Le Mans Université

Licence Informatique *2ème année*

Module Conduite de projet

Rapport de projet : Picruise

Mahouin Julien, Alexis Noliere, Lemrabott Mohamed, Gallup Mohamad Jamil

https://github.com/Projet-Picross

19 avril 2021

Table des matières

1 Introduction

Dans le cadre du module de conduite de projet, il a été décidé de développer un jeu de type picross, ici nommé Picruise, en langage C et SDL.
Le principe d'un tel jeu est de remplir une grille carrée à l'aide de cases noires en respectant un certain arrangement de blocs de cases noires.
Afin de rendre Picruise plus original, des fonctionnalités telles qu'un mode aléatoire, un solveur, une barre de sorts et un design en pixel art ont été définies comme objectifs.
Ce rapport présentera d'abord les règles de Picruise ainsi que ses fonctionnalités. Puis seront définies les procédures de développement et l'utilisation des outils pour chacune des fonctionnalités précédentes. Pour enfin conclure sur l'état final du projet et une analyse critique de celui-ci.

2 Organisation du travail

Tout d'abord, le projet a été divisé en grandes fonctionnalités : interface, système de jeu, etc. Chaque fonctionnalité fut alors divisée en plusieurs sous-fonctionnalités. Par exemple pour le système de jeu, les sous-fonctionnalités sont : noircir une case, blanchir une case, barrer une case. Un diagramme de Gantt a été utilisé (visible en fin de rapport) afin d'organiser le projet sous un format graphique, permettant une vue d'ensemble rapide.

Ces sous-fonctionnalités ont alors été les tâches attribuées en début de séance en fonction du niveau et des outils de chacun. Par exemple, pour la SDL, tout le monde n'ayant pas réussi à installer TTF, il a fallu déléguer à ceux pouvant l'utiliser.
Ainsi, chaque membre du groupe se fixe un objectif pour la séance (par exemple parvenir à faire la génération d'une grille aléatoire). Des tâches plus compliquées sont parfois attribuées à 2 personnes ou plus. À la fin de la séance, un bilan est fait sur l'avancement de chacun. Si une tâche attribuée n'a pas été terminée, elle est terminée ou avancée en dehors de la séance et déposée sur le dépôt Git (https://github.com/Projet-Picross) ou notre serveur discord dédié au projet. Ces deux supports étant les outils définis pour le partage du travail au sein de notre groupe (partage de fichier, partage d'écran).
Le suivi du travail en dehors des séances est alors assuré par des micro-plannings propre à chaque semaine.

Concernant l'organisation chronologique, nous avons commencé à travailler sur la génération de grilles aléatoires et prédéfinies. La génération de grilles aléatoires, un peu plus compliquée, a été réalisée par Julien, Mohamed et Gallup. De son côté, Alexis, s'est occupé de la génération de grilles prédéfinies.

En même temps, Julien est arrivé à générer les conditions des lignes et des colonnes.
Pour afficher ces conditions et la grille, Julien, Mohamed et Alexis ont travaillé ensemble. Alexis est arrivé aussi à faire un fichier de sauvegarde en même temps.
La résolution du solveur a été la partie qui a pris le plus de temps de par sa difficulté. Cette fonctionnalité a été faite par Mohamed, Alexis et Julien. Chacun ayant différentes fonctions à implémenter.
Pour l'interface graphique, chacun a pu réaliser une partie ou une sous-partie. Gallup a réalisé la base de l'affichage de la grille et l'interaction avec celle-ci. Ainsi que la résolution des problèmes d'installations de la SDL et TTF pour les autres membres du groupe. Julien a alors pu réaliser l'affichage des sprites (assemblés par lui et Alexis) et les menus. Alexis, de son côté, a fait la structure de toutes les matrices prédéfinies. Enfin, Mohamed a réalisé le tutoriel du jeu.

3 Conception

Picruise est un jeu de type picross simple. La partie suivante présentera ses règles.

3.1 Présentation du jeu

L'objectif pour le joueur est de remplir une grille carrée avec des cases noires, ceci en respectant les conditions de cette grille. Une condition est une série de chiffres définissant le nombre de cases noires pour chacun des blocs de cases noires d'une ligne ou d'une colonne.

Dans le cas de l'exemple suivant (fig. 1), la ligne encadrée en rouge a pour conditions : 2, 1, 1. C'est-à-dire que pour être considérée comme valide, elle devra comporter trois blocs de, respectivement, deux cases noires, une case noire et encore une case noire, de gauche à droite. Deux blocs doivent être séparés par au moins une case blanche.

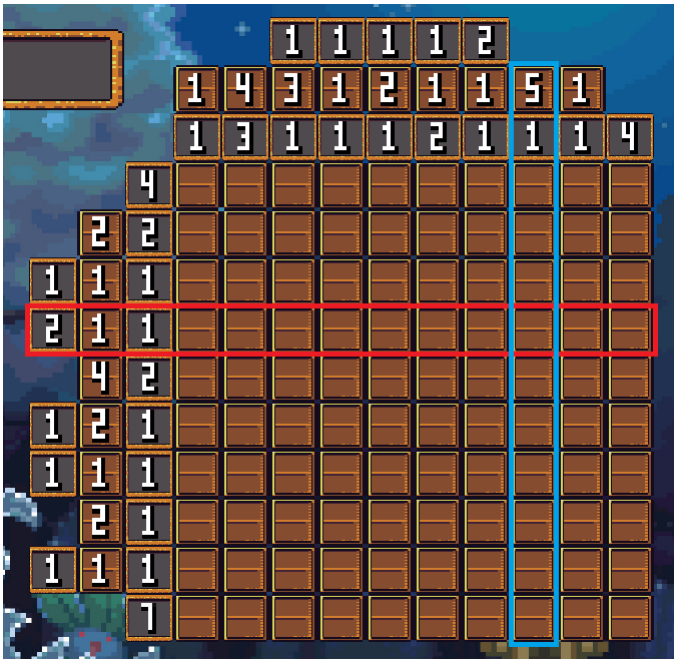


Fig. 1. Capture d'écran de la matrice "Canard" vide

De même, ces règles s'appliquent sur les colonnes, dans le cas précédent, la colonne encadrée en bleu devra comporter un bloc de cinq cases noires puis un bloc d'une case noire, de haut en bas.

Le joueur n'a ni contrainte de temps, ni contrainte d'actions pour terminer la grille. La partie se termine une fois que toutes les conditions des lignes et des colonnes sont validées simultanément.

3.2 Fonctionnalités du programme

3.2.1 La création de grilles

Afin de rendre les différents niveaux diversifiés, il a été décidé de créer préalablement des grilles et de permettre au jeu d'en générer aléatoirement.

