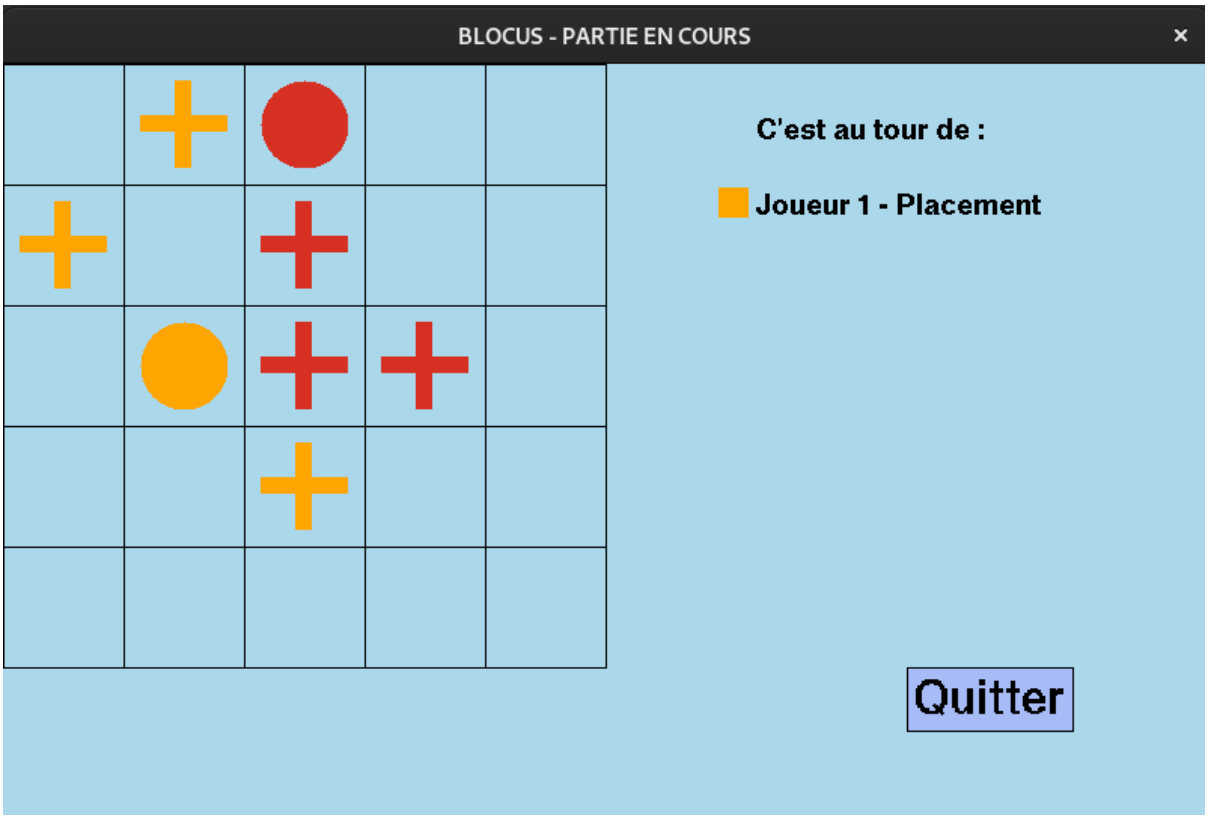
Rapport – Blocus



Théo CHOLLET

Tomy DA ROCHA

1ère année IUT - Fontainebleau

Table des matières

I. Introduction....4
II. Descriptions des fonctionnalités....5
I. Premier écran....5
Fonctionnement....5
Boutons....5
II. Deuxième écran....7
Fonctionnement....7
Moteur principal....8
Fonctions sur la grille....8
Fonctions sur la case....9
Gestion du joueur....10
III. Troisième écran....12
III. Structure du programme....13
Arborescence, découpage en fichiers....13
Optimisation....14
IV. Extras....15
V. Conclusion....16
Théo CHOLLET....16
Tomy DA ROCHA....16

I. Introduction

Premièrement, le jeu doit être jouable uniquement à la souris, par **1 ou 2 joueurs**. Si l'utilisateur choisit 1 joueur, le second participant sera contrôlé par l'ordinateur (une intelligence artificielle). Ensuite, la **taille de la grille** est choisie, de **3 à 9 compris**, afin d'établir une grille carrée (3×3, 4×4, etc.). Le jeu commence : la première étape consiste à placer les joueurs sur la grille : chaque joueur choisit à tour de rôle une position. La seconde étape consiste à alterner un **déplacement** - **blocage**. Le premier joueur choisit une case pour **sa position**, puis choisit une **case à condamner**. C'est ensuite au deuxième de choisir les siennes. Pour la phase de déplacement, les joueurs devront choisir une case **adjacente**, le déplacement oblique étant autorisé. Chaque case occupée actuellement par un joueur est représentée par un pion (**cercle**). Chaque case condamnée est visuellement marquée par une forme d'un "**plus**", et donc plus accessible par les deux joueurs. Le premier des deux joueurs bloqués, a **perdu**. L'écran de fin annonce le **gagnant de la partie**.

