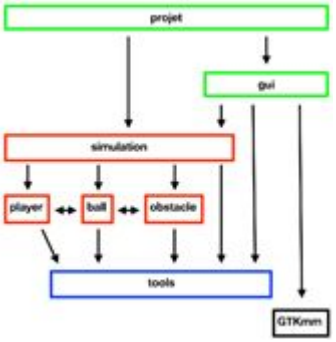
Rapport Projet Dodgball 270354 & 302076

1. Architecture logicielle et description de l'implémentation :

a. Architecture logiciel

Concernant l'architecture du logiciel, nous avons introduit deux modifications :



1. Nous avons remplacé le module *map* par un module *obstacle* qui représente un unique obstacle.
2. Nous avons inclus une dépendance entre les modules *player*, *ball* er obstacle afin de créer les fonctions de collision p.e. : *bool Ball::collide_with(Player p).* Nous avons créé ces fonctions pour tous les « couples » d'objet de style *Ball-Ball*, *Ball-Obstacle*, etc. Donc il y a aussi une dépendance réciproque entre *player* et *obstacle* qui n'est pas dessinée sur le schéma.

a. Structuration des données

Pour stocker nos données, nous avons créé trois classes de donnée :
- Player qui contient la position centre du joueur dans les deux systèmes de coordonnées (coordonnées obstacle (cells) et gtkmm (SIDE)) sous forme de point, ainsi qu'un int nbT (la vie du joueur), un double count (la barre de chargement).
- Ball qui contient la position centre du joueur dans les deux systèmes de coordonnées (coordonnées obstacle (cells) et gtkmm (SIDE)) sous forme de point, ainsi qu'un double angle qui est la direction de la balle
- Obstacle qui contient deux int, la ligne et la colonne de l'obstacle ainsi que les coordonnées en « SIDE » du coin supérieur de l'obstacle.

Ensuite les données sont stockées au moyen de trois *vector* (un pour les Player, un pour les Ball et un pour les Obstacle) dans le module *simulation*. Ensuite, pour transmettre les données à « dessiner » au module gui, le module *simulation* transmet deux *vectors* de forme géométrique (un de *Rond* et un de *Rectangle*) qui contiennent les *Rond* (*Player* et *Ball*) et les *Rectangle* (*Obstacle*) à afficher.

c. Calcul d'une mise à jour

Une mise à jour du programme se déroule comme ci-dessous:

1. On check si le jeu est terminé, sinon :
2. Déplacement des joueurs
3. On tire avec chaque joueur
4. Déplacement des balls
5. Contrôle de collisions entre ball / obstacles / players
6. On "kill" les joueurs, balles et obstacles appropriés
7. S'il reste un seul joueur, game over
8. Et si un obstacle a été détruit, on recalcule la matrice de floyd

d. Calcul cout d'un pas de mise à jour

i. Mémoire

La matrice de Floyd (ici Floyd_Mat) est une matrice en 4D dans notre programme. On la stocke dans la class *Simulation*. C'est simplement un tableau de pointeurs sur des *double*. C'est donc un tableau de $NBCELL^4$ *double*. Si on prend 8 bytes par double, c'est donc 80000 bytes (soit 78kB) en mémoire pour cette matrice. On pourrait l'optimiser, sachant qu'elle est symétrique selon un axe, car notre jeu ne contient pas de case de type "sens

unique”. Cela implique que $FM[x][y][z][t] = FM[z][t][x][y]$. Car la $||(x,\ y) \rightarrow (z,\ t)|| = ||(z,\ t) \rightarrow (x,\ y)||$. On pourrait donc optimiser la place en mémoire d’un facteur *au minimum* 2, ainsi que le temps de calcul.

En faisant un simple *sizeof* sur les entités de type *player, ball, obstacles*, les valeurs de retours sont les suivantes : 248b pour *Simulation*, 56b pour *Player*, 48b pour *ball*, et 32b pour *obstacle*.