Ainsi, Picruise est doté de deux modes de jeux, respectivement intitulés "Prédéfinies" et "Aléatoire", que l'utilisateur peut choisir avant de démarrer une partie.

Tout d'abord, le mode de jeu "Prédéfinies" offre la possibilité à l'utilisateur de résoudre des grilles dont le nombre et l'arrangement des cases noires sont préalablement établis. Ceci dans le but d'obtenir comme résultat final une forme précise. Cette image peut représenter un animal, un objet du quotidien ou encore une personne. Comme le montre l'image suivante (fig. 2) la matrice, une fois terminée, dessine l'image d'un canard en plastique.

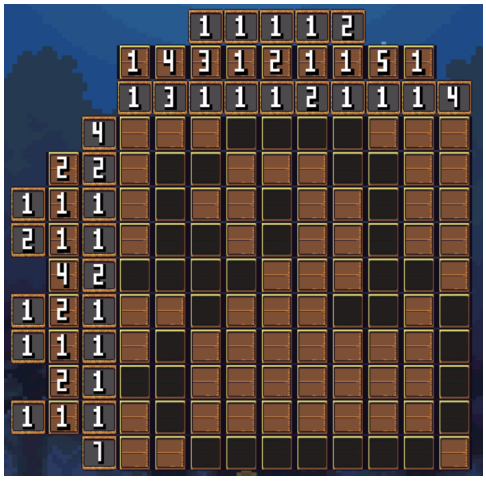


Fig. 2. Une matrice prédéfinie résolue représentant un canard

Ensuite, vient le second mode de jeu que l'utilisateur peut choisir, le mode "Aléatoire". Dans ce mode de jeu, l'utilisateur devra de nouveau faire un choix avant de pouvoir jouer : choisir la taille de la grille. En effet, le joueur a à sa disposition 4 dimensions de grilles possibles qui sont 5x5, 10x10, 15x15, et 20x20. En plus de fournir une ligne de progression pour le joueur, nous nous limitons à un format maximal de 20x20. Ceci afin de garder un affichage relativement lisible et une interface confortable pour l'utilisateur. Une fois les dimensions choisies, le jeu va générer aléatoirement une grille, chaque grille ainsi obtenue comportant un nombre de cases noires allant de 25% à 75% du nombre total de cases. Ceci afin de ne pas avoir de grille ni trop pleine ni pas assez, rendant la résolution soit trop simple, soit trop compliquée.

3.2.2 Le solveur

L'une des consignes de ce projet était la réalisation d'un algorithme de résolution automatique d'une grille de picross.

Le solveur doit permettre au programme de terminer une grille, aléatoire ou pré-définie et d'une taille variable, de zéro. Celui-ci suit une stratégie par "force-brute", c'est-à-dire qu'il teste toutes les combinaisons, ici de lignes, possibles jusqu'à terminer la grille.
Afin de se rapprocher d'une véritable intelligence artificielle, l'algorithme devait être accompagné de fonctions de remplissage évident. Chacune de ces fonctions correspondantes à un cas particulier pour une ligne ou colonne :

— Cas d'une ligne / colonne totalement vide :

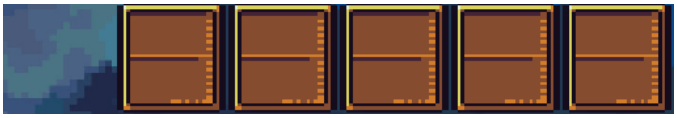


Fig. 3. Capture d'écran d'une ligne vide

— *C*as d'une ligne / colonne pleine :

Fig. 4. Capture d'une ligne pleine

— *C*as où les conditions ne permettent qu'un seul arrangement fixe et évident :

Fig. 5. Capture d'une ligne n'ayant qu'un arrangement possible

3.2.3 Permettre à un utilisateur de jouer

Un jeu sans donner la possibilité au joueur d'y interagir, n'est pas vraiment un jeu, en utilisant la librairie SDL, il a été possible de mettre en place un système d'interaction avec celui-ci.

En lançant Picruise, le joueur se trouve à la page d'accueil qui contient le menu (fig. 12), composé de plusieurs boutons : Jouer, Tutoriel, Options et Quitter, chaque bouton est consacré à un travail spécifique :

— *Jouer* : choisir entre les deux modes de jeu disponibles : mode aléatoire et mode prédéfinie
— *Tutoriel* : indique des informations sur la façon de jouer, les règles du jeu
— *Options* : changer la résolution de la fenêtre (1920x1080px, 1366x768px, 720x480px)
— *Quitter* : quitter le jeu

Le mode aléatoire, comme expliqué précédemment, donne la possibilité de sélectionner la taille de la grille entre 5x5, 10x10, 15x15 et 20x20, puis le joueur peut décider de jouer ou de laisse le solveur compléter la grille générée aléatoirement. Alors que la mode pré-définie, vous permet de choisir entre les différents grille qui a été faites au préalable (fig. 2), une fois la grille terminé, vous trouverez l'image correspondant à la grille choisi.

Pour terminer la grille, il faut cocher les cases en prenant en compte les conditions des lignes et des colonnes. Le joueur peut ici noircir, "blanchir" ou barrer une case. Cette dernière action ayant pour but de marquer une case potentiellement blanche ou éviter de noircir une case par erreur.
Une fois toutes les cases cochées avec la bonne répartition, un message de victoire et deux boutons s'affichent, l'utilisateur peut soit quitter et revenir au menu principal, soit recommencer la partie.

4 Développement

La suite du rapport présentera les données, algorithmes et outils utilisés dans la mise en œuvre des fonctionnalités de la partie précédente.

4.1 La génération des grilles

Du point de vue algorithmique, une grille de picross est une matrice de N lignes sur N colonnes, N étant le nombre de cases par ligne et colonne. De ce fait, la matrice, correspondant au canard, montrée précédemment est une matrice 10x10.
Cette matrice est une matrice d'entier prenant les valeurs 0 pour une case blanche et 1 pour une case noire. Tout d'abord, il faut savoir que les grilles prédéfinies et aléatoires sont gérées de deux façons différentes, bien que les deux soient associées à des matrices carrées. En effet, puisque que les grilles prédéfinies sont censées représenter une forme ou une image une fois résolues, il est normal d'y apporter une attention un peu plus accrue.

Dans le cas des grilles prédéfinies nous avons décidé de stocker dans plusieurs fichiers l'aspect final de chacune d'entre elles. Comme nous le montre l'image ci-dessous (fig. 6), le fichier nommé *canard.txt* comporte une suite de nombres.