Nous avons décidé de présenter la plupart **fonctions** qui nous semblaient pertinentes parmi toutes celles qui ont été codées dans ce **projet**. Nous avons commenté très largement le **code** pour que chaque développeur comprenne les interfaces, le fonctionnement et les définitions des fonctions, mais aussi pour s'y référer pour de **futurs développements**.

II. Descriptions des fonctionnalités

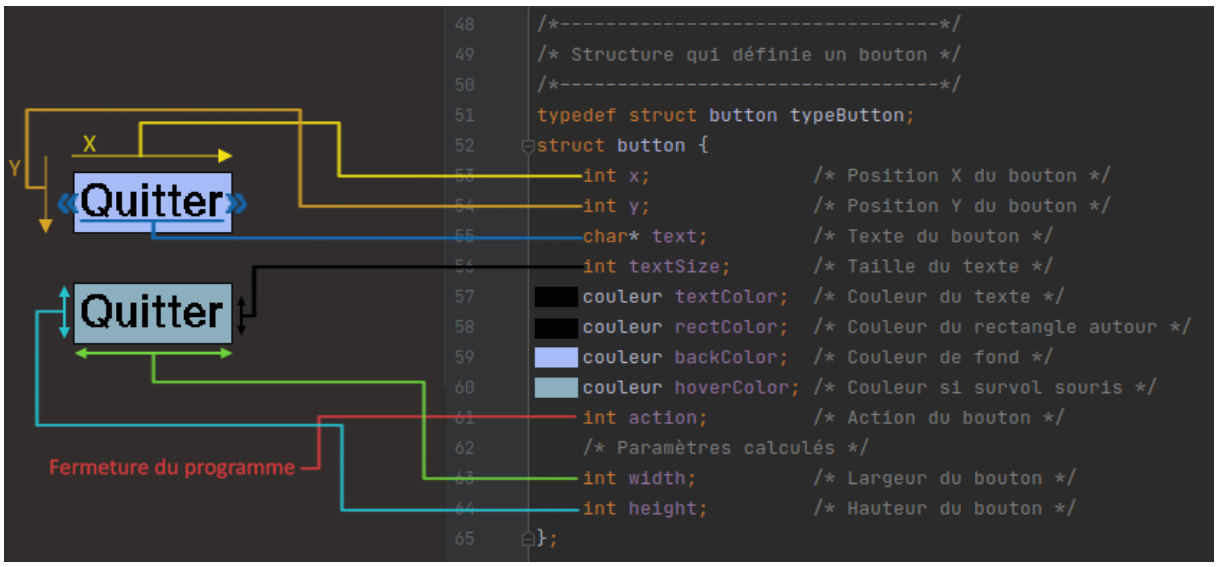
I. Premier écran

Fonctionnement

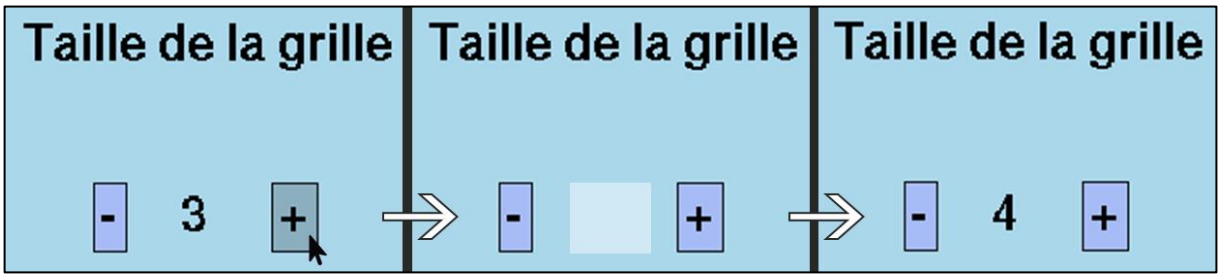
Le premier écran qui concerne le **Menu Principal** du jeu, devait être en grande partie, composé de divers boutons : **Lancement de partie**, **Taille de la grille**, **Quitter le programme**.

