Rapport : simulation d'un dodgeball

1. Introduction

Dans ce qui suit, nous présentons notre implémentation de la simulation d'une partie de dodgeball en nous concentrons sur le processus de développement du projet, son architecture, ainsi que sur certaines aspects intéressants comme la détection d'obstacles.

2. Architecture logicielle

Comme indiqué sur la figure ci-dessous, nous avons opté pour quelques modifications, nous nous sommes permis d'effectuer quelques changements sur l'architecture initiale du projet :

- Nous avons rajouté un module obstacle afin de gérer certaines actions qu'un obstacle "sait" faire, à savoir renseigner sur des informations comme les coordonnées, la taille, disparaître quand nécessaire etc... De plus, pour gérer les collisions, il était pratique d'avoir un tableau d'obstacles à disposition.
- Nous avons créé un module "actor" dont héritent ball, player et obstacle afin d'exploiter le polymorphisme ainsi que de gérer des fonctionnalités communes à tous les acteurs de la simulation : la position, la taille, la présence (acteur dans la simulation ou disparu), la mise à disposition d'une surcharge de l'opérateur de sortie (<<) entre autres fonctionnalités.

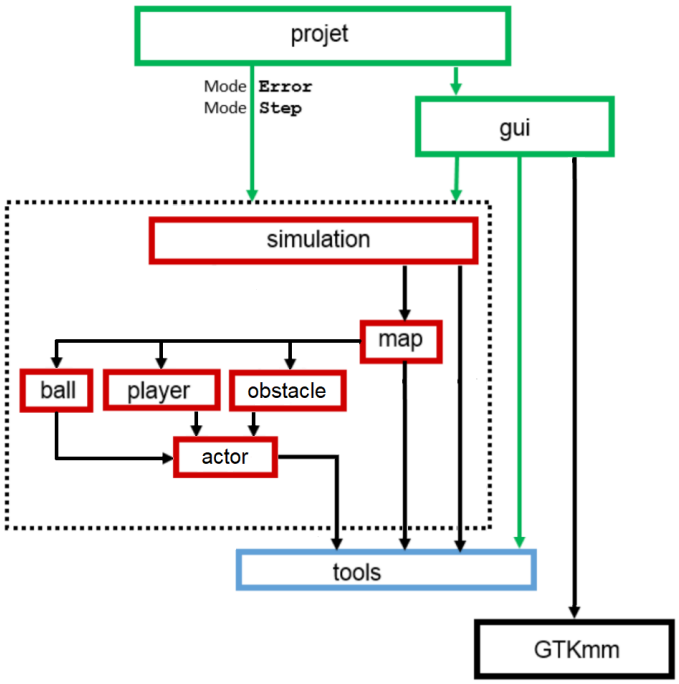


Illustration 1 : L'architecture modifiée du programme

Nous référant à l'architecture demandée pour ce projet, nous nous sommes conformés au modèle MVC : Model-View-Controller que nous présentons brièvement ci-dessous :

- Le module simulation (Controller) : responsable du dialogue entre l'implémentation du modèle, l'interface, ainsi que l'environnement externe du programme (lecture/écriture).
- Le module map (Model) : responsable de l'implémentation du modèle à proprement parler, il a donc le rôle de gérer les différentes instances d'acteurs, de coordonner les différentes étapes d'une mise-à-jour, de faire apparaître ou disparaître des acteurs et surtout de gérer les collisions entre ceux-ci. Ainsi, nous avons choisi de choix de mettre toutes les données de la simulation dans ce

module : 3 vector de pointeurs vers les 3 types d'acteurs respectifs, une matrice représentant la carte(avec les obstacles), ainsi que la matrice de Floyd.

- Le module gui (View) : se charge principalement de quérir les informations sur l'état du jeu du module simulation et les affiche à l'écran. De plus, il met à disposition de l'utilisateur un certain nombre de boutons afin de pouvoir interagir avec le programme (ouverture, sauvegarde etc...). Nous avons en outre pris la liberté de rajouter un bouton "reload" qui permet de réinitialiser la simulation. Pour finir, ce module implémente un "timer" qui, à chaque "tic", va entraîner l'appel du module simulation, qui à son tour délègue le travail de mise à jour du modèle au module map.

3. Le déroulement d'une mise à jour et complexité :

Dans le mode normal (avec affichage graphique), si le scénario est jouable (pas de collisions etc...), la matrice de floyd est calculée une première fois (via le module tools), avec l'état initial de la map.