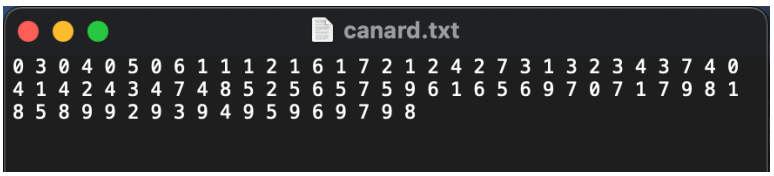


Fig. 6. Capture d'écran du fichier de la matrice canard

Ces nombres représentent des couples de coordonnées indiquant les emplacements des cases noires dans la matrice afin d'obtenir le rendu final préalablement voulu. Par exemple sur la capture d'écran précédente (fig. 6), les deux premiers chiffres, 0 et 3, indiquent qu'une case noire doit être placée à la case de ligne 0 et de colonne 3.

À l'heure actuelle, nous possédons 27 fichiers textes contenant chacun une matrice prédéfinie et dont leur implémentation suit la même syntaxe que celle du fichier *canard.txt* vus précédemment. Afin de gagner du temps et limiter les erreurs, nous n'avons pas rentré manuellement tout ces nombres, mais nous avons utilisé un programme intermédiaire (nommé *analyse.c*) qui a généré les cordonnées à partir d'un modèle. Plus précisémment nous avons saisi dans un fichier texte des O pour les cases blanches et des X pour les cases noires, puis le programme a inscrit les coordonnées correspondantes aux X dans notre fichier global (regroupant toutes les matrices prédéfinies).

Il est d'abord important de noter que chaque génération aléatoire qui va suivre est réalisée avec la fonction *rand()* de la bibliothèque *time.h*.
Afin de générer une grille aléatoire, il a été décidé de d'abord générer le nombre de cases noires pour une grille. Nombre qui, comme dit précédemment, doit être compris entre 25% et 75% des dimensions de la grille. Comme le montre les deux images suivantes (fig. 7 et 8), des matrices trop pleines sont trop simples à terminer tandis que l'inverse génère des matrices pouvant être trop compliquées.

Fig. 7. Trop de cases noires

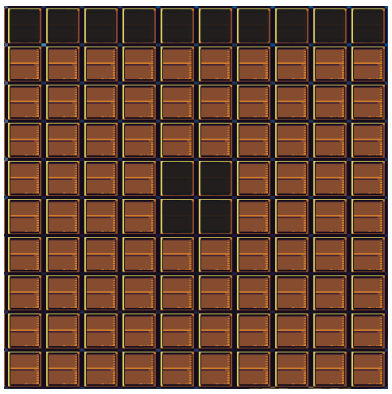


Fig. 8. Pas assez de cases noires

Puis, tant que ce nombre de cases n'a pas été atteint, le programme sélectionne une case au hasard et, si elle est blanche, la colorie. Sinon, il en sélectionne une autre jusqu'à tomber sur une case blanche.
Ainsi, nous sommes assuré d'avoir des grilles parfaitement aléatoires, que ce soit sur le taux de cases noires ou leur position.

4.2 L'algorithme du solveur

La stratégie définie se base sur une méthode par "force-brute".
Une première fonction génère tous les arrangements possibles pour chaque ligne de la matrice dans une matrice cubique de la forme :
int combinaisons[format][arrangement][bloc] avec :

- *format* : le numéro de la ligne (allant jusqu'à format, la taille de la matrice)
- *arrangement* : la n-ème possibilité d'arrangement pour cette ligne
- *bloc* : le n-ème bloc de cases noires ou blanches, l'ordre des blocs suivants le schéma : bloc blanc, bloc noir, bloc blanc, etc

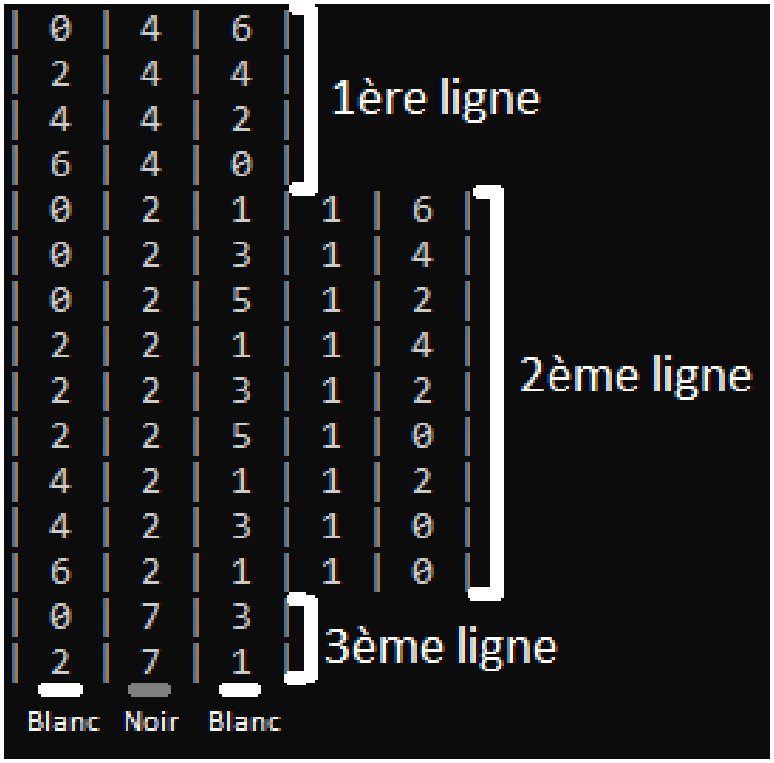


Fig. 9 Exemple de tableau d'arrangements possibles pour une ligne

Dans le cas précédent (fig. 9), nous avons quatre arrangements possibles pour une ligne d'un bloc de quatre cases noires, puis neuf arrangements pour une ligne d'un bloc de deux cases noires puis un bloc d'une case noire, et enfin deux arrangements pour une ligne d'un bloc de sept cases noires.
La première possibilité (0|4|6) étant de la forme :

Fig. 10 Exemple de tableau d'arrangements possibles pour une ligne

Et la huitième (2|2|1|1|4) :