Boutons

Étant donné le nombre de boutons à programmer, nous avons rapidement pensé à réaliser une **structure** incluant les différents paramètres que nous pouvions retrouver sur un bouton (son texte, la taille de son texte, sa couleur, etc.).



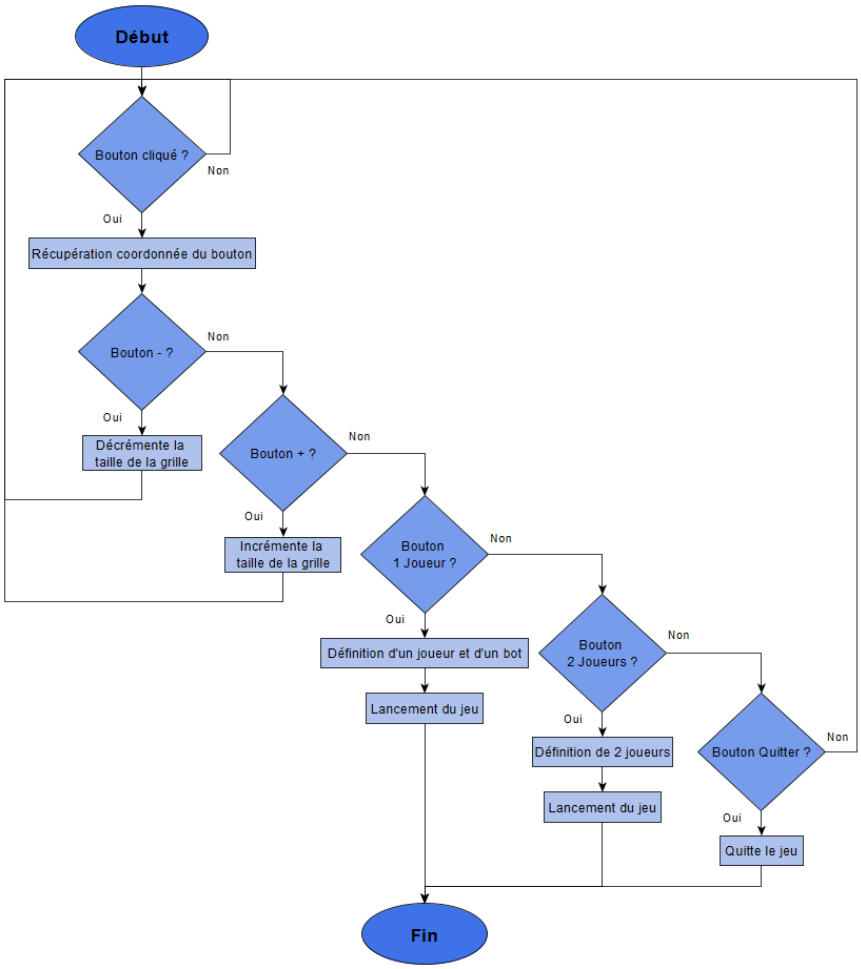
En ce qui concerne la taille de la grille, pour une question d'ergonomie et d'esthétisme, nous nous sommes tournés vers un fonctionnement par **compteur** ("**+**" et "-"), nous trouvions cela désuet de faire (et coder) 6 boutons pour choisir la taille. Le fonctionnement de celui-ci est simple :



Dès que le joueur clique sur le bouton "+", un carré de la couleur du fond apparaît pour masquer l'ancien chiffre, et puis le nouveau chiffre incrémenté de 1 apparaît.

La **fonctionnalité** de chaque bouton se différencie par **l'action** qui leur a été attribuée, nous trouvions l'utilisation d'un "`switch`" approprié. Avec ce "`switch`", nous avons pu regrouper différentes actions selon la **valeur** de la variable qui lui était retournée. De plus, il a l'avantage d'être plus lisible que des conditions écrites à la suite : "`if`, `if else` ou `else`".

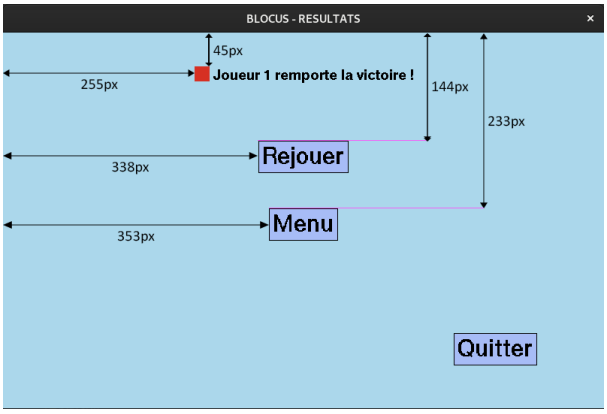
Chaque bouton possède donc une action qui lui est assignée :

