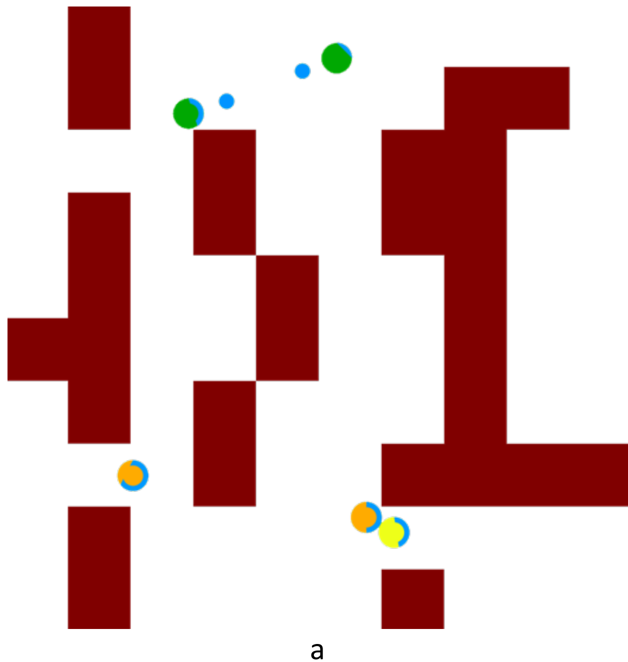
V 1.04

C++ PoP – Sections Electricité et Microtechnique

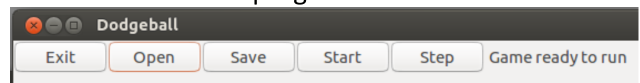
Printemps 2019 : *Obstructed Dodgeball* © R. Boulic & collaborators

Quel joueur restera le dernier en lice ?[1]



a

Figure 1 : (a) exemple d'un état de la simulation dans un système de coordonnées ***Monde*** montrant le site contaminé dans l'espace continu [-DIM_MAX, DIM_MAX] selon X et Y. Exemple d'entités manipulées : **player** (représentation simplifiée avec un disque plein dont le bord bleu indique l'écoulement d'une durée à respecter avant de pouvoir lancer une balle), **ball** (petits disques pleins bleus), **obstacle** (constitué de cellules carrées marron alignées sur une grille). (b)Interface graphique réalisée avec GTKmm montrant les boutons pour lire/écrire un fichier et gérer la simulation (Start/Step). L'espace à droite du bouton Step est destiné à afficher un message sur l'état du programme



b

1. Introduction

Ce projet est indépendant de celui du semestre dernier. Le lien reste néanmoins la mise en œuvre des grands principes (*abstraction, ré-utilisation*), les conventions de présentation du code et les connaissances accumulées jusqu'à maintenant dans ce cours. Le but du projet est surtout de se familiariser avec deux autres grands principes, celui de *séparation des fonctionalités* (*separation of concerns*) et celui *d'encapsulation* qui deviennent nécessaires pour structurer un projet important en *modules* indépendants.

Nous mettrons l'accent sur le lien entre *module* et *structure de données*, et sur la robustesse des modules aux erreurs. Par ailleurs *l'ordre de complexité* des algorithmes sera testé avec des fichiers tests plus exigeants que les autres.

Vous pouvez faire plus que ce qui est demandé dans la donnée mais n'obtiendrez aucun bonus. Dans tous les cas, vous êtes obligé de faire ce qui est demandé *selon les indications de la donnée et des documents des rendus*. En bref, vos éventuels ajouts personnels ne doivent pas interférer avec les présentes instructions.

Le projet étant réalisé par groupes de deux personnes, il comporte un oral final <u>individuel</u> noté pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de <u>l'ensemble du projet</u>. Une performance faible à l'oral peut conduire à un second oral approfondi et une possible baisse des notes des rendus.

La suite de la donnée utilise un certain nombre de constantes (en MAJUSCULE) disponibles dans le fichier **define.h**. Ce fichier est fourni en Annexe A. On utilisera toujours la *double précision* pour les données de position dans terrain de jeu et les calculs en virgule flottante (section 3).

[1] Ou, moins fréquemment, quels joueurs s'élimineront simultanément à la fin ?

2. Règles du jeu

Le but de ce projet est de simuler une partie de ballon prisonnier. Des joueurs, représentés par un disque coloré, sont sur un terrain de taille constante et se lancent des balles en mousse (petit disques bleus dans Fig1a) en se déplaçant dans un espace continu contenant d'éventuels obstacles (formes rectangulaires marron dans la Fig1a). Voici les règles du jeu :

1. tout joueur détermine ***son joueur cible*** comme étant ***celui qui est le plus proche de lui à vol d'oiseau***, c'est-à-dire selon la distance cartésienne qui relie leurs positions en ligne droite et ceci indépendamment des éventuels obstacles entre les joueurs (Fig2). Le choix de la cible est réévalué à chaque mise à jour du jeu.