Fig. 11 Exemple de tableau d'arrangements possibles pour une ligne

Ensuite, nous entamons une boucle qui, pour chaque ligne de la grille, va regarder le nombre de cases noires présentes (d'où l'utilité de notre tableau intermédiaire) ainsi que leur répartition en blocs. Par exemple, si la première ligne comporte 7 cases noires réparties en 2 blocs, alors on sait qu'il y aura

au moins une case blanche entre ces deux blocs noirs (une des règles de base du picross). Ce processus nous permet donc de déterminer combien de cases blanches sont à placer dans la ligne courante. La fonction se sert de la condition associée à la ligne courante pour déterminer le nombre de blocs noirs et suit des contraintes afin de déterminer le nombre de blocs où l'on va répartir les cases blanches. En effet, admettons que nous ayons une matrice de taille 10 et que nous ayons toujours nos 7 cases noires réparties en 2 blocs (disons un bloc de 4 et un bloc de 3). Nous pouvons donc avoir une ligne de forme :

— 4 cases noires - 3 cases blanches - 3 cases noires
-> soit 1 bloc d'une case blanche
— 1 case blanche - 4 cases noires - 2 cases blanches - 3 cases noires
-> soit 2 blocs de cases blanches
— 1 case blanche - 4 cases noires - 1 case blanche - 3 cases noires - 1 case blanche
-> soit 3 blocs de cases blanches

Une fois le nombre de blocs déterminé, la fonction va répartir le nombre de cases blanches dans chacun de ces blocs et enregistrer cette position dans notre matrice cubique. Lorsque tous arrangements possibles sont créés pour la ligne courante, on passe à la ligne suivante de la grille, on se déplace dans la matrice cubique et on recommence ce procédé. À la fin de cette fonction, nous avons une matrice cubique contenant toutes les solutions possibles pour notre grille, il ne reste donc plus qu'à déterminer la bonne combinaison.

Ensuite, le corps du solveur consiste en une fonction récursive testant chacune des possibilités pour une même matrice jusqu'à trouver un arrangement valide par rapport à l'ensemble des conditions des lignes et des colonnes. En effet, même si une grille peut avoir différentes solutions, il est inutile de toutes les générer.
Ainsi, le solveur parcourt chaque ligne de la grille et remplie la matrice avec un arrangement de ces lignes. Après chaque remplissage complet de la grille, le solveur s'assure qu'elle vérifie chaque condition des lignes et des colonnes. Si c'est le cas, alors la première solution est trouvée et il n'y a pas besoin d'aller plus loin. Sinon, il teste les possibilités suivantes.
Il est important de noter que toute grille a au moins une solution. En effet, lors de la création d'une partie, les conditions sont générées à partir de la grille et non l'inverse.
Une fois la première solution trouvée, celle-ci est sauvegardée et affichée à l'utilisateur.

4.3 L'affichage

L'intégralité de l'affichage a été réalisée à partir de la bibliothèque SDL, pour Simple DirectMedia Layer.
Pour commencer, les menus. Il est tout d'abord important de noter que chaque menu correspond à une fonction. Par exemple, le menu d'accueil, ou l'écran titre, correspond à la fonction "accueil". Ce choix a été réalisé afin de faciliter la navigation entre les menus que nous verrons plus tard.

Outre le timer, nous avons opté pour l'utilisation d'images plutôt que de rectangles et de textes. Ceci afin d'obtenir un rendu plus agréable, mais aussi pour avoir une plus grande souplesse sur les modèles utilisés, ici des sprites.
Dans le cas du bouton "Aléatoire", celui-ci est le fruit de l'assemblage du sprite d'un rectangle redimensionné et d'une police d'écriture au format png. Par l'intermédiaire de l'outil graphique Gimp, chaque sprite est découpé et placé sur un calque transparent. Puis les lettres sont superposées sur le rectangle, le tout étant ensuite redimensionné afin d'éviter un éventuel flou lors du redimensionnement par SDL.

Ces images sont alors utilisées afin de développer le rendu visuel avec la bibliothèque SDL.
Après la création de la fenêtre, un rendu est créé, celui-ci sera distribué en tant que paramètre pour chaque fonction de menu appelée, permettant ainsi de gérer la libération des ressources en un simple point du programme. Plus précisément, une fois que le bouton "quitter" de la page d'accueil est préssé et donc, que le programme retourne dans le *main*.
Chaque image chargée suit le même processus. Elle est est d'abord appelée en tant que surface puis une texture est créée à partir de celle-ci et du rendu passé en paramètre. Enfin, elle est affichée selon un rectangle définissant les coordonnées et dimensions d'affichage voulues pour l'image.

4.4 L'interaction de l'utilisateur

La gestion d'événements se fait à l'aide de la fonction *SDL_WaitEvent* pour laquelle le programme attendra que le joueur clique sur un bouton, ici le fait de relâcher le clic gauche de la souris.
Afin de simplifier ce processus, mais aussi pour permettre une adaptation automatique au format de la fenêtre, les positions et dimensions de chaque bouton sont définies par des formules pré-établies.
L'image suivante (fig. 12) présente, en noir, le rectangle correspondant au bouton jouer avec, en bleu la position en x (horizontal), en vert la position en y (verticale) puis la largeur et la hauteur respectivement en rouge et en rose.

Fig. 12. Capture d'écran de l'écran de titre

Ainsi, pour déterminer si l'utilisateur a cliqué sur le bouton, il suffit de vérifier que les coordonnées de la souris, au moment de relâcher le clic, sont comprises entre la position initiale en x et la largeur additionnée à cette position, de même pour la position en y avec la hauteur.

Comme dit précédemment, chaque menu est une fonction. Et chaque fonction est liée à celles des menus qui lui sont directement liés. Donc, pour passer d'un menu à un autre, il suffit de déterminer le bouton correspondant sur lequel l'utilisateur à cliqué. Puis lancer la fonction du menu avec les bons paramètres. Dans le cas du menu de choix du format de la grille aléatoire, par exemple, si l'utilisateur clique sur le bouton "10x10", le paramètre 10 sera alors passé dans les menus suivants.
Il en va de même pour le parcours inverse des menus, un bouton retour renvoyant alors à la fonction du menu précèdent.
Vous pouvez consulter l'architecture des menus sur l'organigramme en annexe 7.