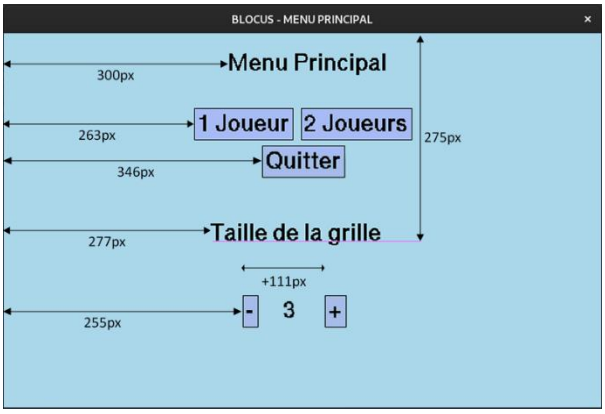


Pour une question d'optimisation et de simplicité, nous avons délaissé l'idée de créer un bouton "**Jouer**" unique. Pour démarrer une partie, l'utilisateur a le choix entre deux boutons : "**1 joueur**" ou "**2 joueurs**".

La variable "`play`" change selon le bouton cliqué, elle reste à 1 pour n'importe quel bouton, mais si le joueur décide d'appuyer sur "**Quitter**", la valeur est mise à 0 et cela quitte la boucle, et ferme la fenêtre graphique.

Enfin au niveau des **coordonnées** de chaque bouton, nous sommes passés par le logiciel **Photoshop** en réalisant une reproduction pour organiser le positionnement des boutons, et évaluer les distances, les ratios par rapport à la taille de l'écran :





Afin de simplifier l'affichage, et l'utilisation des boutons, nous les avons rangés dans un **tableau**. Grâce à une variable correspondante au **nombre de boutons** : "`nbButtons`", nous affichons et traitons tout le tableau avec une simple boucle "`for`". La variable "`action`" est initialisée à "`ACTION_NONE`" (=0), et une attente d'interaction entre le joueur et l'interface est implémentée pour agir sur les différentes variables de **réglages de paramètres**.

Une **fonction** de **recherche de clic** sur un **bouton** a été développée pour **localiser la souris** au moment du clic, et déterminer sur quel bouton le clic était effectué.

Une **fonction** de **dessin** a aussi été développée pour dessiner le bouton avec un calcul automatique de sa **hauteur** et **largeur** selon la **taille du texte** qui était demandée.

II. Deuxième écran

Fonctionnement

Le deuxième écran, qui porte sur le **plateau de jeu**, intègre plusieurs **fonctionnalités**. Dans un premier temps, une **grille** est générée selon la **taille** choisie par le joueur : pour rappel, par défaut, sa valeur est initialement de trois cases par trois cases (taille réglée dans le premier écran).

Le jeu se déroule en **3 phases** :

1. Placement d'un pion : au premier tour.
2. Déplacement / blocage : aux tours suivants.
3. Recherche d'un gagnant.

Chacun leur tour, les joueurs devront disposer leur **pion** dans une **case non-occupée** et bloquer une **case libre** de la grille. Le déplacement est contraint dans les **cases adjacentes** à leur **position actuelle**. Au début de nos développements, nous avions codé le principe suivant : l'ancienne position de pion du joueur était considérée comme une place occupée/ Cela formait un serpent grandissant selon les déplacements de chaque joueur : à la manière de "**TRON**". Nous avons rectifié le tir en recodant avec la possibilité de bloquer n'importe quelle case libre de la grille, après s'être déplacé.

Un **compteur** du nombre de tours effectués est stocké dans la **structure** de chaque **joueur**, celui-ci permet de détecter s'il joue son **premier tour**. De plus, dans cette même structure, nous stockons son **mode de jeu** : phase de **déplacement** ou de **blocage** ainsi que sa **position actuelle**.

Moteur principal

Le moteur principal dépend d'une boucle, elle s'arrête lorsque le bouton "**Quitter**" est cliqué ou sur la fin de partie (un **joueur bloqué**).

La séries d'étapes exécutées est la suivante :

1. Affichage des textes et des boutons.
2. Dessin de la grille de jeu.
3. Calcul du nombre de cases libres pour le joueur en cours.
4. Gestion du joueur en cours :
    - Affichage du tour.
    - Récupération de la case cliquée ou générée (pour l'IA).
    - Positionnement (si possible) du joueur en mode blocage / déplacement.
5. Calcul du passage au joueur suivant.

