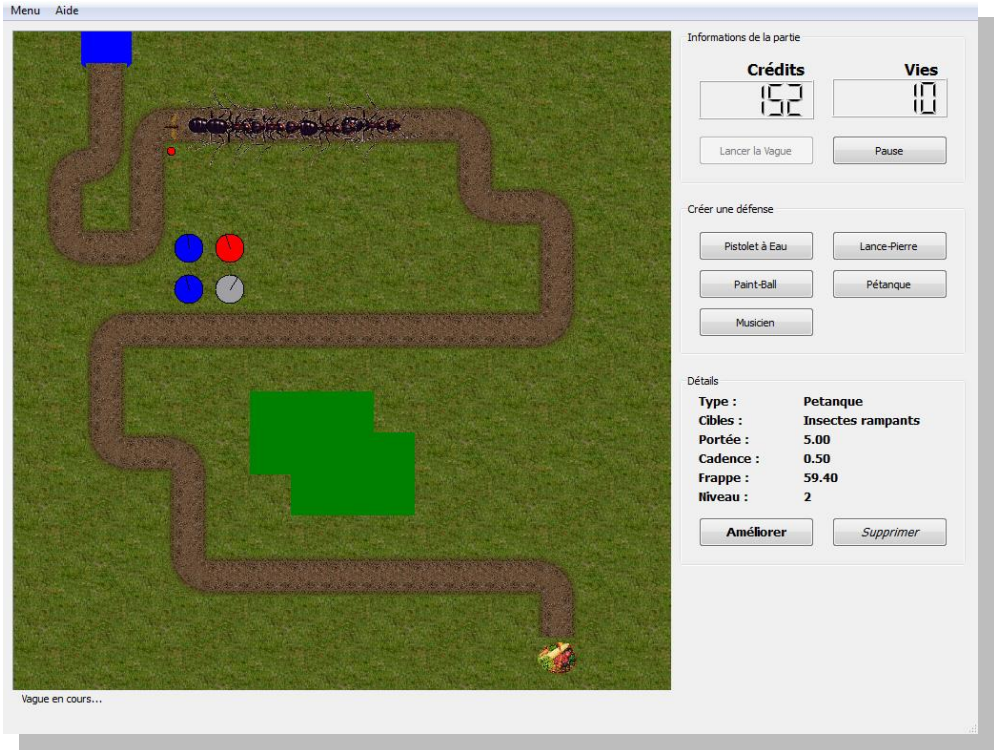
Rapport de projet LO21 – P11

Réalisation d'un Tower Defense



Devaux Tony
Donin Stéphan

Introduction

Description du projet

Dans le cadre de l'UV LO21 - Programmation et conception orientées objet, il nous a été demandé de réaliser un jeu de Tower Defense.

Le but de ce projet est de mettre en pratique les différents aspects théoriques vus en cours ou en TD (héritage, design patterns,…) en utilisant le framework Qt.

Le but du jeu est de réussir à détruire des vagues successives d'ennemis (ici des insectes) qui doivent atteindre un endroit de la carte (notre pique-nique) en suivant un chemin précis (fourmis à travers un fichier texte).

Afin d'empêcher l'invasion, les défenseurs (des étudiants) ont à leur disposition différentes armes (pistolet de paint-ball, boules de pétanque,…).

Complément d'information

Ce rapport est destiné à compléter la documentation du code source générée automatiquement sous Doxygen.

Le diagramme de classes est également disponible en annexe de ce rapport.

Conventions d'écriture de code

On respectera les conventions d'écriture de code données dans le cadre du TP2. Ainsi, on aura les propriétés suivantes :

- Une variable commence par une minuscule et introduit une majuscule si elle est composée de plusieurs mots
  Exemple : maVariable
- Une classe commence par une majuscule et introduit une autre majuscule si son nom est composé de plusieurs mots
  Exemple : MaClasse
- Une constante sera écrite uniquement en majuscule. On pourra introduire un tiret-bas si son nom est composé de plusieurs mots, mais on évitera cependant ce cas au maximum
  Exemple : UNE_CONSTANTE
- Les méthodes s'écriront comme les classes.
  Exemple : MaMethode()
- Les commentaires seront à placer dans les fichiers d'en-tête (*.h) et respecteront la norme Doxygen pour le C++.
- Les commentaires et le code seront écrits en français.

- Tout ce qui sera développé dans le cadre de ce projet sera sous le namespace TOWERDEFENSE.
-

Pour ce qui est de la structure du code, on respectera la syntaxe suivante :

```
Type Nom_module(param….)
{
Private :
        Element1 ;
        Element2 ;
        …
Public :
        Element3
        Element4
        ….
} ;
```

Composition et fonctionnements de l'interface graphique avec Qt.

La fenêtre principale de ce programme sera un objet héritant de la classe *QWidget*. Cette dernière contient d'autre *QWidget*, notamment deux principaux : l'un contenant toute l'interface graphique et les animations de jeu, et l'autre contenant tous les boutons de contrôle pour le jeu (les boutons pour ajouter des défenses, lancer des vagues d'ennemis, etc...).

