Borkhoche Johnny: 296169 El Khoury Evan: 302659

Rapport du projet de simulation: Dodgeball

Structuration des données:
Tout module ne gère que les entités dont il est responsable : gui.cc gère gtkmm, simulation.cc gère les étapes du jeu et les relations inter-modulaires, player.cc gère les joueurs, map.cc gère les obstacles et ball.cc gère les balles).
Cependant on emploi des accesseurs dans la quasi-majorité de nos modules pour pouvoir accéder à quelques éléments nécessaires au module simulation qui seraient inaccessibles par ailleurs.

Gui.cc s'occupe des interractions utilisateur-programme par les boutons de l'interface graphique, et surtout de la notion de temps.
C'est quand même le module simulation qui décide de ce qui se passe durant l'intervalle du timeout.

Les taches lors d'une mise à jour sont réparties de la manière suivante:

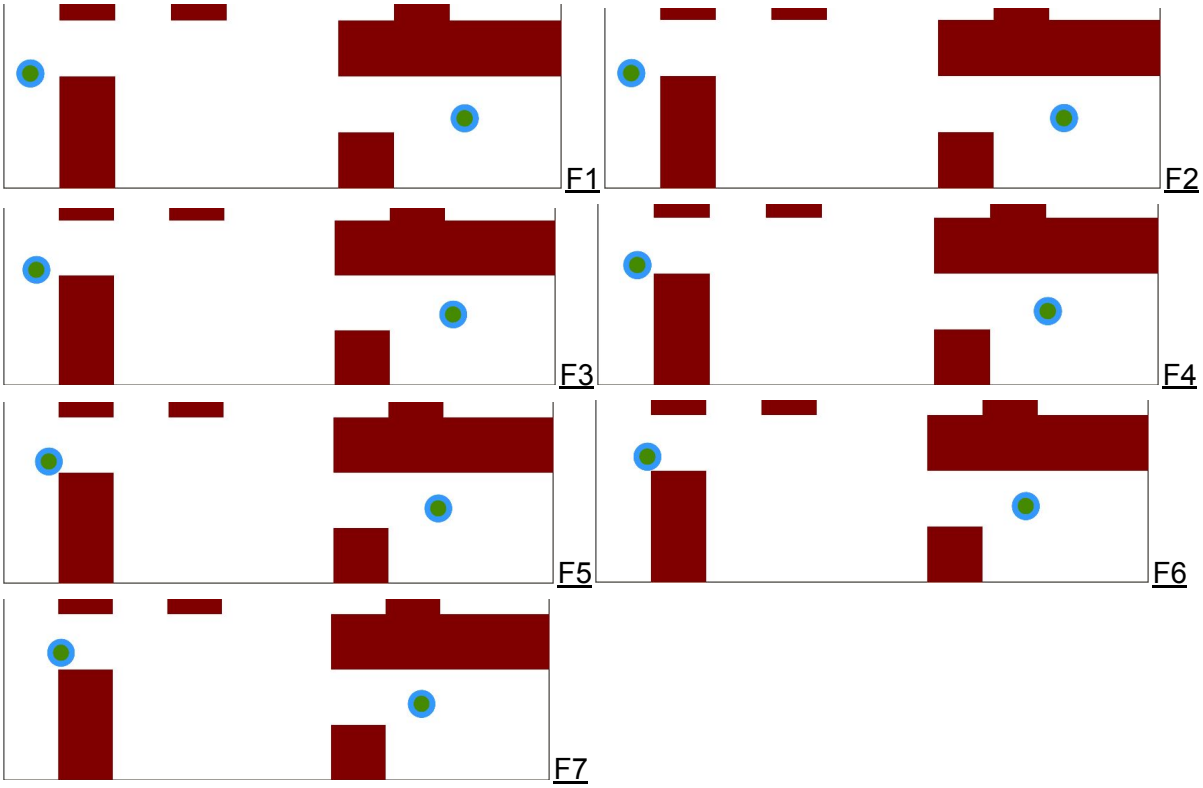
-Simulation.cc détermine les plus petites distances entre les joueurs en laissant le module player.cc lui renvoyer les données dont il a besoin puis les garde en mémoire jusqu'à la fin de la portée de la fonction simulation.
-Ce même module détermine ensuite si la voie est libre. Ceci requiert l'aide du module map.cc qui contient les coordonnées des obstacles. il reçoit donc une copie du vecteur statique qui n'est inaccessible que dans map.cc.
-C'est maintenant à la matrice des distances d'être créée. Nous utilisons l'algorithme de Dijkstra donc notre matrice ne comporte que des distances initiales elle n'est pas retravaillée. Si c'est la première fois que le fichier courant est ouvert, cette matrice est créée. Sinon c'est la destruction d'un obstacle qui sollicite une modification mineure.
-L'algorithme peut alors être mis en oeuvre pour chaque cible, et grâce à la comparaison du plus court chemin d'une cible vers les voisins du joueur, le spot vers lequel le joueur se déplace est décidé.
-Le module simulation demande finalement a player.cc d'incrémenter l'attribut count des joueurs et au module balles de créer des balles quand il le faut, pour ensuite vérifier les collisions entre les entités par le module simulation lui-même(sauf pour la collision balle-balle faite dans ball.cc).
-Si la simulation est términée ou interminable, c'est a gui.cc de changer les labels de l'interface graphique.
chaque module s'occupe de ses propres entités pour réduire au maximum les dépendances entre les modules et faciliter la réutilisation.