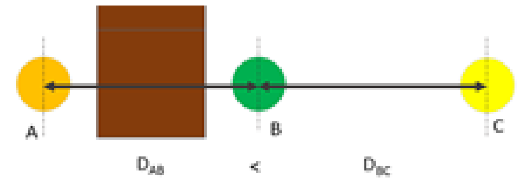


Fig 2 : la cible du joueur vert est la joueur orange (à gauche) car il est le plus proche à vol d'oiseau.

2. chaque joueur doit se déplacer vers son joueur cible ***selon le plus court chemin en prenant en compte les obstacle***s tout en restant dans le terrain de jeu (Fig3). Il ne doit y avoir aucune collision/superposition avec les éventuels obstacles présents sur le terrain de jeu. Le déplacement s'effectue avec une vitesse constante (section 2). Il n'y a pas de concept d'orientation du joueur. Le joueur peut se déplacer dans n'importe quelle direction sans avoir à tourner.

3. Quand un joueur voit sa cible sans intersection d'obstacle (Fig 3) ***il lance une balle vers sa cible avec une vitesse constante (section 2)*** ; plus précisément, la balle se déplace en ligne droite (Fig 4). Chaque joueur continue d'avancer vers sa cible même quand il lance une balle. Deux balles qui entrent en collision sont éliminées du jeu (Fig 4). Un décompte est lancé après chaque lancement de balle d'un joueur ; celui-ci doit ***attendre MAX_COUNT mises à jour avant de lancer une nouvelle balle***.

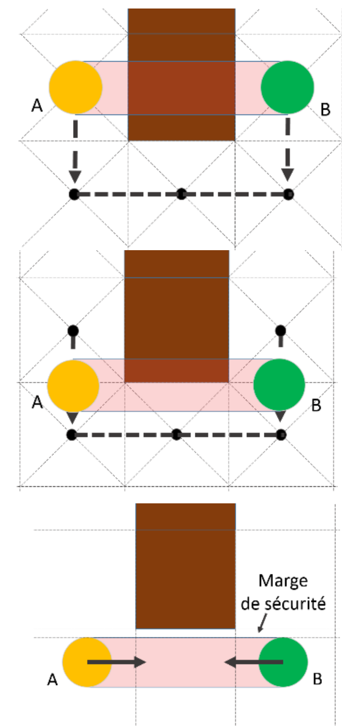


Fig 3 : tant que le couloir reliant les deux joueurs intersecte un obstacle, il faut suivre le plus court chemin passant par les centres des cellules de la discrétisation du plan.

4. Un joueur peut être touché par une balle ***MAX_TOUCH fois.*** Au-delà de ce nombre il est retiré du jeu.

5. Si deux joueurs arrivent au ***contact***, ils restent dans cette position. On continue d'appliquer les règles concernant le décompte temporel pour autoriser le lancement de balle et le nombre maximum de touches avant d'être retiré du jeu. On ne visualise pas la balle dans ce cas car la touche de l'autre joueur est instantanée.

6. Une ***balle perdue*** est une balle qui poursuit sa route ***car elle n'a pas intercepté sa cible initiale*** (la cible a bougé ou est éliminée). Elle peut toucher un joueur quelconque dont le nombre de touches augmente alors d'une unité. Si une balle perdue touche un obstacle, ***la cellule carrée de l'obstacle qui est en collision avec la balle perdue disparait***. Il peut y avoir plus d'une seule cellule carrée qui sont touchées en même temps. Une balle sortant du terrain est éliminée.

7. Un message « Game's over ! » s'affiche quand il reste zéro ou un joueur.

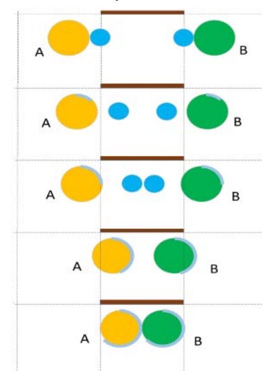


Fig 4 : lancer de 2 balles qui disparaissent au contact, puis contact des 2 joueurs (décompte visible en bordure)

2. Représentation du terrain et des éléments du jeu

La simulation est effectuée dans un système de coordonnée ***jeu*** d'origine (0,0) ; **X est l'axe horizontal, orienté positivement vers la droite**, **et Y est l'axe vertical, orienté positivement vers le haut**. Le terrain de jeu est limitée à [**-DIM_MAX, DIM_MAX**] selon X et Y. Cet espace est matérialisé par le dessin d'un carré de cette taille. Le centre des joueurs et des balles doit rester à l'intérieur du terrain de jeu.

Les joueurs et les balles se déplacent dans cet espace continu du terrain de jeu. A chaque mise à jour, s'ils peuvent bouger, leur position est déplacée d'une distance obtenue par le produit de DELTA_T par leur vitesse. La section 3 précise comment la mise à jour de tous les éléments doit être synchronisée.

L'information déterminant la taille de tous les éléments du jeu est la résolution **nbCell** de la grille qui sert de support pour construire les obstacles. Cette information est fournie au lancement du programme ; elle doit être au minimum de MIN_CELL.

Du point de vue de la définition des obstacles, le terrain de jeu est traité comme une grille de taille **nbCell** x **nbCell** cellules. Un obstacle est UNE cellule de cette grille : il est indiqué par les indices de ligne et de colonne de cette cellule entre **0** et **nbCell-1**. L'indice (0,0) correspond au coin contenant le point en haut à gauche du terrain de jeu (-DIM_MAX, DIM_MAX).

