**Projet de programmation**

**Semestre 2**

**Première partie : l'arène**

*Equipe UbiHard*

**Bienvenu dans le Цолисёе !**

Le Цолисёе ou "colisée" est la plus grande arène du pays. C’est dans celle-ci que se déroulent toute l’année des combats sanglants et barbares. Barbares mais non moins dotés d’une règlementation stricte palliant toute tricherie et tout anti jeu. Venez donc faire valoir vos compétences en matière de stratégie et tenter de vous faire sacrer Champion.

Avant de pouvoir t’inscrire au tournoi tu dois prendre connaissance des règles très spécifiques qui font de notre arène la plus impitoyable de toutes. Tu trouveras toutes les informations nécessaires dans le codex du Цолисёе, regroupant toutes ses règles et ses subtilités. Il est aussi appelé par son matricule : 007.

En voici une copie tiré de nos archives :

* Le nombre minimum de participants est de 2 mais il n’existe aucun nombre maximum.
* Le nombre de participant ainsi que le nombre de tours sont choisis arbitrairement par les mécènes du tournoi.
* La case en haut à gauche de l’écran est la « case morte ».
* C’est un combat à mort : chaque participant n’a qu’une seule vie.
* Chaque tour se déroulera selon deux phases : une de déplacement et une d’action.
* Tous les participants jouent EN MÊME TEMPS à chaque phase de chaque tour.
* Durant la phase de déplacement, les participants devront choisir une des quatre directions correspondant aux quatre points cardinaux et se mouvrons alors d’une case dans cette direction.
* Durant la phase d’action, les participants devront choisir l’une des deux actions à leur disposition : « guard » ou « shoot » et en prendront le statut.
* Le statut « guard » confère à celui qui le porte une immunité totale contre les balles.
* Le statut « shoot » permet à celui qui le possède de choisir un participant à cibler.
* Le participant alors choisi n’est ciblé que si il est dans le rayon d’action du shooter.
* Si un participant est ciblé et non protégé, il est éliminé.
* Il n’est pas possible de recharger.
* Il est possible à un participant de choisir la même action deux tours consécutifs au plus.

\*Cas particuliers :

- Tout déplacement hors de la map ou sur la case morte entraine la mort du participant.

- Deux participants se déplaçant sur la même case entrainent une explosion menant à leur mort.

- Toute dérogation aux règles (notamment de récurrence d’action et de rayon d’action) peut associer à un joueur le temps d’un tour un statut « vide » le rendant vulnérable.

- En cas d’égalité, le joueur avec le plus de kills à son actif est sacré champion.

C’est bon tu as tout saisis ?

Alors profitons de ce que l’arène soit en travaux pour faire un tour dans les coulisses, inconnues du grand public…

**Le code**

La visite va se faire en deux parties car l’interface graphiques de l’arène et son corps ont étés codés séparément et n’ont pas encore été réunis.

La SDL

Le fichier main.c est le cœur du programme. C’est dans ce fichier que l’ensemble des fonctions, déclarations sont écrites. Il y a beaucoup de lignes mais le code est séparé et commenté en sections distinctes et vous devriez donc pouvoir vous y retrouver.