En ce qui concerne l'interaction de l'utilisateur avec la grille, la même méthode est utilisée. À l'aide des fonctions prédéfinies de la librairie SDL, on détermine les coordonnées de la souris, et on vérifie si le curseur est entré dans une case, et pour cela, grâce à deux boucles, on parcourt la grille, et on vérifie l'emplacement des cases par rapport au curseur en additionnant la

position et la taille de chaque case. (fig. 13)

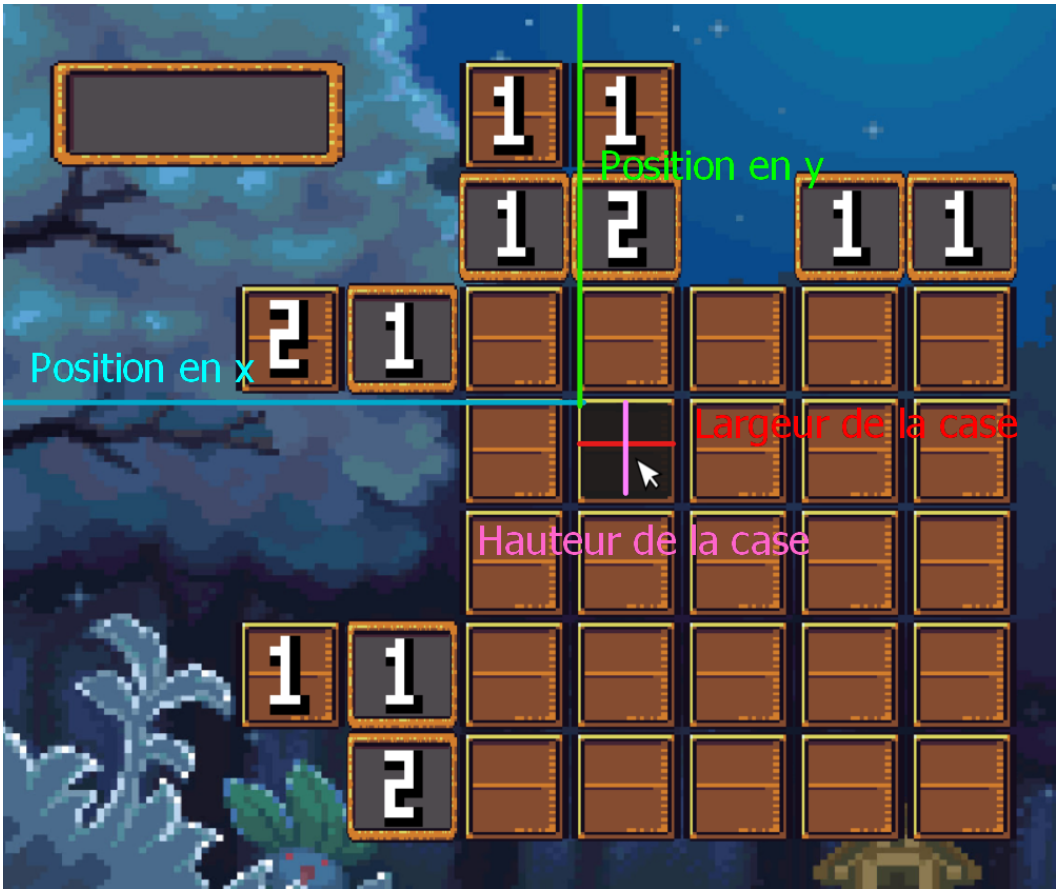


Fig. 13. Capture de l'écran de jeu

En utilisant cette méthode, lors d'un clic sur une case, on pourra changer la couleur de la case ou plutôt changer l'image, comme dans notre cas. Le joueur peut alors noircir, "blanchir" ou barrer une case.

4.5 Les jeux de test

Afin de vérifier la bonne fonctionnalité de notre programme tout au long de son développement, plusieurs jeux de tests ont été réalisés, du plus simple au plus extrême.

Plusieurs jeux de test ont été réalisés à chacune des étapes de développement de notre projet, allant de la génération des matrices à l'interface graphique. De même, lorsqu'une fonctionnalité est implémentée elle est testée avec les précédentes afin de vérifier leurs compatibilités.

Concernant la génération des matrices, une fonction a été écrite pour parcourir la matrice et afficher chacune des cases afin de s'apercevoir des di-

mensions de celles-ci ainsi que la conformité de leur contenu. Cette fonction assez simple est particulièrement utile pour la vérification des matrices prédéfinies, dont le contenu doit suivre un modèle précis sous réserve d'erroner le rendu final. Ainsi sur l'annexe 1, une des matrices prédéfinies est affichée via la fonction de parcours et produit un affichage différent selon le contenu des cases. Comme nous utilisons des matrices d'entiers, contenant uniquement des 0 et des 1 dans les grilles prédéfinies et aléatoire, nous affichons sur le terminal des "X" pour les cases contenant un 1 et des "." pour les cases contenant un 0, ceci afin d'améliorer la lecture. Cela permet de constater si la matrice prédéfinie à bien été générée et si elle est conforme à nos attentes.

Concernant les matrices aléatoires, leurs générations sont moins exigeantes puisqu'elles n'ont pas pour objectif de produire une image, mais plutôt d'éviter la redondance. Plusieurs matrices aléatoires sont alors générées et affichées sur le terminal tout en vérifiant qu'elles sont différentes les unes des autres et qu'elles soient bien équilibrées. Un résultat est visible en annexe 2, le but est, ici, de générer deux matrices aléatoires à la suite et vérifier les écarts de génération.

Ensuite, les annexes 3 à 5 présentent des jeux de test de cas de lignes (*grille_test_l1/2*) et de colonnes spécifiques (*grille_test_c1/2*) ainsi que leurs conditions correspondantes (*mat_cond_1/2*). Les matrices des grilles et des colonnes sont les mêmes, mais adaptées selon le cas.
Ces jeux de tests ont été écrits avant de vérifier le fonctionnement des fonctions d'affichage, de générations des conditions, de leur vérification pour valider la grille, et le solveur.
Ainsi, pour chaque exécution, on affiche la matrice utilisée (jeu ou conditions) et on vérifie la bonne conformité par rapport à ce qui est attendu. De ce fait, il a été possible de corriger quelques erreurs, en particulier lors de l'implémentation du solveur qui générait des erreurs de segmentation.

4.6 Exemple de débogage

Cette partie détaillera un cas de débogage réalisé au cours de la programmation du solveur.

L'annexe 6 présente un rapport d'erreur généré par l'outil valgrind.
Le message d'erreur initial est un *Segmentation fault (core dumped)* lors de l'exécution de la première version de notre solveur. Cette information unique nous permet de connaître le type d'erreur. Mais on ne peut pas déterminer s'il s'agit d'un dépassement de la taille d'un tableau lors de son parcours, l'utilisation d'un pointeur libéré ou l'appel d'un pointeur inadapté.
Exécuter le programme avec l'outil valgrind permet d'avoir plus d'informations. Ici, outre le problème de libération de la mémoire (2 frees pour 3 allocs), nous savons que le problème vient de l'appel de la fonction *copie_mat* dans

la fonction de remplissage évident *extremite_ligne*.
Toutefois, il est aussi écrit qu'une erreur de type *Stack overflow* est présente. Ces informations ont permis d'arriver à la conclusion que la récursivité du solveur était trop profonde. Ainsi, plutôt que de partir sur une méthode de résolution case par case par force-brute, il a fallu partir sur un modèle en ligne par ligne.