Pour simplifier la suite des définitions de taille des éléments on pose que la constante SIDE vaut (2*DIM_MAX). Par construction un obstacle sera donc un carré de coté **SIDE/nbCell**. Le fichier define.h contient les coefficients permettant de construire les autres éléments ainsi que les vitesses et la marge de sécurité (Fig 3) :

- Rayon d'un joueur = COEF_RAYON_JOUEUR * (SIDE/nbCell)
- Vitesse d'un joueur = COEF_VITESSE_JOUEUR * (SIDE/nbCell)
- Rayon d'une balle = COEF_RAYON_BALLE * (SIDE/nbCell)
- Vitesse d'une balle = COEF_VITESSE_BALLE * (SIDE/nbCell)
- ***Marge de sécurité pendant le jeu MJ*** = COEF_MARGE_JEU * (SIDE/nbCell)
- Marge de sécurité en lecture de fichier ***ML*** = (COEF_MARGE_JEU/2) * (SIDE/nbCell)

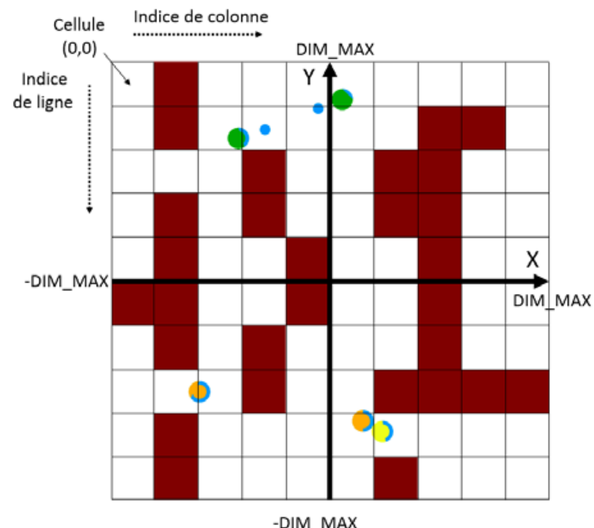


Fig 5 : visualisation de la grille associée au terrain de la Fig1. On voit l'impact du nombre de cellule nbCell par coté sur la taille relative des éléments du jeu. Noter les différentes conventions pour exprimer les positions des joueurs et des balles (espace continu selon X et Y) et pour indiquer les obstacles (indices de ligne et colonne)

Nous verrons en section 3.2 que cette approche par discrétisation du terrain de jeu permet d'effectuer efficacement des calculs de plus court chemin.

3 Actions à réaliser

Le but du programme est de simuler le déroulement du jeu conformément aux règles précédemment indiquées. L'exécution de la simulation peut être activée ou en pause. Dans le mode Pause, il doit être possible de réaliser les actions suivantes par l'intermédiaire de l'interface graphique (détails section 5):

- **Lecture** d'un fichier pour initialiser l'état de la simulation (section 4). Si nécessaire, avant de commencer la lecture elle-même il faut ré-initialiser les structures de données et libérer la mémoire. La simulation reste en mode Pause après la lecture du fichier.
- **Ecriture** d'un fichier décrivant l'état de la simulation à l'instant courant (section 4)
- **Lancement** de la simulation en continu ou seulement pour une seule unité de temps DELTA_T

3.1 Structure de la simulation

Lorsqu'elle est activée la simulation consiste en une boucle de mise à jour calculant l'évolution de l'état de tous ses constituants après un intervalle de temps DELTA_T depuis la mise à jour précédente. La boucle s'exécute tant qu'il y a plus d'un joueur.

Pour ce projet, la mise à jour est constituée *de plusieurs étapes* dont il faut respecter l'ordre :

- Analyse de la situation relative entre chaque joueur pour déterminer sa cible puis pour définir sa direction de déplacement et déplacer tous les joueurs qui le peuvent vers leur cible. Traitement d'éventuels contacts joueur-joueur (section 3.3)[2].
- (après le déplacement de tous les joueurs) lancer éventuel de balle
- (après tous les lancements de balle) déplacer toutes les balles.
- Détection de collision entre les balles et les joueurs, les obstacles, ou entre elles
- Purge effective des balles, joueurs, obstacles qui ont été éliminés pendant la mise à jour.

3.2 Détermination du chemin d'un joueur vers sa cible

Parmi les étapes indiquées dans la section précédente, celle de déterminer le chemin vers la cible d'un joueur est la plus délicate. Elle sera évaluée dans le rendu final. Voici les grandes lignes du raisonnement à suivre pour résoudre cette tâche.

3.2.1 Savoir quand on peut aller en ligne droite vers sa cible ou pas

Hypothèses de travail : contexte initial valide dans lequel il n'y a aucune superposition entre les éléments du jeu (joueur, balle, obstacle).