La fonction **principale** de gestion du deuxième écran retourne le **gagnant** de la partie ou "`NULL`" si le joueur a quitté en cours de partie.

Fonctions sur la grille

Il y a plusieurs **fonctions** qui concernent la **grille**. Constituée de plusieurs **cases**, une **structure** permet d'enregistrer le **tableau de cases** et **son nombre**. Le tableau de cases est indexé en **X** et en **Y** (2 dimensions). Avec autant de lignes que de colonnes afin d'obtenir une **grille carrée**.

Fonction d'initialisation de la grille "InitGrid"

Lors du commencement d'une partie, nous avions besoin d'une **fonction** qui **affiche la grille** en entier. "`InitGrid`" permet de calculer la **taille d'une case** selon la **taille de la grille** demandée au **Menu Principal** (3×3, 4×4, etc.). Avec ces informations, elle dispose chaque case sur **X** et sur **Y**, ce qui forme la **grille de jeu**. Chaque champ de sa structure est paramétré (position, taille, couleur), et notamment le champ "`whoIn`" qui définit **l'état d'occupation** d'une case.

Ainsi, en prenant en entrée le pointeur vers la grille, elle **alloue** suffisamment de **mémoire** pour stocker le nombre de cases en X, et en Y.

La **largeur** et la **hauteur** de la grille dépendent de la taille de la **fenêtre d'affichage** selon un choix que nous avons fait :

$$largeurGrille = largeurEcran / 2$$

La case, quant à elle, dépend de la **largeur de la grille** :

$$tailleCase = largeurGrille / nombreCases$$

Fonction de dessin de la grille "DrawGrid"

Avec cette deuxième **fonction**, nous redessinons la grille **régulièrement** pendant toute la **durée de la partie**. Elle permet ainsi d'actualiser le **placement** des joueurs. Cette méthode d'actualisation continue de la grille permet aussi de **redimensionner** la fenêtre de graphique et d'avoir toujours tous les éléments affichés.

Fonction de détection de case cliquée "GetCaseFromMouse"

Basée sur les mêmes principes que pour les **boutons**, cette fonction permet de récupérer un **clic** dans une **case**. Nous récupérons les coordonnées grâce au pointeur de la souris en X et en Y. Si la fonction détecte que les coordonnées correspondent, elle renvoie 1 pour dire qu'elle l'a trouvée, sinon 0.

Fonctions sur la case

Fonction de dessin de case "DrawCase"

Cette fonction est utilisée dans "`DrawGrid`" (description ci-dessus), elle prend en entrée une case.

Son utilité première est de **dessiner** la case éventuellement occupée par un **pion** correspondant à la **couleur du joueur** actuel grâce au champ "`whoIn`". Ce champ définit qui est dans la case

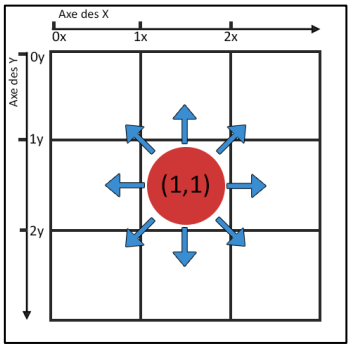
(voir ci-dessous). Sa seconde utilité, quant à elle, est esthétique. Elle ajoute un effet de "**hover**" (de survol) sur les cases parcourues par le passage de la souris, seulement si la **case est libre**.

Si la **case est occupée**, le champ "`whoIn`" sert à décrire quel est le **type de l'occupant** de la case concernée, soit un **pion**, soit une **case bloquée**.

La **fonction** est appelée en permanence dans la boucle de **l'écran 2**. C'est pour cela que nous effaçons la case actuelle avec l'appel de "`ClearCase`". La fonction "`ClearCase`" permet de vider le contenu graphique de la case, donc de la redessiner.

Une fonction de calcul de déplacement disponible "CaseIsPossible" :

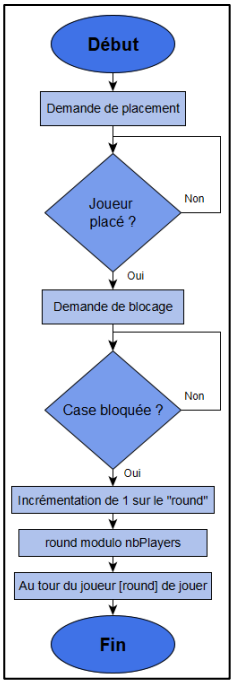
Cette fonction permet de tester si une position de case est possible parmi les **cases libres** autour de la **position** du **joueur actuel**. Elle définit donc les **restrictions**. Si une case est cliquée en dehors de cette zone possible de déplacement du joueur, **rien ne se passe**, et nous attendons que le joueur clique sur une **case libre adjacente**.