5 Bilan et résultats

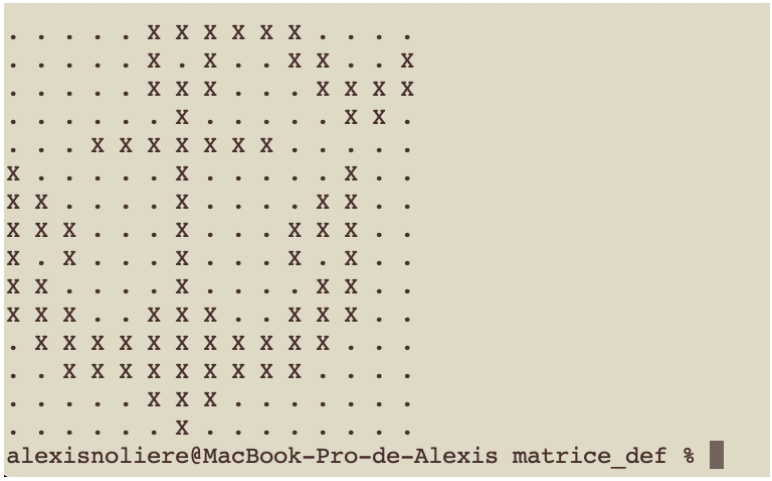
Au terme de ce projet, la majorité des fonctionnalités de base du jeu ont pu être réalisées. C'est-à-dire une interface menu simple mais efficace, un système de jeu fonctionnel ainsi que la possibilité de choisir entre différents modes de jeu et matrices pré-construites, le tout dans un rendu en pixel art. Toutefois, bien qu'il soit fonctionnel, nous n'avons pas pu finaliser la conception du solveur de façon à ce qu'il soit efficace.
Ceci est explicable par un manque de temps dû à un manque d'organisation. Malgré la mise en place de micro-planning chaque semaine, il n'a pas été possible de tous les respecter.
La première cause concerne les problèmes techniques tels que la perte de connexion ou, surtout, la difficulté à installer SDL et la bibliothèque TTH.
Viennent ensuite la communication, plus précisément les problèmes de compréhension au niveau de ce qui était souhaité et la synchronisation de nos horaires de disponibilité.
Enfin, le dernier problème concerne les fausses pistes, notamment pour le solveur. Problème nécessitant de tout reprendre depuis de nouvelles bases.

Ainsi, sous réserve de plus de temps ou d'organisation, il aurait été possible d'améliorer le solveur par l'implémentation des fonctions de remplissage évident déjà présentées. L'implémentation de telles fonctions avant l'appel récursif permettrait de traiter moins de possibilités en ne testant pas les lignes préalablement complétées.
Ensuite, l'implémentation de remplissage pour des cas plus précis, après chaque génération de possibilité pour une ligne, sur les colonnes. Un cas pourrait être la présence d'un seul bloc noir et d'au moins deux cases noircies séparées, ainsi, il est évident que les cases intermédiaires sont noires.
Plusieurs de ces fonctions ont pu être écrites, mais pas implémentées, vous pouvez les retrouver sur le dépôt Git.

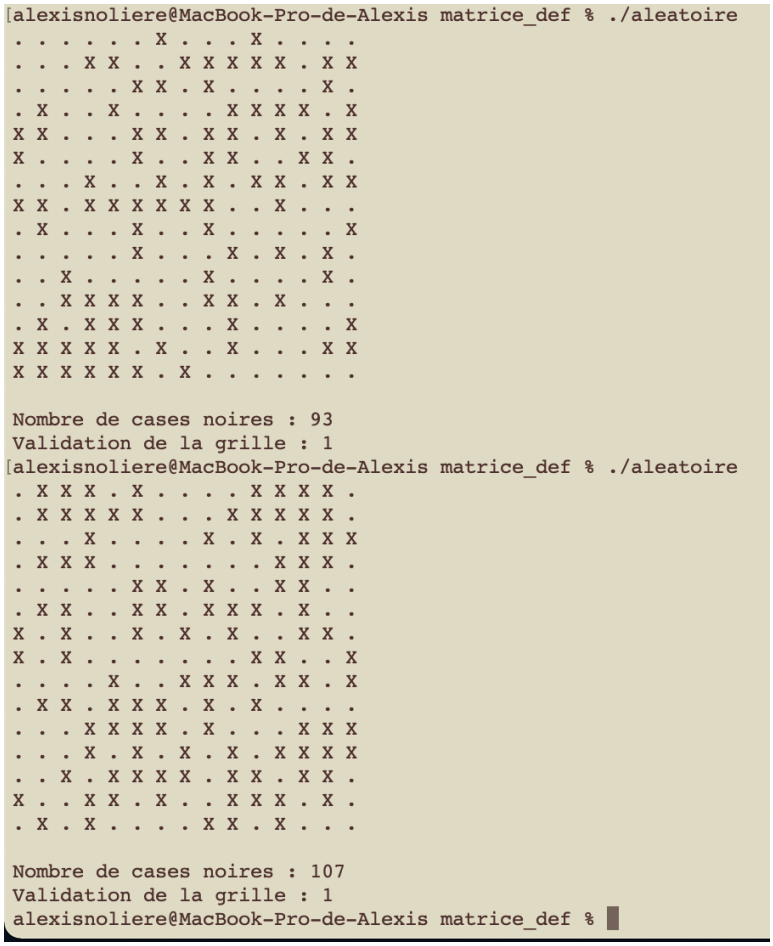
Malgré ces problèmes, cette conduite de projet nous a apporté plusieurs outils et compétences.
Tout d'abord, à travers l'apprentissage de SDL, nous possédons maintenant la maîtrise de ses bases, nous permettant alors d'avoir à disposition un outil pour la création d'application graphique. Il en va de même pour Gimp du côté de la création de rendu graphique. La manipulation de sprites étant un

bon exercice pour appréhender ses différents outils.
Ensuite, nous avons pu prendre conscience des aléas du travail de projet, qui plus est en distanciel.
Comme dit précédemment, le problème majeur fut la communication puis les soucis liés à SDL et TTF.
Il nous a donc fallu apprendre à nous organiser en conséquence. C'est-à-dire, s'entraider pour la résolution de ces problèmes et donc savoir les expliquer clairement. Mais aussi distribuer les tâches en fonction des outils disponibles à chacun et nos compétences. Car nous n'avons pas tous les mêmes facilités et difficultés dans un domaine donné.