Supposons qu'un joueur B sait que sa cible est le joueur A parce que c'est le plus proche à vol d'oiseau (Fig 2). La première étape avant de déplacer B directement en ligne droite vers A est de s'assurer que la voie est libre, c'est-à-dire que le rectangle reliant les cercles A et B n'intersecte aucun obstacle. La Fig3 montre un cas où ça n'est pas le cas. Voici une reformulation de cette question qui est un peu plus simple à traiter géométriquement. Posons que les deux joueurs sont représentés par des cercles de rayon R et qu'une distance D sépare leurs centres. Comme illustré sur la Fig3 la voie entre A et B est représentée par un rectangle de taille Dx(2R). Supposons maintenant qu'on s'intéresse au test vis-à-vis d'un obstacle carré de coté L. On peut montrer que le test d'intersection entre le rectangle Dx(2R) et le carré LxL se ramène au test des deux cotés du rectangle du relient A à B vis-à-vis du carré LxL. Comme indiqué sur la Fig3 on s'assurera que la ***marge de sécurité MJ*** définie en section 2 existe entre le segment et le carré.

3.2.2 Que faire quand on ne peut PAS aller en ligne droite vers sa cible ?

Il faut trouver un chemin qui contourne les obstacles.
Cependant l'environnement créé avec les obstacles peut devenir complexe c'est pourquoi on simplifie le problème le discrétisant le terrain de jeu sous forme de la grille de taille nbCell x nbCell. Avec cette simplification, on ne s'intéresse qu'aux distances entre les centres des cellules de la grille. Chaque centre de cellule est un point de passage potentiel pour les joueurs. Appelons un tel point de passage un **Spot** ; il y en a **nbSpot** = **(nbCell x nbCell) – nbObstacle** en tout. Par construction, le joueur A est dans une cellule qui le représentera (son Spot) tandis que le joueur B sera dans une cellule différente (un autre Spot).
Il suffit alors d'exploiter l'algorithme de Floyd vu dans le cours ICC pour trouver le plus court trajet entre deux gares (slides 39-42). Pour cela il faut d'abord construire une matrice contenant nbSpot x nbSpot distances entre chacune des paires de Spots.

La matrice est initialisée avec les distances connues, c'est-à-dire la distance aux spots voisins :
- 1 selon l'axe vertical ou horizontal
- racine de 2 selon la diagonale.

Exceptions : on met **2** si cette diagonale est tangente à **un seul** obstacle. On met **nbCell²** qui représente l'infini pour ce problème si la diagonale est tangente à **deux** obstacles comme entre A et B ci-contre.
Pour les paires dont on ne connait pas la distance on met également **nbCell²**.

[2] La section 3.2 décrit comment les collisions joueur-obstacle sont évitées

Ensuite on fait tourner l'algorithme de Floyd et on obtient la matrice complète des plus courtes distances entre chaque paire de Spots. Se pose alors la question de comment utiliser cette information pour en déduire le chemin entre A et B. La première chose à se dire est qu'on n'a pas besoin de calculer tout le chemin entre A et B à partir de cette matrice ; ce dont on a besoin lors d'une mise à jour est seulement du *début* du chemin, c'est-à-dire le Spot voisin de celui où on se trouve et qui est le plus proche de la cible. Pour cela il suffit de comparer au maximum les distances des 8 voisins et de retenir celui qui offre la plus courte distance vers la cible. Exception : On ne peut pas utiliser la diagonale si elle est tangente à un obstacle (risque de collision).

Cette méthode ne donne pas systématiquement le plus court chemin mais présente trois avantages :

- Elle signale avec une distance supérieure à **nbCell**^{2} que le joueur est « prisonnier » dans un secteur du jeu qui n'a pas accès au secteur où se trouve sa cible. Dans ce cas particulier, le jeu doit se terminer et afficher le message « Cannot complete the game! » dans l'interface graphique (section 5)
- La matrice est en fait utilisable pour n'importe quelle paire de joueur.
- Une fois évaluée, la matrice reste valable jusqu'au moment où un obstacle est détruit par une balle perdue (règle 6), ce qui rajoute un Spot et requiert de relancer l'exécution de l'algorithme sur une matrice agrandie.

3.3 Traitement des collisions entre éléments

Les sections précédentes ont fait apparaître l'importance de détecter les collisions pour prendre un certain nombre de décisions, par exemple : valider le contact de 2 joueurs ou une touche sur un joueur, décider un calcul de chemin, détecter une configuration incorrecte à la lecture d'un fichier.

3.3.1 Lecture de fichier

La **marge de sécurité ML** à appliquer *en Lecture* est la moitié de celle utilisée pendant le jeu (son calcul est donné en section 2). Cette marge **ML** est à appliquer sur la détection de superposition sur les couples : joueur-joueur, joueur-balle, balle-balle, joueur-obstacle et balle-obstacle. Exemples :

- joueur-joueur : il y a superposition si **D < 2xR + ML** où D est la distance D séparant leurs centres et R est le rayon des joueurs.
- Joueur-obstacle (Fig6) : on transforme le problème initial (Fig6a) : le carré s'agrandit sur les côtés d'une longueur R tandis que les angles sont arrondis de rayon R (Fig 6b). On teste seulement l'appartenance du centre du joueur vis-à-vis de cette nouvelle forme.

La section 7.3 précise le comportement attendu du programme en cas d'une telle détection.