Le coût calcul et le coût mémoire lors d'une mise à jour:

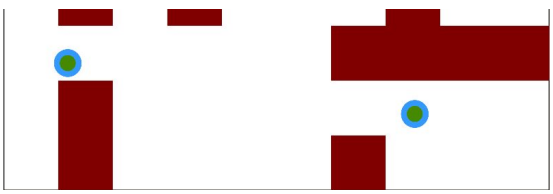
-la détermination des cible se fait en (nbPlayer*nbPlayer)
-la matrice des distances est initialisée en (nbCell*nbCell*8) puisque nous n'avons besoin de connaître que la distance jusqu'aux voisins dans cette matrice.

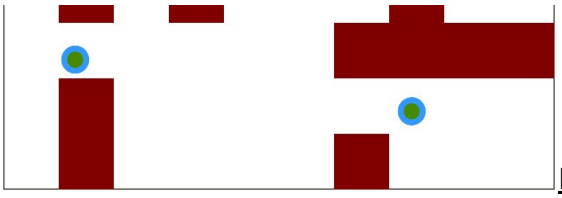
Borkhoche Johnny: 296169 El Khoury Evan: 302659

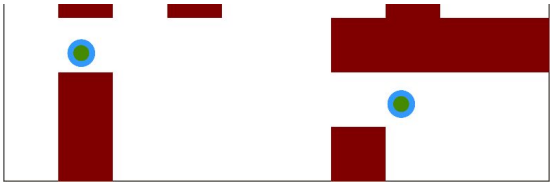
-La matrice est ensuite modifiée par l'ajout des obstacles en (nbObstacles*16) puisqu'il faut remplir les cases des obstacles par nbCell_au_carre puis modifier les distances des voisins qui ont un obstacle devant eux.
-L'algorithme de Dijkstra se fait en (nbPlayers*nbCell*nbCell) si tout les joueurs n'ont pas de cible direct.
-Pour finir les collisions se font respectivement en (nbObstacles*nbBalles) et (nbObstacles*nbPlayers) la dernière permettant au joueur d'éviter un obstacle dans sa position future.
-Le coût mémoire est toujours maximale dans notre cas puisque nous usons d'une matrice complète nbCell*nbCell mais cela permet de réduire grandement le coût calcul. Nous avons donc opter pour ce choix.

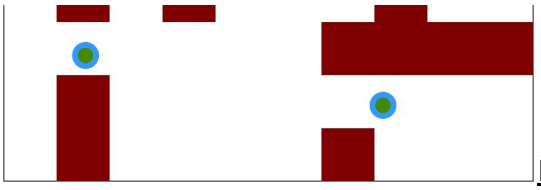


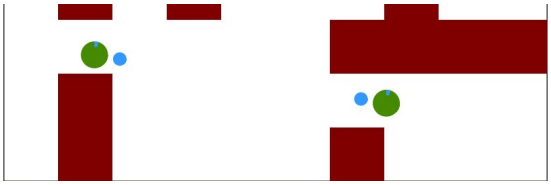
Borkhoche Johnny: 296169 El Khoury Evan: 302659


F8


F9


F10


F11


F12

Methodologie de travail:

Nous avons initialement essayé de travailler chacun sur ses modules en mettant en commun petit à petit notre travail. Cela dit la synergie n'était pas présente et nous n'avons pa pu être synchronisés. Pour y remédier nous avons écrit tous les modules ensemble en utilisant la technique rubber-duck pour nous éviter des problèmes simple qui passerait inaperçu sans que nous ayons une personne prête à écouter une explication explicite et détaillée du code. Même si cela a pris un peu de temps au début, quand nous nous sommes adaptés l'écriture du code se faisait beaucoup plus efficacement et avec étonnamment peu d'erreurs de compilation. Pour tester nos modules nous avons établi des fichiers tests avec nos propres données adéquate à l'erreur que nous voulions cerner. Nous avions un fichier .cc supplémentaire qui a servi d'assurance à la compréhension des fonctions du l'include<cmath> de c++ et nous avons surtout utilisé la méthode "cout" pour débugger notre programme. Le bug le plus fréquent a été cerner lors du rendu final et comprend la détermination de la voie libre des joueurs. Nous avons mal ménagé notre temps lors de l'écriture de cette partie ce qui nous a induit en erreur au tout début car nous pensions que l'erreur provenait de l'algorithme du plus court chemin. Nous n'avons donc pas eu la chance de revenir calmement sur cette détection des obstacles. Nous l'avons résolu en nous convaincant que tout le reste du programme fonctionnait parfaitement et avons par la suite attaqué le problème mathématiquement sur papier-crayon.

Nos points forts ont été l'écriture de code sans fautes syntaxique et le courage d'effacer des parties entières de code lorsque nous trouvions une solution plus efficace.
Le fait que nous étions sur la même page nous a permis d'avancer sans avoir des querelles inutiles. Cela nous a permis d'optimiser l'algorithme du plus court chemin pour notre cas spécifique de simulation d'un jeu.