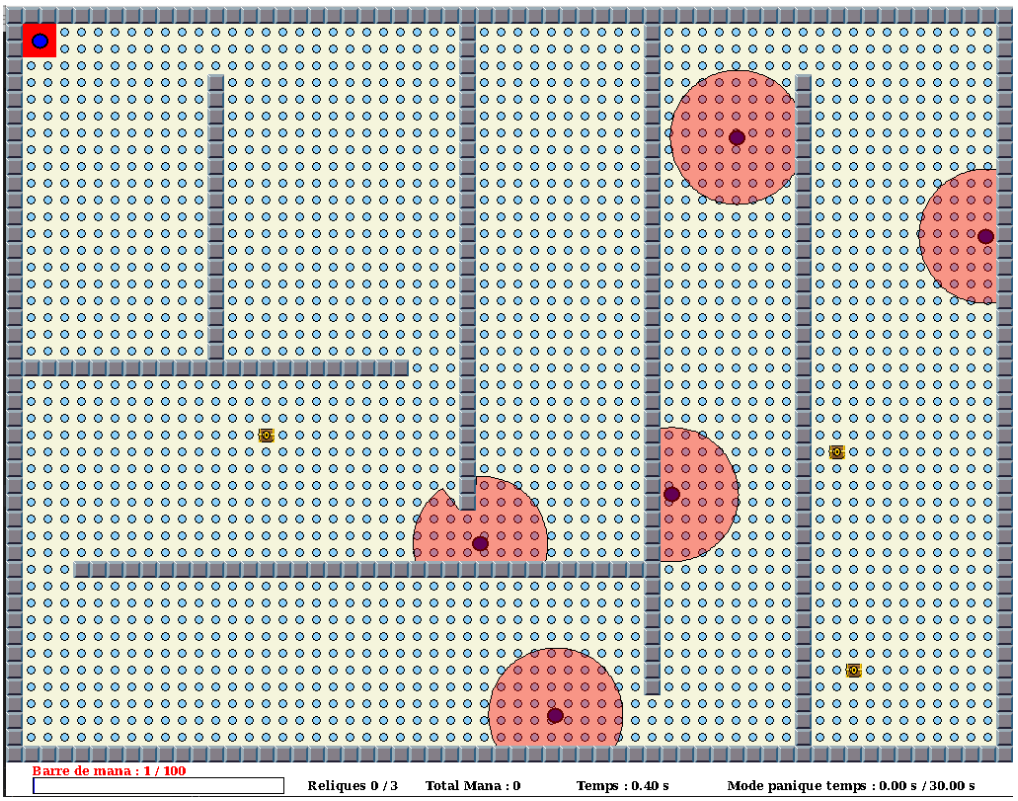
Hakim AOUDIA
Ilhan ACHERCHOUR

Semestre 5
Licence 3 informatique

Université Gustave Eiffel

Projet de programmation C : Jeu d'infiltration



Introduction

Lors de ce semestre nous devions programmer un jeu d'infiltration en 2D en C. Pour ce projet nous avions des indications générales dans le sujet notamment le pour diviser ce projet en sous partie par exemple une partie Terrain, Gardiens, Joueur etc. Ce projet d'apparence plutôt simple comporte de nombreux points qui nécessitent une réflexion avant de commencer à programmer, on peut penser à la génération de terrain par exemple ou la collision des gardiens sur un mur.
Les nouveautés dans ce projet c'est l'utilisation de **git** par les membres du binôme pour s'organiser. Cette partie nécessite d'autres explications donc nous en reparlerons de git plus tard dans le rapport.

Le projet comporte de nombreux modules à implémenter nous avons d'abord lu le sujet et tenté de diviser ce le projet en plusieurs sous parties pour faciliter la programmation. Le makefile découle des divisions en sous partie du projet. Durant toute la durée de développement du projet nous avons modifié, supprimé et/ou ajouté des sous parties. Le projet se complexifié de plus en plus et à certains moments nous n'avons pas prévu le bon module. Mais cela n'est rien car il suffit de rajouter des modules (.c et .h) et modifier le makefile.

Génération de terrain

1 - Initialisation du terrain

On a d'abord commencé par générer le terrain car c'est la partie essentielle du jeu. On a alors décidé de créer un tableau à deux dimensions d'entier. Comme le plateau comportera différent type de case, on a décidé de créer un **Enum objet** qui possédera tous les types de cases possibles.