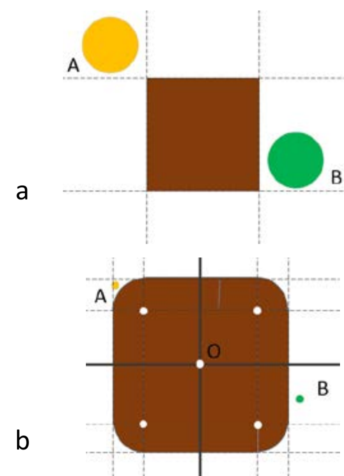


Fig 6 : détection joueur-obstacle

3.3.2 Pendant le jeu

Sachant que le jeu commence à partir d'une lecture de fichier, on pose que l'état initial du jeu ne contient pas de collision. Ce sont les déplacements des joueurs et des balles qui peuvent produire la superposition d'éléments du jeu. On procédera comme suit : la *possible* future position d'un élément mobile est calculée pour DELTA_T et testée vis-à-vis d'éventuelles collisions avec la marge MJ. En cas de collision, la future position n'est pas validée ; l'élément ne bouge pas mais son état peut changer du fait de la détection de collision (cf règles en section 2).

3.4 Tâche de bas-niveau

Plusieurs actions du projet ont besoin, dès le premier rendu, d'effectuer des calculs vectoriels et/ou géométriques pour prendre certaines décisions : détection de collision entre des cercles, ou entre un cercle et des carrés, etc…

Du point de vue de la programmation modulaire, il faut factoriser autant que possible les opérations communes (Principe de Ré-utilisation). Pour ce projet, un module **tools** de bas-niveau de gestion d'éléments géométriques dans l'espace 2D doit être crée pour gérer cet aspect .

Contrainte sémantique importante : par construction, étant de bas-niveau, le module **tool** ne doit connaitre que des entités génériques telles que points, vecteurs, segments de droite, cercles et carrés dans le plan. Ce module doit être indépendant des entités du jeu afin d'être ré-utilisable ultérieurement dans d'autres applications. Cela veut dire que le module **tool** n'a aucune idée de ce qu'est un *joueur*, une *balle* ou un *obstacle* : CES MOTS/CONCEPTS NE DOIVENT PAS APPARAITRE dans le module **tool**. C'est seulement au niveau des appels des fonctions dans les modules player, ball et map que les données fournies en argument seront explicitement des cercles représentant des joueur et des balles ou des carrés représentant des obstacles.

A l'inverse, voici une relaxation de contrainte pour le module **tool** : **L'usage de types concrets est autorisé pour ce module tool**. Cela veut dire que pour ce module vous pouvez définir des nouveaux types pour des *point*, *vecteur*, *segment de droite, cercle* et même *carré*, à l'aide de structures ***dont les modèles sont visibles dans l'interface tool.h***. Il y a certes une perte de contrôle en exposant les modèles de structure à l'extérieur du module mais il s'agit d'entités de bas-niveau dont on suppose qu'elles peuvent et doivent être validées et stabilisées rapidement.

4. Sauvegarde et lecture de fichiers tests : format du fichier

Votre programme doit être capable d'initialiser l'état du jeu à partir d'un fichier texte. Il doit aussi pouvoir mémoriser la configuration courante de la simulation dans un fichier texte également. Cela vous permettra de pouvoir reproduire une simulation donnée et de créer vos propres scénarios de tests avec un éditeur de texte.

Caractéristiques de fichiers tests :
Le nombre maximum de caractères par ligne est de MAX_LINE.

Les lignes vides commençant par **\n** ou **\r**, les commentaires commençant par **#** précédé éventuellement d'espaces, et les espaces avant ou après les données doivent être ignorés ; *il peut y en avoir un nombre différent d'un fichier à un autre*. Un commentaire peut aussi suivre les données.

Les fins de lignes peuvent contenir **\n** et/ou **\r** à cause du système d'exploitation sur lequel le fichier a été créé ; votre programme doit pouvoir traiter ces cas (cf série fichier).

Voici le format général	remarques
# Nom du scenario de test **#** **nbCell**	
nbPlayer **x0 y0 nbt0 count0** **x1 y1 nbt1 count1** …	Pour chaque joueur on indique sa position: x et y, son nombre de touches restant avant élimination et son compteur. Il y a un joueur par ligne.
nbObstacle **ligne0 colonne0** **ligne1 colonne1** …	Pour chaque obstacle on indique les indices entiers de ligne et de colonne. Il y a un obstacle par ligne.
nbBall **x0 y0 α0** **x1 y1 α1** …	Pour chaque balle, on indique sa position x et y suivi par l'angle en radian que fait la direction de son déplacement avec l'horizontale. Il y a une balle par ligne.

5. Description de l'interface utilisateur (GUI)

Nous utilisons une seule fenêtre graphique dont la partie supérieure est utilisée pour les boutons (Fig 1b) et la partie inférieure pour dessiner l'état courant du terrain de jeu (Fig 1a).