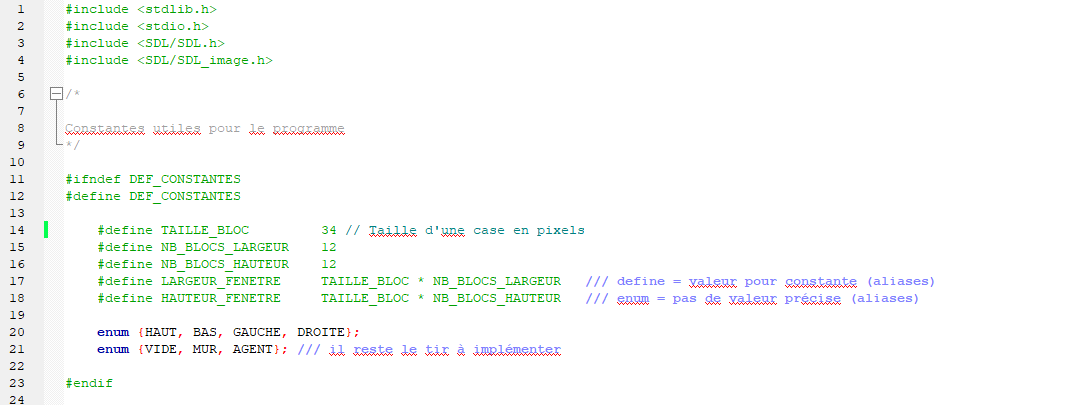


Fig 1.2 : Déclaration des constantes utiles

La première partie concerne la déclaration des constantes ce qui va simplifier la tâche dans la compréhension du code par la suite. De plus, les #include des lignes 3 et 4 montrent la nécessité de la présence des deux fichiers .dll dans le dossier de notre projet (ou à défaut, d’avoir installé la SDL …).

Pour les #define, à chaque fois que le nom qui y est associé apparait dans le code, la valeur correspondante l’est également. Les enum ont le même rôle mais n’ont pas d’affectation de valeur particulière.

A partir de la ligne 49 (cf fig 1.3) , on déclare une fonction chargerCarte qui porte bien son nom, rappelez vous nous parlions d’un fichier carte.lvl tout à l’heure. En effet, le rôle de cette fonction va être de lire ce fichier dans lequel on aura rempli des chiffres de 0 à 4 chacun de ces nombres étant associés à un objet (case\_vide, un agent, un mur ou une particule de tir). Le programme effectue donc une lecture de fichier et pour chaque chiffre associe une des situations évoquées ci-dessus.

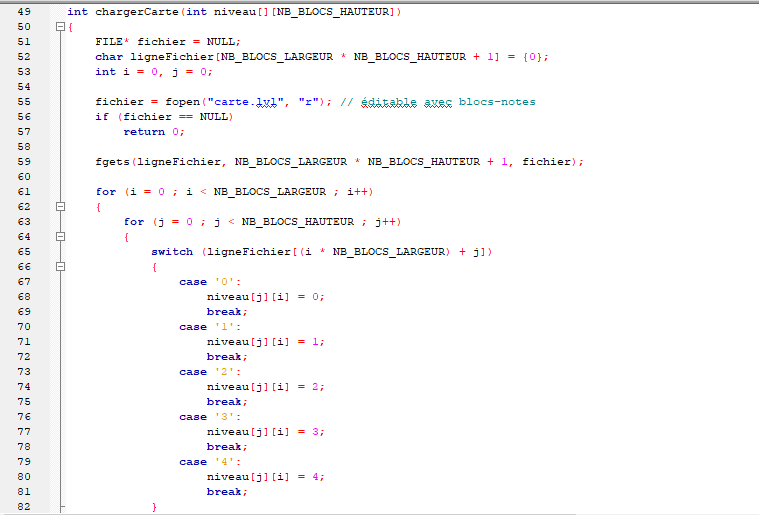


Fig 1.3 : Fonction chargerCarte

Le paramètre « r » dans la variable fichier indique que le programme va effectuer une simple lecture du fichier et c’est ce que l’on souhaite. Le fichier n’est pas pour l’instant modifié pour le programme.

A terme (et c’est d’ailleurs une amélioration sur laquelle nous travaillons en ce moment), c’est que le programme génère une grille type en fonction du nombre d’ia à jouer. Le fichier ne serait donc pas éditable. Par conséquent il va également falloir que nous adaptions le code au niveau des #define de la taille en pixels pour les cases et ce pour adapter l’affichage au nombre d’ia qui joueront. Actuellement, la carte a été affichée en 12 x 12 cases.

https://i.gyazo.com/1ce505fdd1edcd904b23f19fb1d1767c.pngLa fonction qui va suivre est la fonction la plus importante du programme (après le main qui apparait en fin du fichier main.c), c’est le cœur de notre jeu et c’est dans cette fonction que tout se produit. La fonction jouer est relativement longue et pour une question de clarté nous allons détailler les étapes les plus importantes au sein de cette fonction.