Gestion du joueur

Affichage et calcul du tour du joueur

Sur la **partie droite** de la fenêtre, il est indiqué le **tour du joueur** et la **nature de son action** (**placement** ou **blocage**). Pour se faire, une **fonction** prenant en compte le "`round`" est utilisée, et fonctionne comme ceci :



Les joueurs sont rangés dans un **tableau**. A chaque fois qu'un joueur finit de jouer ses coups, une **variable** "`round`" qui définit **l'indice du joueur** dans le **tableau**, est **incrémenté** de 1. Nous appliquons alors un **modulo** de "`nbPlayers`" (ici 2 joueurs) pour remettre automatiquement à 0 cet indice de tour de joueur. Ce qui fait que nous passons d'un joueur à l'autre, sans se préoccuper du nombre de **joueurs définis**. Par exemple, pour 2 joueurs, **l'index** "`round`" est initialisé à **0**, ce qui veut dire que c'est au **joueur 1** de jouer, il choisit une case, puis bloque une **case vide**. Enfin, la valeur de l'index sera incrémentée et passe à 1, à la prochaine incrémentation de cet index, il sera à 2 modulo 2 = 0 (retour au **premier joueur**).

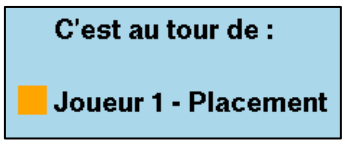
Fonction de gestion des joueurs "ManagePlayer"

La **fonction** permet de **gérer le tour** de chaque joueur, et ses **interactions** avec **l'écran** dans sa globalité. Nous avons dû ajouter en paramètre le bouton quitter dans cette fonction afin d'appeler qu'une seule fois la fonction et savoir si le bouton **a été cliqué**.

L'action de **l'Intelligence Artificielle** est définie telle que, si elle n'est pas placée, elle choisit une case dans sa **portée**, si elle doit choisir une case à **bloquer**, nous cherchons un **adversaire** à bloquer. Pour cela, nous choisissons un **index aléatoire** de joueur (fonction "`rand % nbPlayers`") dans le tableau. Cela n'a pas de conséquence si le nombre de joueurs est de 2, mais permet de facilement implémenter une version **multi-joueurs** en Intelligence Artificielle. Si cet index est lui-même, nous imposons de refaire appel à la fonction "`rand()`" jusqu'à ce qu'un nouvel index soit généré.

Fonction "DisplayPlayerRound"

Cette fonction permet d'afficher l'**état d'avancement** dans la partie. En prenant comme argument le pointeur vers un joueur, nous affichons sur le coin supérieur droit le type du joueur actuel, soit "**CPU**" pour l'IA sinon "**Joueur**". Nous affichons un carré de la couleur correspondante au joueur avec son numéro (de 1 à x joueur(s)).



Fonction de dessin "DisplayEndGame"

Cette fonction sert à dessiner au milieu de l'écran un **message** qui indique au(x) joueur(s) que la **partie est finie**, et après un court temps d'arrêt, va passer à l'écran 3.

Fonction de placement de joueur "PlacePlayer"

La fonction de **placement de joueur** prend en compte la case sélectionnée par un joueur, en mettant à jour sa **position**. Cette **fonction** fait appel à la fonction "`CaseIsPossible`", une fonction qui permet de tester si **la case choisie** est disponible pour un placement (déplacement adjacent seulement), selon la **position actuelle du joueur**.

De plus, lorsqu'une case est condamnée par un blocage, son paramètre "`whoIn`" est modifié en "`CASE_MODE_BLOCKED`", il est donc impossible d'y poser quoique ce soit. Dans la **structure** qui définit l'état de la case, sont stockés **différents types de valeurs** :

- `CASE_MODE_FREE` : La case est **libre**.
- `CASE_MODE_PAWN` : La case est **occupée** par un **pion**.
- `CASE_MODE_BLOCKED` : La case est **occupée** par une **croix**.

Fonction de recherche de cases par l'IA "GetIAFreeCase"

La spécificité du **mode "1 joueur"**, c'est l'utilisation d'un adversaire de type **IA**. Pour **l'algorithme**, nous avons utilisé une simple fonction aléatoire, qui sélectionne une case autour de sa position. Pour ce qui concerne ses **blocages**, n'étant pas totalement satisfait de leurs **positionnements**, nous avons ajouté le fait qu'elle les pose vers la **position actuelle** d'un joueur c'est-à-dire le **joueur humain** du moins pour le mode "1 joueur".