Le schéma d'appel d'une mise à jour est le suivant :

1 Time out du Timer dans le module gui. Ce dernier fait appel au module simulation.
2 Le module simulation fait appel au module map afin de mettre à jour les éléments du jeu.
3 Le module map exécute les fonctions suivantes (en faisant appel aux fonctions des acteurs impliqués) : [Les complexités sont indiquées en bleu, en notation big-O]
  3.1 updatePlayers() : appel à la fonction update de chaque joueur : O(nbPlayer)
    3.1.1 update() (dans player) : Mise à jour d'un joueur en appelant les fonctions de player :
      3.1.1.1 chooseTarget() : O(nbPlayer)
      3.1.1.2 computePath() >...> canGoStraight() : O(nbCell)[1]
      3.1.1.3 updateCoolDown() : O(1)
      3.1.1.4 shoot() >...>map est appelée pour créer une balle : O(1)
  3.2 updateBalls() : pour chaque balle, les fonction move de ball est appelée, puis sont appelées les fonctions de détection de collision de map : O(nbBall)
    3.2.1 move() : O(1)
    3.2.2 détection de collision avec un obstacle : O(nbObstacle)
      3.2.2.1 Si collision, recalcul de la matrice de floyd :
        3.2.2.1.1 Initialisation des distances : O(nbCell^4)
        3.2.2.1.2 Exécution de l'algorithme de Floyd : O(nbCell^6)
    3.2.3 détection de collision avec un joueur : O(nbPlayer)
    3.2.4 mise à jour des points de vie des joueurs : O(1)
  3.3 clearDeads() : purge effective des acteurs signalés comme morts. O(max{nbPlayer, nbBall, nbObstacle})

Ainsi, selon les complexités que nous avons développées ci-dessus, nous pouvons déduire les complexité suivantes pour une mise à jour (sans calcul de la matrice de Floyd) :

O(updatePlayers()) = O(nbPlayer) * [O(nbPlayer) + O(nbCell) + O(1) + O(1)]
= O(nbPlayer) * O(max{nbPlayer, nbCell}) = O(max{nbPlayer, nbCell}^{2})
O(updateBalls()) = O(nbBall) * [O(1) + O(nbObstacle) + O(nbPlayer)+O(1)]
= O(nbBall) * O(max{nbObstacle, nbPlayer}) = O(max{nbBall, nbObstacle, nbPlayer}^{2})
O(clearDeads()) = O(max{nbBall, nbObstacle, nbPlayer})

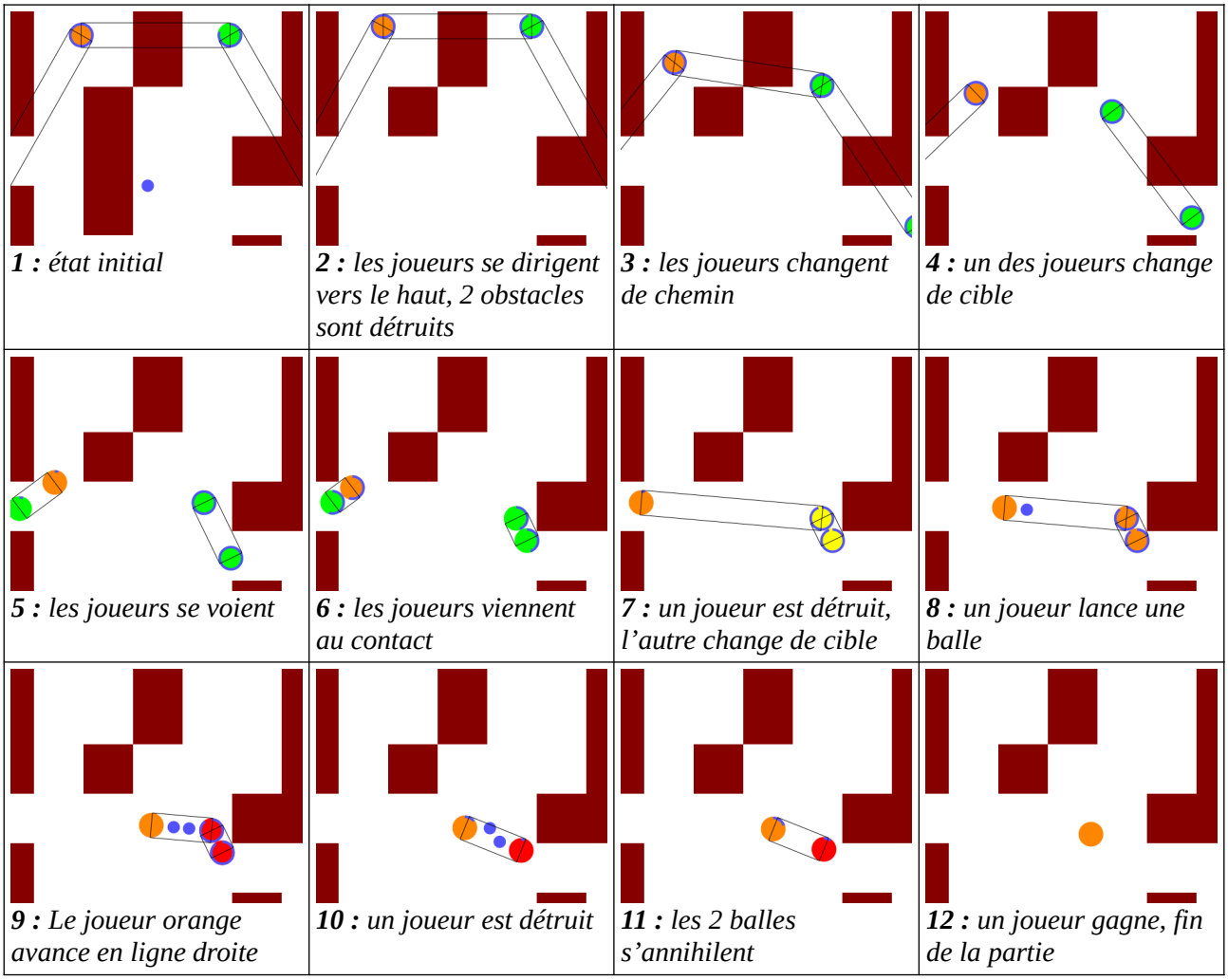
Donc, le coût calcul d'une mise à jour est donné par :