6 Annexes



Annexe 1. Capture d'écran de la vérificaton de la génération d'une grille prédéfinie



Annexe 2. Capture d'écran de la vérificaton de la génération de deux grilles aléatoires

```
//Matrices de test
int grille_test_l1[N][N] = {{1, 0, 0, 0, 0, 0, 0, 0, 1, 1},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
                            {1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {0, 0, 1, 0, 1, 0, 0, 0, 0, 0},
                            {1, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                            {0, 0, 0, 0, 0, 0, 0, 1, 0, 1},
                            {0, 1, 0, 0, 1, 0, 1, 0, 1, 1},
                            {1, 0, 1, 0, 1, 0, 0, 0, 0, 0},
                            {0, 0, 0, 0, 0, 1, 0, 1, 0, 1},
                            {0, 0, 1, 0, 1, 0, 1, 0, 1, 0},};

int grille_test_c1[N][N] = {{1, 0, 1, 0, 1, 0, 0, 1, 0, 0},
                            {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
                            {0, 0, 0, 1, 1, 0, 0, 1, 0, 1},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {0, 0, 0, 1, 0, 0, 1, 1, 0, 1},
                            {0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
                            {0, 0, 0, 0, 0, 0, 1, 0, 0, 1},
                            {0, 0, 0, 0, 0, 1, 0, 0, 1, 0},
                            {1, 0, 0, 0, 0, 0, 1, 0, 0, 1},
                            {1, 1, 0, 0, 0, 1, 1, 0, 1, 0},};
```

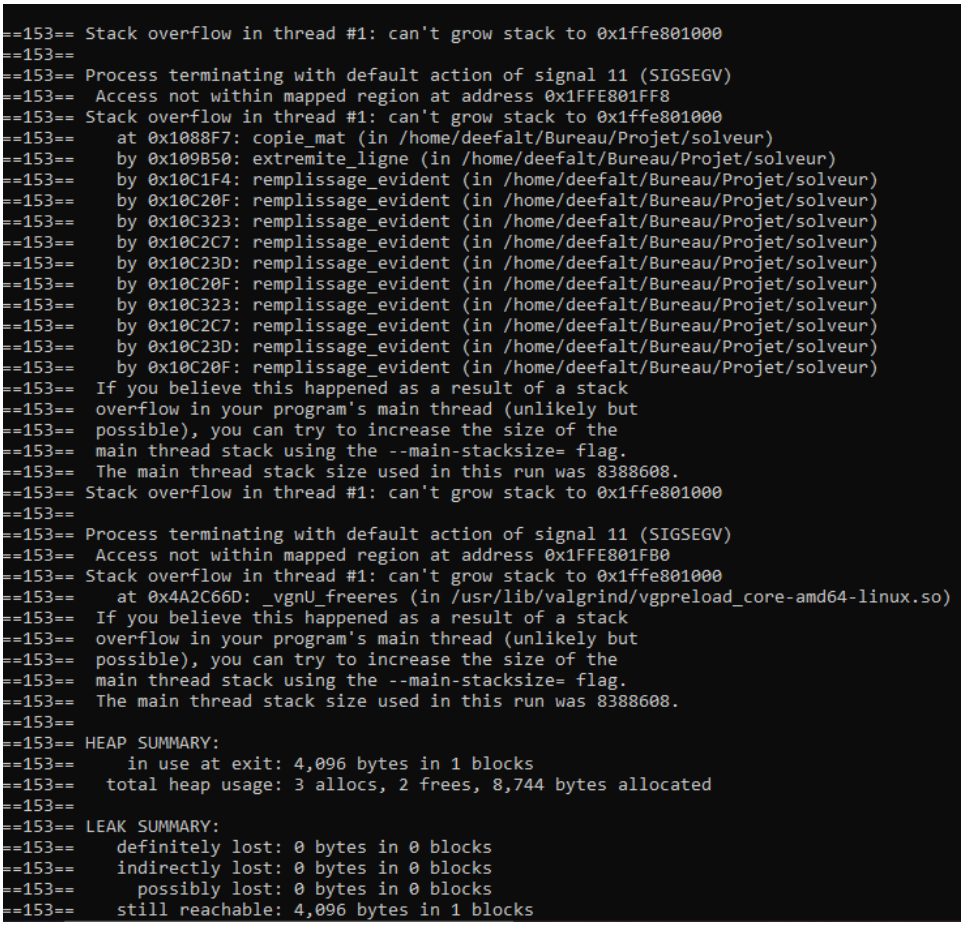
Annexe 3. Capture d'écran du premier jeu de tests spécifiques pour les lignes puis les colonnes

```
int grille_test_l2[N][N] = {{1, 0, 1, 0, 1, 0, 1, 0, 0, 0},
                            {0, 0, 0, 1, 0, 1, 0, 1, 0, 1},
                            {0, 0, 1, 0, 0, 0, 1, 0, 1, 1},
                            {1, 0, 0, 0, 1, 0, 1, 1, 0, 0},
                            {0, 0, 1, 0, 0, 0, 1, 0, 1, 1},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};

int grille_test_c2[N][N] = {{1, 0, 0, 1, 0, 0, 1, 0, 0, 0},
                            {0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
                            {1, 0, 1, 0, 1, 0, 1, 0, 0, 0},
                            {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
                            {1, 0, 0, 1, 0, 0, 1, 0, 0, 0},
                            {0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
                            {1, 0, 1, 1, 1, 0, 1, 0, 0, 0},
                            {0, 1, 0, 1, 0, 0, 1, 0, 0, 0},
                            {0, 0, 1, 0, 1, 0, 1, 0, 0, 0},
                            {0, 1, 1, 0, 1, 0, 1, 0, 0, 0}};
```