Fig 1.4 : Déclarations de deux fonctions essentielles

On déclare un peu plus tôt la fonction jouer qui prend en paramètre un pointeur sur écran. On déclare aussi la fonction deplacerJoueur dont on parlera après.

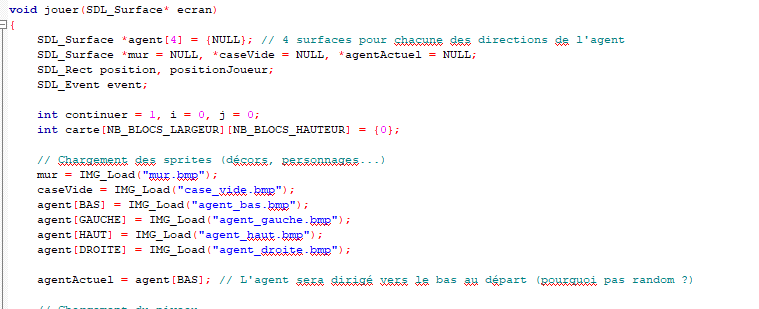


Fig 1.5 : Première partie fonction jouer

Dans cette partie, on va pouvoir voir quelle est l’utilité des #define du début de notre programme. On commence par déclarer un pointeur sur agent de type SDL\_Surface. On l’initialise à NULL par sécurité et on met un 4 car agent a 4 arguments : BAS, GAUCHE, HAUT et DROITE. Il ne suffit plus qu’à associer chacun d’eux à nos sprites présentes dans le dossier principal ! On remarque donc ici que les #define ont été utiles pour la clarté du code.

Pour le moment nous avons créé une variable agentActuel qui définit la position de l’agent, par défaut, à l’ouverture de la map. L’objectif est que la direction de l’agent soit aléatoire au moment de l’initialisation de la carte ou en tous cas orientée vers l’intérieur.

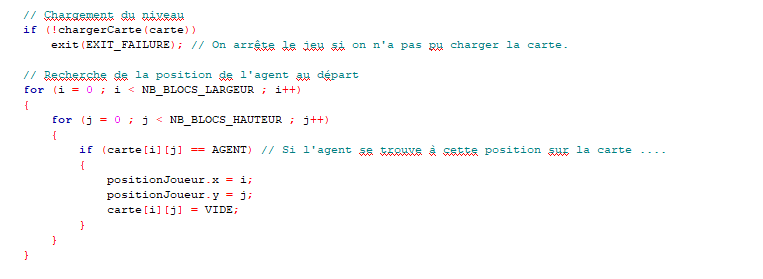


Fig 1.6 : Deuxième partie fonction jouer

Ensuite, si la fonction chargerCarte n’a pas fait son travail correctement (cf fig 1.6), on ferme le programme. La seconde étape consiste à parcourir le tableau pour déterminer la position de l’agent au départ avec une variable positionJoueur créée pour cela.

Par la suite, on retrouve également une partie importante du programme. La boucle while va contenir l’ensemble des évènements au clic ainsi que le placement des images à l’écran. Chacune des ses parties est exécutée (ou pas) en fonction de la valeur de continuer qui peu prendre la valeur 0 ou 1.

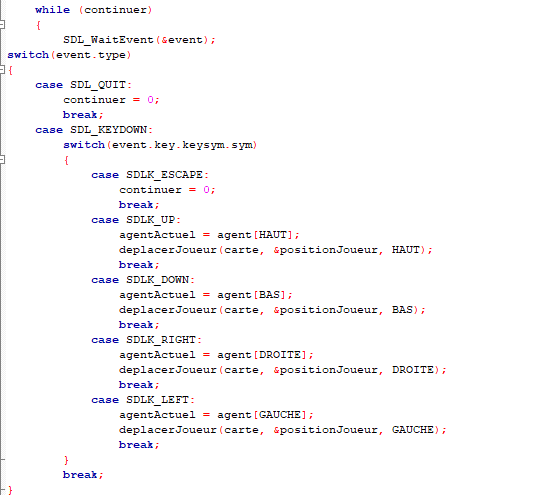


Fig 1.7 : Gestion d’évènements dans le while

La fonction SDL\_WaitEvent qui prend en paramètre l’adresse d’event va attendre qu’un élément se produise. Les évènements qui nous intéressent sont répertoriés dans la structure switch juste en dessous (cf fig 1.7). Cette partie du code est amenée à évoluer puisqu’il s’agit ici d’évènements au clic or ce seront des ias qui devront jouer et pas l’utilisateur.