O(max{nbPlayer, nbCell}^{2}) + O(max{nbBall, nbObstacle, nbPlayer}^{2}) + O(max{nbBall, nbObstacle, nbPlayer}) =
O(max{nbBall, nbObstacle, nbPlayer}^{2})

1 L'algorithme exploite un itérateur qui se déplace du joueur vers sa cible en détectant s'il y a des obstacles sur le trajet. Il s'exécute ainsi en C1*sqrt(2)*nbCell = C2*nbCell. Ce qui implique O(nbCell).

4. Algorithme de déplacement des joueurs

Les figures suivantes illustrent l'évolution complète d'une simulation. Les rectangles noirs représente l'espace un joueur et sa cible. Si ce rectangle n'intercepte aucun obstacle, les joueurs peuvent se déplacer en ligne droite :



1 : *état initial*

2 : *les joueurs se dirigent vers le haut, 2 obstacles sont détruits*

3 : *les joueurs changent de chemin*

4 : *un des joueurs change de cible*

5 : *les joueurs se voient*

6 : *les joueurs viennent au contact*

7 : *un joueur est détruit, l'autre change de cible*

8 : *un joueur lance une balle*

9 : *Le joueur orange avance en ligne droite*

10 : *un joueur est détruit*

11 : *les 2 balles s'annihilent*

12 : *un joueur gagne, fin de la partie*

5. Méthodologie

Afin de réaliser ce projet, nous avons partagé les tâches de la manière suivante :

- N : s'est occupé de gérer le projet et son architecture, intégrer les différents fonctions dans le programme, communication entre modules, développement de diverses fonctions, notamment tout ce qui est collision
- J : s'est occupé de divers modules et fonctions, notamment celles de dialogue avec l'utilisateur : lecture, écriture, gui etc... ainsi que l'algorithme de Floyd.

Etant donné que le projet est composé d'un certain nombre non négligeable de modules, nous avons choisi d'utiliser Git, afin de gérer les différentes versions du code et de mieux coordonnées les tâches de chacun.

Pour ce qui est de l'intégrité des éléments de l'architecture, chacun a testé et fait en sorte que les fonctions/modules qu'il développait soient opérationnels indépendamment des autres modules, si possible. A quelques reprises, des bugs sont apparus suite à la mise en commun des modules ou après des

changements apportés à l'interface de ceux-ci et qui entraînaient des problèmes en cascade. Cela dit, ce fut facile à gérer car le projet reste de très petite taille.

Une attention particulière a été portée au début de chaque phase pour que chaque module définisse les attributs (+ leurs getters/setters) et fonctions importantes (via des stubs) de la façon la plus exhaustive possible afin que chaque membre du groupe ait connaissance des différentes fonctions exportée par le module développé par l'autre. Ainsi, après avoir atteint un commun accord sur la déclaration d'une interface donnée, il ne reste plus qu'à proprement implémenter les différentes fonctions.

Travaillant avec la librairie gtkmm pour la première fois, et comme nous ne connaissions pas l'API de celle-ci, la partie interface graphique du programme était plus compliquée à prévoir et à développer au début.

Le plus grand problème que nous avons rencontré est l'imposition de 3 systèmes coordonnées différents au sein d'un petit programme :

1. Coordonnées gtkmm
2. Coordonnées projet pour joueurs et balles
3. Coordonnées discrètes pour obstacles.

Et donc, à maintes reprises, et à cause de tous les changements de coordonnées effectués, notamment pour gérer les collisions joueurs/obstacles pendant le jeu, nous avons rencontrés beaucoup de bugs (à cause de l'inversion des axes x et y, line et column, coordonnées discrètes et non discrètes, axes inversés etc...), ce qui rendait le débugage de certaines fonctionnalités un peu pénible, ne sachant pas si le problème venait de l'algorithme ou bien du système d'axes. Cependant, nous avons fini par développer un certain nombre de fonctions de changement de coordonnées fiable et parfaitement fonctionnel, ce à quoi nous aurions dû consacrer plus de temps en phase de préparation.

6. Conclusion

Noureddine : ce projet m'a permis de mettre en pratique les différentes notions vues en cours en langage c++, ainsi que de découvrir la librairie gtkmm. Il m'a également permis de mieux mettre en œvre le principe d'encapsulation.

Jennyfer :

En somme, le développement de ce projet, en plus d'avoir été un plaisir, a été un grand enregissement pour chacun de nous.

Petite conclusion + auto-évaluation.