Fonction de libération de la mémoire "FreeGrid"

Elle est appelée à la fin de l'écran 2 afin de **libérer la mémoire** allouée pour l'initialisation de la **grille de jeu**, et libérer chaque case, une après l'autre (dans la fonction "`InitGrid`").

III. Troisième écran

L'écran de fin est appelé à **deux conditions** :

- Soit l'utilisateur a voulu quitter la partie en appuyant sur le **bouton** prévu à cet effet, il n'y donc **pas de gagnant**.
- Sinon la partie est finie et il y a un **gagnant**, qui est forcément le **joueur précédant** celui qui ne peut plus faire de **déplacement**.

III. Structure du programme

Arborescence, découpage en fichiers

Pour **structurer** le **programme**, nous avons réfléchi dès le début de sa création aux **fonctions** que nous devions implémenter. Il est apparu que des fonctions portaient sur le **même thème**, ou **sur des objets** que nous manipulions régulièrement : les **joueurs**, les **écrans**, la **grille**, les **cases**... Et pour les regrouper, il suffisait de les trier dans des **fichiers** selon leurs **fonctionnalités** et sur quoi ils agissaient : "`players.c`", "`display1/2/3.c`", "`grid.c`", etc.

De plus, l'importance de créer un fichier regroupant **différentes constantes**, était selon nous essentiel, car il permettait de se **retrouver** dans le programme et d'éviter de coder en les rappelant régulièrement, par exemple pour la **taille de l'écran**, des **couleurs prédéfinies**, des **actions**, etc. Chacun de nous pouvait aussi facilement développer dans un **fichier** pendant que l'autre **développeur** était sur d'autres. Malgré ce découpage, nous avons beaucoup travaillé en partage d'écran.

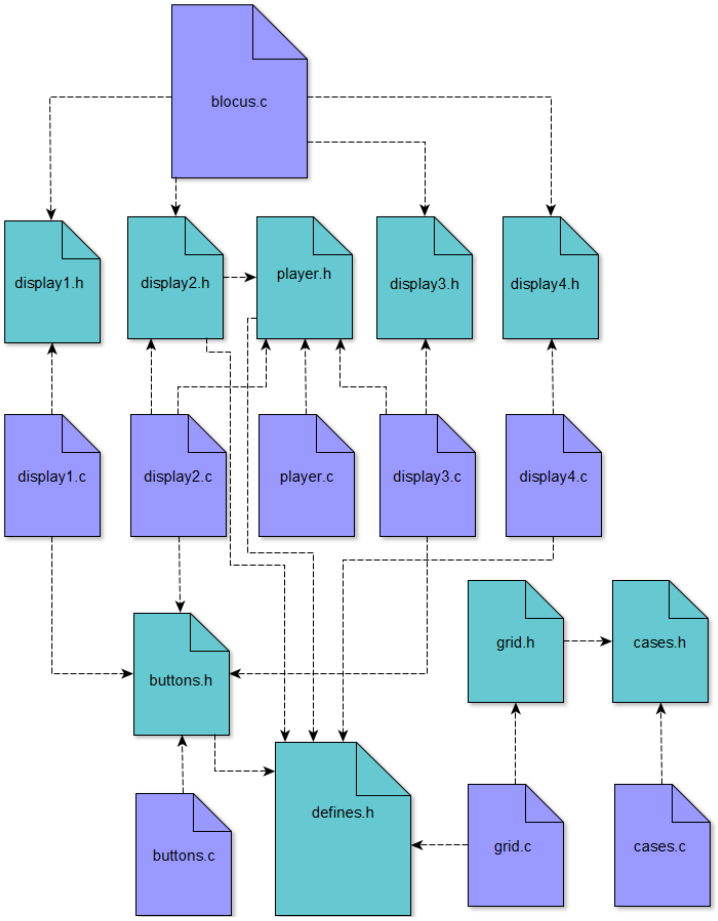
La fonction "`main`" se trouve dans le fichier "`blocus.c`", c'est dans ce fichier que le jeu va appeler les **différentes fonctions** nécessaires à l'affichage de chaque écran.

Nous avons décidé de regrouper toutes les **définitions communes** dans le fichier "`defines.h`", où nous avons défini une série de constantes ("`define`") qui permettent de paramétrer facilement le fonctionnement du programme en termes de :

- Temps d'attente.
- Taille de l'écran, de la grille.
- Couleurs.
- Action des boutons.