La fenêtre d'interface utilisateur doit contenir (Fig 1b):

- **Exit :** quitte le programme de jeu
- **Open** : remplace la simulation par le contenu du fichier dont le nom est fourni.
- **Save** : mémorise l'état actuel de la simulation dans le fichier dont le nom est fourni.
- Simulation :
  - **Start** : bouton pour commencer/stopper la simulation en continu
  - **Step** (lorsque la simulation est stoppée) : calcule seulement un pas de mise à jour
  - Un message est affiché à droite du bouton Step pour indiquer certains états du programme. Les 3 messages possibles sont les suivants :
    - Lorsque aucun fichier n'a été lu ou lorsque la lecture a échoué, afficher « No game to run »
    - Lorsqu'un fichier a été ouvert avec succès, afficher « Game ready to run »
    - Si un seul jouer est détecté comme « prisonnier », afficher « Cannot complete the game! »
    - S'il n'y a que zéro ou un joueur, afficher « Game's over ! »

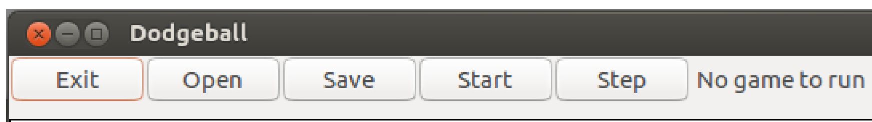


Fig 7 : Interface graphique (GUI)

6. Affichage dans la fenêtre graphique et synchronisation

A partir des rendus 2 et 3, l'exécution du programme ouvre une fenêtre GTKmm dont la partie supérieure est l'interface graphique utilisateur (Fig 7) et le reste est la partie destinée au dessin, appelée *canvas,* dont la taille est (2*DIM_MAX) en largeur et en hauteur. La taille de la fenêtre GTKmm ne change pas durant l'exécution du programme.
La Fig1 montre un exemple d'affichage du jeu. Il faut respecter les tailles des éléments. Une couleur distincte doit être utilisée selon le nombre de touches restant avant élimination. L'état du décompteur de chaque joueur est visualisé en bordure du disque qui le représente (Fig1a).

Dans ce projet l'affichage de l'évolution du jeu doit être synchronisé avec le temps du monde réel (*wall-clock time*) à l'aide de timer.

7. Architecture logicielle

7.1 Décomposition en sous-systèmes

L'architecture logicielle de la figure 8 décrit l'organisation minimum du projet en sous-systèmes avec leur responsabilité (Principe de Séparation des Fonctionnalités) :

- **Sous-système de Contrôle : son but est de gérer le dialogue avec l'utilisateur.** Si une action de l'utilisateur impose un changement de l'état de la simulation du jeu, ce sous-système doit appeler une fonction du **sous-système du Modèle** qui est le seul responsable de gérer la simulation (voir point suivant). Il est mis en œuvre avec deux modules :
  - Le module **projet** qui contient la fonction **main** : son rôle est modeste car il est responsable de traiter les éventuel paramètres fournis sur la ligne de commande au lancement du programme.
  - le module associé **gui** qui rassemble toutes les dépendances vis-à-vis de la bibliothèque GTKmm**.** Du point de vue du « contrôle » ce module gère le dialogue utilisateur à l'aide de l'interface graphique mise en œuvre avec GTKmm.

- **<u>Sous-système du Modèle</u>** <u>(rectangle en pointillé dans la figure 8 et 9)</u>: est responsable de gérer la simulation. Il est mis en œuvre avec plusieurs modules sur plusieurs niveaux d'abstractions selon les Principe d'Abstraction et de ré-utilisation (section 7.2).

- **<u>Sous-Système de Visualisation</u> :** son but est de traduire l'état courant de la simulation par un dessin. Ce sous-système est mis en œuvre avec le module **gui** qui rassemble les dépendances vis-à-vis de GTKmm vis-à-vis du **dessin**. Il demande les informations minimales sur l'état courant de la simulation au sous-système du modèle.