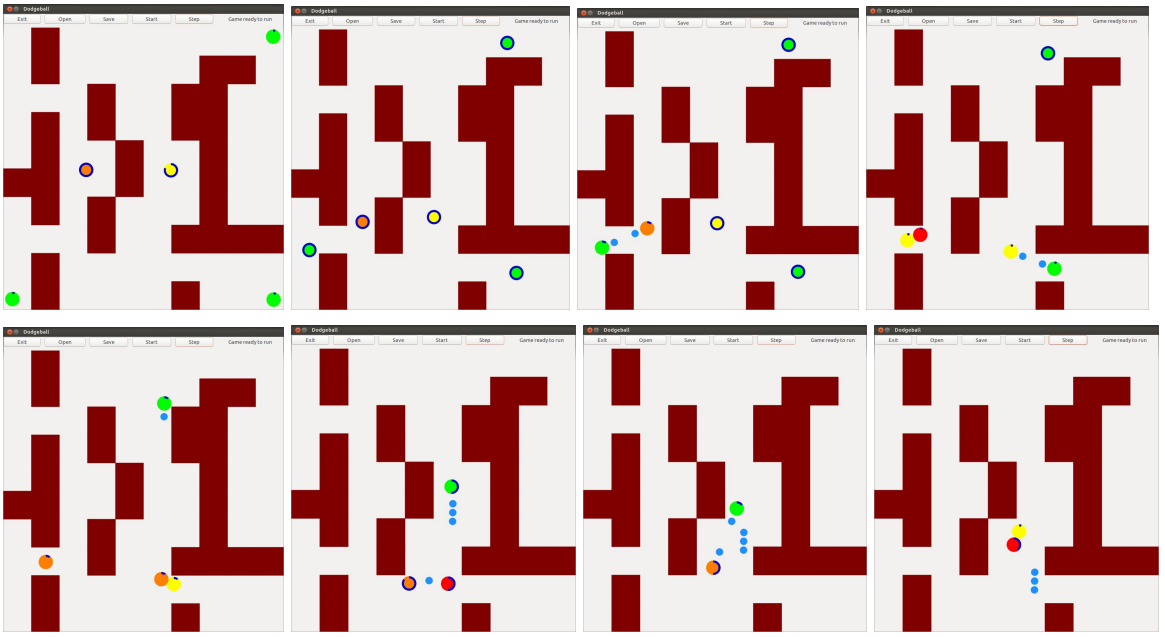
On peut estimer que lors d’une simulation classique (10 joueurs, 30 obstacles et 10 balls, cela fait environ 2200 bytes, sans compter la matrice de floyd, bien sur.

ii. Calcul

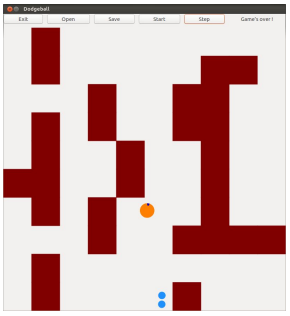
Le calcul de cette matrice est *coûteux*, surtout si **NBCELL** est grand. En effet, l’initialisation de la matrice est en $O(NB_{CELL}{}^{4})$, est son calcul est en $O(NB_{CELL}{}^{6})$. Heureusement, ces calculs ne se font pas à chaque itération, mais une fois au début, et à chaque modification de map (obstacles détruits, etc …).

e. Illustration de rencontre entre deux joueurs avec obstacles

i. Photos



ii. Explications



On voit sur la figure 1 l’initialisation du jeu. On voit donc que les joueurs se déplacent en diagonale vers l’espace entre les obstacles. Lorsque les joueurs arrivent dans ce qu’on a appelé *“l’espace visible”* (qui est simplement l’espace quand les joueurs se voient directement), ils commencent à se tirer dessus. On remarque que dès la figure 6, le joueur qui était en haut à droite, à tiré 3 balls en direction des autres joueurs.

2. Méthodologie et conclusion :

a. Répartition des tâches

Rendu	Arnaud Sansonnens	Loïc Gremaud
1	*tools, player, ball, obstacle*	*simulation, projet*
2	*tools, player, ball, obstacle, gui* (affichage forme géométrique)	*simulation, projet, gui* (affichage de l'état du jeu avec les fonctions faîtes par Arnaud)
3	*tools, simulation, gui, projet* (création du timer, des fonctions *run(), move_ball, collide(), shot(),* gestion du mot clé *step*)	*simulation, player* (mouvement des joueurs, algorithme plus court chemin, correction bug)

b. Organisation travail

Pour chaque rendu, on a commencé par les modules de bas niveau réalisé par Arnaud avant de passer aux modules plus « haut » niveau réalisés par Loïc. A chaque rendu, les modules de bas niveau ont été testés avec des « *main* » différents, plus simple afin de pouvoir facilement tester les modules. Ceux de plus haut niveau ont été testés avec des fichiers de test « réalistes » du jeu.
S'il fallait refaire le projet, on serait plus systématique sur notre travail, on aurait fait un fichier tools beaucoup plus conséquent dès le début, afin d'être plus consistant sur la suite de notre programme. Par exemple, on aurait dû créer des formes géométriques dès le rendu 1 pour caractériser nos joueurs, nos balles, etc. Ou alors le concept de segment et de droite qui a été ajouté au rendu 3.

c. Bugs fréquents

On a souvent eu des bugs liés au marge dans le programme, cela vient principalement du fait que nous avions commencés par stocker les valeur en "cell". On entends par là que les valeurs des players n'étaient pas stocké en position type (x, y), mais dans une fraction de cell. Nous avons au final un mélange de type de stockage, qui n'est pas optimal quand on cherche à afficher les objets, calculer par dessus, etc …

d. Auto-évaluation

Concernant l'architecture du projet, le stockage des éléments de base(*player, ball, …*) est un point qu'on changerait. En effet, le fait de ne pas stocker **uniformément** les données (certains en fraction de cell, d'autres en (x, y)), n'aide pas pour faire du développement efficace, trop de temps a été passé pour le debogage, à notre avis.