Pour tous les autres fichiers de définitions **d'entête** ("`.h`"), nous avons aussi défini des constantes ("`define`") mais aussi les **types et structures** qui concerne chaque élément du programme, par exemple pour "`player.h`" :

- Type de joueur.
- Sa position actuelle dans la grille.
- Sa couleur de pion.
- etc...



Et voici donc un diagramme des dépendances de fichiers.

Optimisation

Nous avons remarqué que le "**CPU**" était souvent **trop sollicité** quand nous attendions une action de la souris par l'utilisateur dans les **boucles infinies**. En réponse à ce problème d'optimisation, nous avons décidé d'utiliser un "`sleep()`" (temps d'attente) de quelques **millisecondes**. La fonction "`usleep()`" remplit ce rôle, malgré des avertissements ("warning") à la compilation, de définition implicite. Nous avons essayé de régler ce problème en utilisant plutôt "`nanosleep`" mais nous n'avons point réussi à la faire fonctionner. Ce problème s'est posé à un **stade intermédiaire** du développement du jeu, et après quelques recherches, nous avons vu qu'utiliser du **multithreading** pourrait améliorer grandement les performances du jeu. Or, nous ne l'avions jamais expérimenté auparavant, donc nous n'avons pas décidé d'appréhender cette notion "en cours de route". Nous avons cherché les meilleures **valeurs** de temps d'attente pour que le jeu puisse répondre aux **actions** de l'utilisateur.

IV. Extras

Nous avons glissé ce qu'on appelle un **œuf de pâques** en bon français, à l'intérieur de notre jeu. Vous pouvez le découvrir par vous-même ou bien le chercher dans le code.

Indice : Tenter d'agrandir la fenêtre dans le **Menu Principal**... ;)

Pour tester notre **architecture**, et s'amuser pendant sa conception, nous avons essayé d'introduire plus de **2 IA**, qui se battent entre elles. Cela nous a aiguillé vers quelques "bugs", pour enfin arriver à une **version stable**. Cela nous a permis de **modéliser** plusieurs cas d'utilisation qui sont testés finalement par le "**CPU**" lui-même. Nous avons donc dissimulé hors des coordonnées de l'écran un bouton supplémentaire afin de faire partager cette **expérience unique**.

V. Conclusion

Théo CHOLLET

Ce projet a été pour moi une réelle concrétisation de ce que j'avais pu expérimenter en cours précédemment. Malgré un début difficile pour la maitrise des pointeurs ou des structures, le fait de les avoir utilisés tout au long de notre programme, m'a permis de mieux les appréhender et de comprendre leur puissance. Avant de commencer à se plonger directement dans du code, mon camarade et moi avions mis en place sous le format texte, différentes idées autour de la création du programme. Le fait d'avoir fait ce travail de préambule, a été un point important pour moi pour ne pas me perdre lors de son développement. Sur le travail en groupe, il s'effectuait d'abord chacun de notre côté, par la définition de plusieurs tâches par personne. Par la suite, nous avons trouvé selon moi, un meilleur compromis, nous nous connections à distance en même temps, pour travailler sur une session en partage d'écran. L'idée de la répartition des tâches fut donc abandonnée, au détriment d'une avancé plus rapide sur un sujet précis. Il est à noter que nous avons eu un problème de push/pull sur notre git, que l'on a tout de même pu régler, mais dont j'en tire une bonne leçon : ne jamais travailler sur le même fichier en même temps. Ce format de création de projet, m'a poussé à expérimenter différentes étapes avant d'aboutir à un résultat, tout en me stimulant durant l'entièreté de son élaboration.

Tomy DA ROCHA

Avant de commencer à programmer le jeu, j'ai longuement réfléchi et conceptualisé sur papier, à l'aide de schémas des fonctionnalités que nous aurions probablement eu à utiliser plus tard. Et puis en commençant d'abord par la création d'un menu, des problèmes sont venus se greffer. Les revues de codes régulières ont permis d'optimiser la plupart des fonctions, de les mutualiser jusqu'à arriver à cette version. Le travail en groupe étant plus difficile à distance, nous nous sommes adaptés en faisant régulièrement des sessions avec partage d'écran, en essayant de réunir nos idées pour la suite du projet. Le travail de finition fut selon moi le plus complexe, en testant de nombreuses fois le jeu, il m'arrivait de remarquer différents bugs qui devaient être corrigés. En ajoutant le fait que nous voulions souvent optimiser notre programme, suite à cela, il arrivait que d'autres bugs fassent leurs apparitions. Malgré de longues sessions de développement qui s'avéraient parfois éprouvantes, la satisfaction ressentie à clôturer le jeu en valait la peine.