Enfin la dernière partie la plus importante reste la fonction deplacerJoueur proprement dite

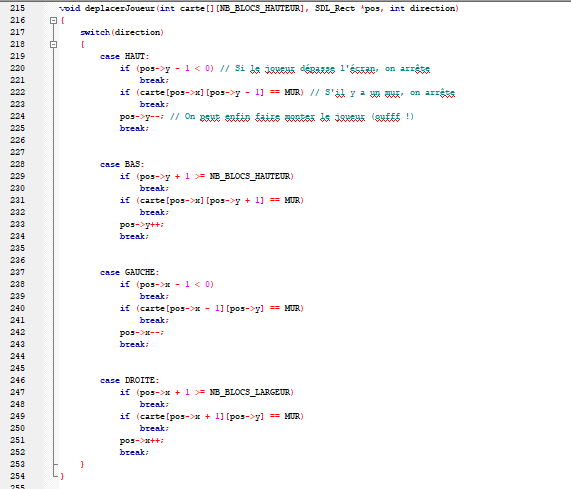


Fig 1.8 : Fonction deplacerJoueur

Dans une structure switch qui prend en paramètre direction, le programme va examiner et déterminer ce qu’il doit faire dans chaque cas c’est-à-dire pour chaque direction. On oublie de dire au programme de s’arrêter lorsqu’il rencontre un mur et on incrémente la valeur de la position sinon.

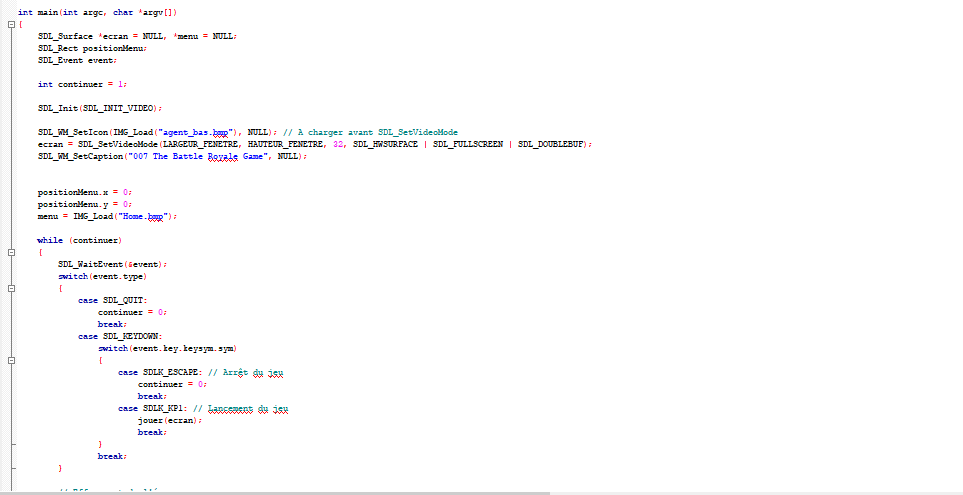


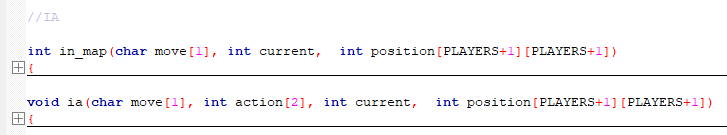
Fig 1.9 : Le main

Le code machine

Les différentes fonctions composant ce code sont divisées en familles. Je m’explique :

* Il y a d’abord les « IA », qui sont random et servent à tester la structure.