Parlons tout d'abord du *QWidget* contenant toutes les animations. Ce dernier fonctionnera avec des objets de type *QGraphicsScene*, *QGraphicsItem* et *QgraphicsView* ; le choix de ces classes pour l'animation a été conseillé par l'énoncé du projet, certainement dû à son traitement des animations optimisé et aux méthodes implémentées par la classe pour tout ce qui est interactions avec la souris (clic, survol, etc…). Ces classes fonctionnent de la façon suivante : le scène principale des animations est représentée par un objet de type *QGraphicsScene*, qui contient plusieurs objets de type *QGraphicsItem*. Ceci sera totalement « théorique » dans le sens où, à eux seuls, ces objets ne permettront pas un affichage graphique. Pour permettre l'affichage d'une scène, on associe notre scène à un objet de type *QGraphicsView*, qui s'occupera donc d'afficher comme il le faut tous les objets contenus dans la scène.
Une fois ces objets construits, il faudra alors remplir notre scène : pour ceci, nous allons nous occuper d'un objet de type *Carte*, que nous auront créé au préalable, s'occupant de la lecture d'un fichier texte représentant une carte donnée. À l'aide de cet objet, nous créerons donc nos différents objets issus de la lecture de la carte fournie. On ajoutera ainsi à la carte le chemin tracé, la boue, le but des ennemis et le départ de ces derniers.
Les objets de type *QGraphicsItem* nous permettent de coller à ces derniers des informations de façon plutôt libre ; ainsi, un objet *QGraphicsItem* pour avoir un tableau à deux dimensions, avec un index numérique à gauche, et une donnée (de type varié) à droite. Ceci nous aidera pour identifier les différents objets de la carte.

Concernant le *QWidget* de droite, il sera tout simplement composé de boutons et autres objets graphiques permettant le contrôle du déroulement du jeu.

Par défaut, seule la suppression d'objet est activée ; ainsi, en cliquant sur un objet de la carte, on pourra l'identifier grâce aux données associées aux *QGraphicsItem.* De là, deux solutions s'offrent à nous : soit le joueur aura cliqué au préalable sur un des boutons d'ajouts d'éléments de défense, et un clic aura alors comme conséquence de placer un élément sur la carte, à condition qu'il ne soit pas localisé sur le chemin, la boue, le but ou le départ ennemi ; soit le joueur cliquera sur un élément de défense déjà créé et le clic aura alors comme conséquence la suppression de cet élément.
Pour la communication des objets entre eux, on utilisera le système de slots et de signaux offerts par le langage C++.

Mécanismes du jeu

Principales classes

Plusieurs classes d'objet sont présentes au sein de notre scène.

Ennemis

Les insectes sont représentés par une classe *ennemi*.

Les différents types d'ennemis dérivent de cette classe, avec un certain nombre de fonction à réimplémenter (fonctions virtuelles pures).

Défenseurs

Les défenseurs sont représentés par une classe *defenseur*.

Comme pour les ennemis, les différents types de défenseurs dérivent de la classe mère.

Projectiles

Une classe *projectile* a été créé afin de gérer la partie tir et collision du jeu.

Vagues

Les vagues sont constituées d'une succession d'ennemis, qui sont créés à intervalles prédéfinis. Chaque vague est ainsi composée d'un ensemble d'ennemis.

Création des éléments - Factory

Afin de pouvoir créer facilement éléments du jeu, le design pattern Factory a été implémenté pour ces 3 classes objets.

Une méthode permet de récupérer un nouvel élément, en se chargeant d'effectuer toutes les opérations annexes (affichage dans la scène, insertion dans les mécanismes,…).

Système central - Singleton

Afin d'assurer le bon fonctionnement du jeu et la liaison entre les mécanismes du jeu et l'interface, une classe Singleton a été créé.

Cette classe permet de gérer les opérations de base du jeu (chargement de la carte, des vagues, lancement de nouvelles parties,…).

Elle permet également aux classes objet d'interagir avec l'interface via le système de signaux/slots. Le passage de ces signaux via le Singleton a été utilisé afin de limiter la complexité du code au niveau des différentes classes. L'interface n'a qu'à écouter le Singleton, et les objets n'ont qu'à effectuer leurs demandes auprès du singleton.

Principaux algorithmes

Suivi de chemin

Le suivi du chemin s'effectue au niveau de chaque ennemi.

Lorsque la méthode *advance()* d'un ennemi est appelée, celui-ci repère sa position sur la carte.

Il regarde ensuite la direction dans laquelle il doit se déplacer en fonction de la nature de la case (vers le bas si la case est dirigée vers le sud,…).

En connaissant sa vitesse de déplacement, il est ensuite possible de calculer la nouvelle position à rejoindre afin de se déplacer sur le chemin.

Détection des ennemis

Afin de défendre leur territoire, il est nécessaire que les défenseurs détectent les ennemis en approche.

Dans cette optique, les défenseurs peuvent récupérer une liste des ennemis qui se trouvent à la portée, à travers la EnnemiFactory.

En connaissant la position du défenseur ainsi que sa portée, il est aisé de déterminer si un ennemi peut être attaqué par un défenseur, en fonction de la distance les séparant (la EnnemiFactory se charge de ces opérations).

Une fois la liste des cibles potentielles récupérée, le défenseur peut choisir la cible à viser en fonction de différents critères (plus fort, plus faible, plus vieux, plus rapide,…).

Lorsque la cible est déterminée, un projectile est lancé en direction de la cible.

Détection de la collision des projectiles et des ennemis

Malheureusement, les défenseurs ne sont pas parfaits, et ils ne sont pas assurés de toucher les cibles à chaque fois.

Lorsque les projectiles sont créés, ils sont déplacés lors de chaque tick du timer. Une détection de la collision entre les ennemis et les projectiles, basée sur la méthode collidesWithItem() héritée des QGraphicsItem.

Si une collision est détectée, le projectile a atteint une cible (pas forcément celle visée au départ !), et l'ennemi se voit alors infliger des dégâts en fonction de la puissance du projectile.