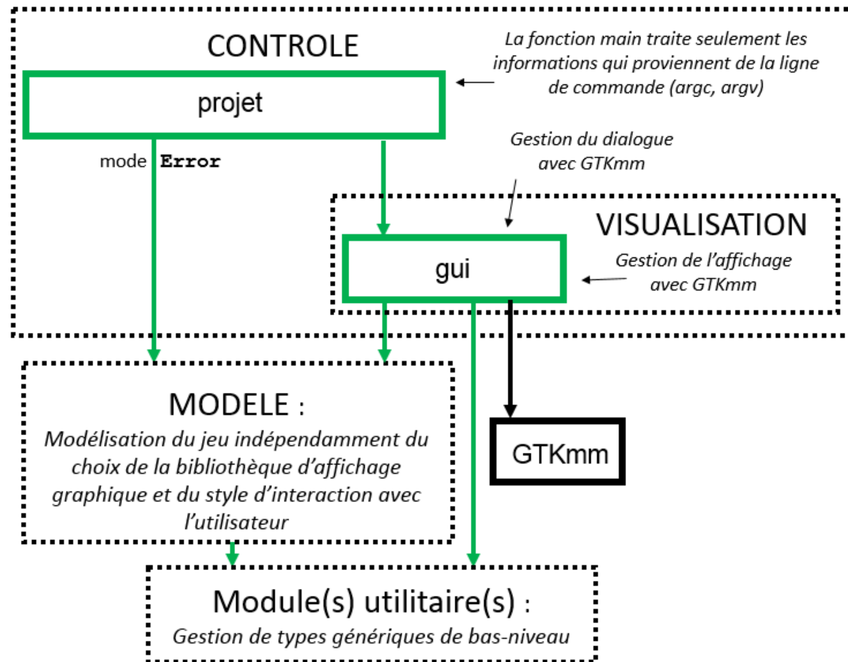


Fig 8 : Architecture logicielle minimale à respecter

7.2 Décomposition du sous-système Modèle en plusieurs modules

Le Modèle doit être organisé en plusieurs modules, pour maîtriser la complexité du problème et faciliter sa mise au point. La Figure 9 présente l'organisation minimale à adopter en termes d'organisation des modules :

- *Au plus haut niveau*, le module **simulation** gère l'évolution de la simulation et joue un rôle de coordinateur et d'intermédiaire pour les opérations de lecture et d'écriture de fichier et d'accès aux donnée minimales pour le dessin. Ce module doit garantir la cohérence globale du Modèle. C'est pourquoi, en vertu du principe d'abstraction le module **simulation** *est le seul module* dont on peut appeler des fonctions en dehors du Modèle (Fig 9)[3].

- *Niveau(x) intermédiaire(s)*: player, ball, map. Chaque module doit réaliser les actions utiles pour chaque entité (création, modification, destruction, lecture, écriture, accesseurs, calculs de haut-niveau, etc…)**.**

<u>Remarque</u> : Normalement c'est le module **simulation** qui est responsable de gérer la cohérence des relations entre ces trois modules. On peut introduire une dépendance entre les modules player, ball et map si celle-ci est clairement justifiée par un ensemble de fonctions mises à dispositions dans l'interface. Ces choix se décident dans la phase d'analyse du problème.

- <u>Module de bas niveau</u> **tool** (Principe de ré-utilisation): comme indiqué en section 3.4 il est demandé de créer un module **tool** de bas niveau mettant à disposition des **<u>types concrets</u>** pour réaliser des opérations dont vous avez besoin sur **des** éléments **2D** tels que des **points,** des **vecteurs**, des **segments** de droite, des **cercles** ou des **carrés**.

[3] si le sous-système de Contrôle veut modifier l'état de la simulation cela doit se faire par un appel d'une fonction de **simulation.h** comme le montre l'unique flèche de dépendance entre le module **main** et le module **simulation**.

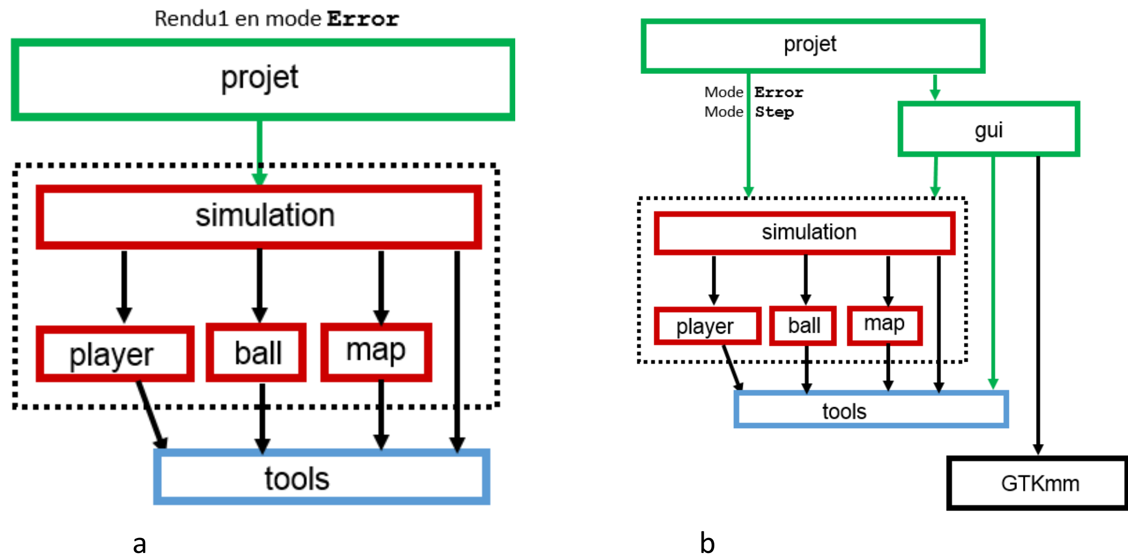


Figure 9 : (a) architecture minimale montrant les dépendances entre modules du sous-système *Modèle* pour la recherche d'erreur dans le fichier (mode **Error** détaillé en section 7.3); (b) dépendances supplémentaires pour la mise en œuvre de l'interface graphique (rendus 2 et 3)

7.3 Modes d'utilisation du programme et aperçu des trois rendus

7.3.1 Les quatre possibilités d'appel du programme

Votre exécutable doit s'appeler **projet**. On doit pouvoir lui transmettre des arguments optionnels au lancement, sur la ligne de commande. Quatre syntaxes d'appels sont autorisées. Deux syntaxes indiquent un *mode* d'exécution restreint du programme car celui-ci s'arrête immédiatement après avoir réalisé l'action liée à ce mode.