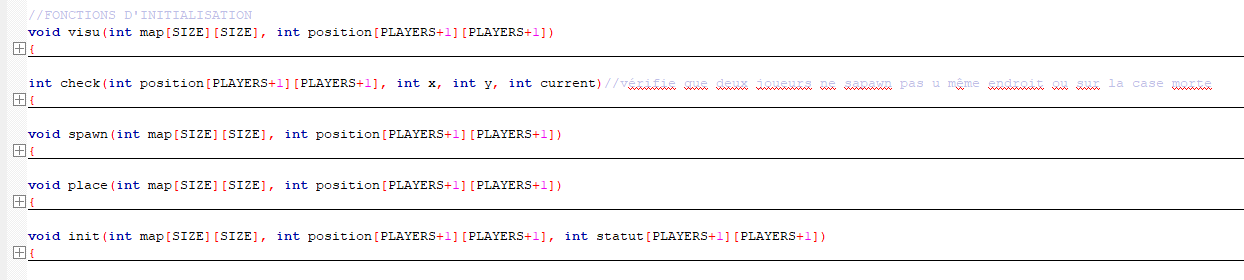
Cette famille contient deux fonctions : l’ia random qui sert de base à chaque participant ainsi que la fonction in\_map() qui permet d’éviter que l’aléatoire conduise l’ia au suicide par éjection hors de la map. ☺



* Puis viennent les « fonctions d’initialisation » qui déterminent les conditions initiales de l’arène au début de chaque tournoi et sur lesquelles on peut se reposer pour les processus de réinitialisation.

Cette famille comprend 5 fonctions :

* + La fonction visu() permettant de visualiser la map en version machine à tout moment.
  + La fonction init() qui initialise les carte, position et statut de chaque participant.
  + Les fonctions place(), spawn() et check() sont en charge de placer les joueurs aléatoirement sur la carte initiale en évitant les accrochages et la case morte.



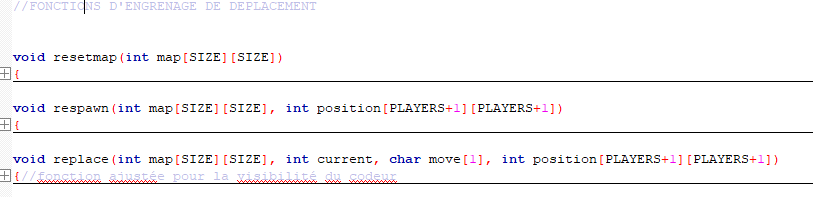
* En suivant, les « fonctions d’engrenage » que ce soit d’action ou de déplacement. Ce sont elles qui font vivre le jeu.

Il ya trois fonctions d’engrenage d’action et trois fonctions d’engrenage de déplacement.

* + Déplacement :

Resetmap(). Avec son nom assez évocateur, cette fonction est appelée à chaque tour pour permettre l’utilisation des deux suivantes.

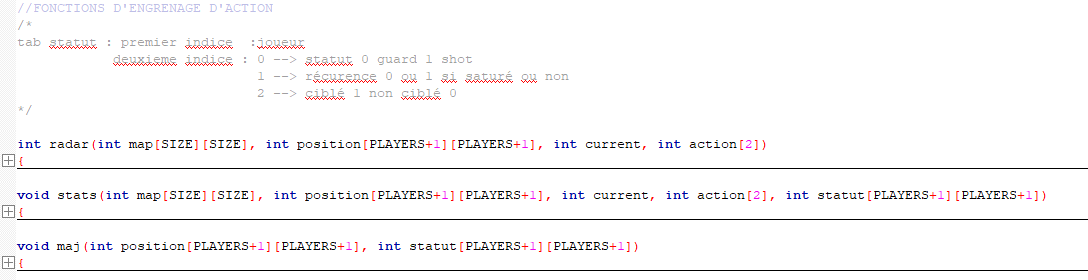
Replace() et respawn() mettent à jour les position de chaque joueur en vérifiant s’ils peuvent continuer à jouer ou s’ils sont éliminés pour faute de déplacement.



* + Action :

Stats() et radar() mettent à jour les statuts des joueurs en prenant compte de leur position (en cas de shoot).

Maj() vérifie s’ils peuvent continuer à jouer ou s’ils sont éliminés par balle.



* Enfin, il y’a le « corps du jeu » qui contient les fonction phase() et jeu() ainsi que le « main » qui concentrent l’appel de toutes les fonctions précédentes.