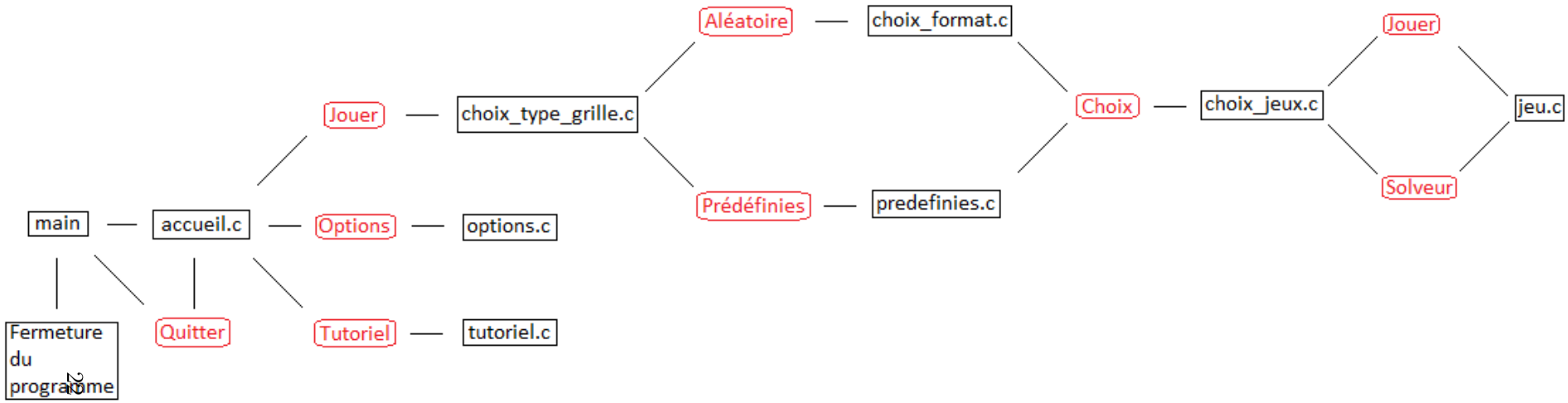
Annexe 4. Capture d'écran du deuxième jeu de tests spécifiques pour les lignes puis les colonnes

```
int mat_cond_1[N][N] =    { {3, 3, 0, 0, 0, 0, 0, 0, 0, 0},
                            {2, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {2, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {3, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {3, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {3, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {2, 3, 1, 0, 0, 0, 0, 0, 0, 0},
                            {1, 3, 0, 0, 0, 0, 0, 0, 0, 0},
                            {3, 1, 0, 0, 0, 0, 0, 0, 0, 0},
                            {3, 3, 0, 0, 0, 0, 0, 0, 0, 0}};

int mat_cond_2[N][N] =    { {3, 3, 0, 0, 0, 0, 0, 0, 0, 0},
                            {3, 3, 0, 0, 0, 0, 0, 0, 0, 0},
                            {2, 3, 1, 0, 0, 0, 0, 0, 0, 0},
                            {2, 3, 1, 0, 0, 0, 0, 0, 0, 0},
                            {2, 3, 2, 0, 0, 0, 0, 0, 0, 0},
                            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {10, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {7, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                            {2, 2, 1, 2, 0, 0, 0, 0, 0, 0},
                            {10, 0, 0, 0, 0, 0, 0, 0, 0, 0}};
```

Annexe 5. Capture d'écran des matrices conditions correspondantes



Annexe 6. Capture d'écran d'un rapport d'erreur généré par l'outil valgrind



Annexe 7. Organigramme des menus, les fichiers en noir et les boutons en rouge

GANTT CHART TEMPLATE

Smartsheet Tip ➜ A Gantt chart's visual timeline allows you to see details about each task as well as project dependencies.

PROJECT TITLE	Picross	**COMPANY NAME**	TD B Groupe 4
PROJECT MANAGERS	MAHOUIN J., NOLIERE A., Mohamad Jamil G., LEMRABOT	**DATE**	22/01/2021

WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	**Project Conception and Initiation**					
0	Création du Gantt Chart	Tout le monde	01/22/21	2/5/21	2	100%
1.1	Génération de grilles aléatoires	Gallup, Mohamed, Julien	2/18/21	2/18/21	1	100%
1.1.a	Affectation aléatoire des cases noires	Gallup, Mohamed, Julien	2/12/21	2/12/21	1	100%
1.1.b	Validation de la grille	Julien, Mohamed	2/12/21	2/12/21	2	100%
1.2	Générations des conditions	Julien	02/05/21	2/12/21	2	100%
1.2.a	Validation des conditions	Julien	02/05/21	2/12/21	2	100%
1.3	Génération de grilles prédéfinis	Alexis	01/22/21	2/8/21	16	100%
1.3.a	Fichier contenant les matrices prédéfinis	Alexis	01/22/21	2/8/21	16	100%
1.4	Affichage de la grille et des conditions	Julien, Alexis, Mohammed	2/18/21	2/18/21	1	100%
1.5	Réalisation d'un solveur	Alexis, Julien, Mohammed	2/8/21	3/1/21	23	90%
1.5.1	Vérification des contradictions	Alexis, Julien	2/25/21	2/25/21	1	100%
1.5.2	Fonctions de remplissage évident	Alexis, Julien, Mohammed	2/25/21	3/5/21	10	80%
1.5.3	Fonction récursive	Alexis, Julien, Mohammed	2/25/21	3/10/21	15	90%
1.6	Fichier de sauvegarde	Alexis	2/8/21	2/18/21	10	100%
1.7	Système de jeu	Gallup, Julien	3/1/21	3/22/21	21	100%
1.7.1	Noircir une case	Gallup	3/1/21	3/22/21	21	100%
1.7.2	Blanchir une case	Gallup	3/1/21	3/22/21	21	100%
1.7.3	Barrer une case	Julien	3/1/21	3/22/21	21	100%
1.8	Interface	Julien, Alexis	3/22/21	4/18/21	26	100%
1.8.1	Sprites	Julien, Alexis	3/22/21	4/18/21	26	100%
1.8.2	Menus	Julien	3/22/21	4/18/21	26	100%
1.9	Test unitaire/integration	Tout le monde	01/22/21	4/22/21	90	100%
2	**Project Definition and Planning**					
2.1	Mise en place/formation Git	Tout le monde	2/5/21	2/5/21	1	100%
2.2	Formation SDL	Tout le monde	2/22/21	3/1/21	9	100%
2.3	Compte-rendu	Tout le monde	4/2/21	4/18/21	16	100%
2.4	Soutenance Orale	Tout le monde	4/6/21	4/18/21	12	100%
2.5	Document Doxygen	Tout le monde	4/10/21	4/19/21	9	100%
3	**Bonus**					
3.1	Timer	Gallup	3/22/21	4/12/21	20	90%
3.2	Objets d'aide					0%
3.2.1	Flèche					0%
3.2.2	Bombe					0%
3.2.3	Sabre					0%
3.2.4	Shurikens					0%
3.3	Appel du solveur via bouton					0%
3.4	Système de gain d'xp / niveau					0%
3.5	Surbrillance des lignes/colonnes à la sélection d'une case					0%
4	**Project Performance / Monitoring**					
4.1	Objectifs du projet					85%
4.2	Performance du programme					80%
4.3	Qualité					90%