Pour le rendu1 *et les suivants* :

```
./projet Error nom_fichier.txt
```

Le mode **Error** sert à détecter s'il y a une erreur dans le fichier dont le nom suit le mot-clef « Error ». Le programme cherche à initialiser l'état du jeu mais il s'arrête dès la première erreur trouvée dans le fichier. Il s'arrête aussi après la lecture du fichier s'il n'y a aucune erreur ; dans ce mode il y a affichage d'un message indiquant le succès de la lecture. Ce mode n'a aucun affichage graphique.

A partir du rendu2 :

```
./projet nom_fichier.txt
```

ou

```
./projet
```

Avec ces deux syntaxes on lance le programme avec l'interface graphique. La première syntaxe ouvre immédiatement le fichier fourni sur la ligne de commande et affiche l'état initial du jeu. En cas d'erreur de lecture du fichier, les structures de données sont effacées puis le programme attend qu'on utilise l'interface graphique pour ouvrir un autre fichier. Avec la seconde syntaxe, le programme attend qu'on utilise l'interface pour indiquer un fichier à ouvrir.

En plus, pour le dernier rendu, le mode suivant est pratique pour la mise au point et les tests des algorithmes sans lancer l'interface graphique :

```
./projet Step nom_fichier_in.txt nom_fichier_out.txt
```

Le mode **Step** sert à 1) ouvrir le fichier dont le nom suit le mot-clef « Step» et initialiser l'état du jeu puis 2) à exécuter une seule mise à jour de l'état du jeu et 3) à sauvegarder l'état obtenu dans le fichier dont le nom suit en dernier argument. Le programme s'arrête dès la première erreur trouvée dans le fichier à lire. Il s'arrête aussi après l'écriture du fichier.

7.3.2 Résumé des aspects importants:

Les trois rendus auront environ le même poids dans le barème.

Rendu1 : test du format des fichiers dont vérification des collisions initiales (3.1.1),
Ce rendu travaille avec l'architecture de la Fig 9a. Le module projet contient la fonction main en charge d'analyser si la ligne de commande est bien conforme au mode **Error** qui est le seul exploité pour ce rendu. Si c'est le cas elle délègue l'action de lecture et d'initialisation du jeu au module simulation. Celui-ci, à son tour exploite les modules de plus bas niveaux pour initialiser les structures de données et vérifier les points suivantes :
- la position du centre de tous les joueurs et balles sont dans le terrain de jeu
- cohérence des indices d'obstacle avec nbCell et absence de duplication des obstacles
- absence de collision joueur-joueur, joueur-balle, balle-balle, joueur-obstacle, balle-obstacle

=> **nous fournissons un fichier en-tête avec les messages d'erreur que vous DEVEZ utiliser.**

Rendu2 : initialisation du GUI et dessin de l'état initial d'un jeu (section 5)
Ce rendu travaille avec l'architecture de la Fig 9b. Le module projet contient la fonction main en charge d'analyser si la ligne de commande est bien conforme aux trois possibilités d'appels indiquées plus haut pour le rendu2. Le mode Error reste inchangé. Pour les deux autres possibilités, la fonction main doit déléguer au module gui l'initialisation de l'interface graphique ; ce module peut ensuite traiter directement avec le module simulation pour lancer la lecture et effectuer l'affichage de l'état initial.

Ce rendu sera testé en effectuant plusieurs lectures et sauvegardes dans des fichiers puis relecture pour vérifier que l'affichage initial est bien correct. On s'assurera du fonctionnement des autres boutons à l'aide de stubs dans le module simulation.

Rendu3 : simulation, performances, rapport.
Ce rendu continue de travailler avec l'architecture de la Fig 9b. Le mode supplémentaire à mettre en œuvre avec le mot-clef Step est utile pour obtenir rapidement le résultat d'une mise à jour du jeu dans un fichier texte. Ce mode n'exploite pas le graphique. C'est un complément efficace à la visualisation à l'aide du bouton step.

Ce rendu sera testé en lançant des scénarios de test de complexité croissante (sur les variables nbCell, nbPlayer, nbBall) pour vérifier la bonne mise en œuvre des règles du jeu et estimer les performances de votre approche.

ANNEXE A : constantes définies dans define.h

L'unité de distance suggérée ci-dessous donne une taille de joueur à l'échelle humaine quand nbCell vaut 8.

#define MIN_CELL	3	
#define MAX_TOUCH	4	
#define MAX_COUNT	20	
#define DELTA_T	0.0625	seconde
#define DIM_MAX	400	cm
#define SIDE	(2.*DIM_MAX)	cm
#define COEF_RAYON_JOUEUR	0.25	pour taille en cm
#define COEF_VITESSE_JOUEUR	1.0	pour vitesse en cm/s
#define COEF_RAYON_BALLE	0.125	pour taille en cm
#define COEF_VITESSE_BALLE	1.25	pour vitesse en cm/s
#define COEF_MARGE_JEU	(1./256)	pour distance en cm
#define MAXLINE	80	