On souhaite expliquer quelque objet de notre Enum ici. Par exemple, le enum objet **DOOR** sert lorsqu'on génère des gardiens et des reliques pour éviter de les placer dans les ouvertures des murs.


Après la subdivision, les gardiens et les reliques sont placés aléatoirement à l'intérieur des compartiments, donc pas sur les ouvertures, et aussi à une distance euclidienne d'au

Enum objet **NO_RELIC** représente une case lorsque le joueur à récupérer une relique sur le sol et qu' aucun des gardiens n'a encore détecté son vol.
Tandis **DETECTED_NO_RELIC** indique que la relique a déjà été volée et qu'un gardien l'a déjà remarqué dans le passé.

2 - Algorithme de génération du terrain

L'algorithme donné pour la génération de mur est plutôt clair. Nous avons seulement un problème au début du développement de l'algorithme car nous avions mal créé le prototype de la fonction. Au lieu d'utiliser deux variables x et y pour indiquer de quel point

jusqu'à quel autre point le terrain s'implémente on avait utilisé un point et une largeur, longueur. Ce qui complexifie la fonction pour rien. On a corrigé cela.

```
/* Version début */
void generation_mur(int** terrain, int pos_x, int pos_y, int taille_x, int
taille_y);
/* Version intermédaire */ void generation_wall(int** terrain, int x1, int
y1, int x2, int y2);
/* Version final */ void generation_wall(int** terrain, int x1, int y1,
int x2, int y2, int minside);
```

Dans la version finale on a ajouté une variable **minside** car elle peut être modifiée selon l'utilisateur dans le menu paramètre.

Déplacement du joueur

1 - Déplacement ne respectant pas la consigne

Le joueur peut se déplacer horizontalement, verticalement et aussi les deux au même temps. Ce qui nous pose problèmes dans l'implémentation du mouvement c'est cette phrase de la consigne. Cela nous a mis un doute et on a refait le déplacement du joueur.

Cependant, quand le personnage démarre ou change de direction, sa vitesse initiale commence à $0{,}1v$, et accélère de $0{,}03v$ par frame (ou une autre valeur qui vous semble appropriée) où la touche de direction reste enfoncée.

Quand le joueur change de direction, nous devions changer sa vitesse à celle initiale. Il fallait donc détecter le cas où par exemple il passe de la direction DROITE a la direction DROITE + HAUT.

2 - Résolution du problème avec deux enum Direction

Pour cela on a créé deux Enum **Direction_Horizontal** et **Direction_Verticale** cela nous permet de connaître la direction précédente et vérifier en permanence si le joueur change de direction même dans un seul des axes. Ce qu'un Enum Direction (englobant les 4 directions possibles) seul ne peut pas faire.
Cela a aussi eu pour effet de corriger une lenteur lors des mouvements en diagonale car avant lorsque le joueur bougeait en diagonal il gardait constamment sa vitesse initiale car le programme détecter un changement de direction en permanence.

Détecter la collision des murs

1 - Comprendre les formules et implémenter

Le principe est assez clair mais il fallait bien lire les consignes. Pour être sûr de comprendre on a pris un cas simple et on a essayé de le faire sur papier nous même pour déterminer à quel moment le maximum est atteint dans une certaine position du joueur.

2 - Bug

Bien qu'on ait compris comment utiliser les formules on avait fait une erreur simple mais qui nous aura fait perdre **beaucoup** de temps c'est qu'on a écrit dans notre formule 1 / 4. Au lieu d'utiliser la taille d'une case et donc d'un joueur/gardien. Cela faisait des collisions mais pas précises on a beaucoup chercher pour se rendre compte qu'on avait juste mal implémenter la formule

Collecter et replacer la mana

1 - Première étape, comprendre le sujet

Dans le sujet il y a une phrase qu'on a trouvé peu claire c'est celle-ci.

> Le mana dépensé retourne sur des tuiles choisies aléatoirement et dont les traces de mana avaient été absorbées. Les compétences ne peuvent pas être utilisées si le personnage n'a pas assez de mana.

On ne savait pas si il fallait la comprendre de la façon suivante :

- On choisit une tuile aléatoirement dans la totalité du terrain et on met de la mana.

Ou alors:

- On choisit une tuile aléatoirement parmi les cases déjà vide sur le terrain et on met de la mana.

Comme cette fonctionnalité de mana n'est pas primordiale pour le bon déroulement du jeu, on a d'abord implémenter un placement de la mana qui prend un case aléatoire et place de la mana dessus. On c'était dit que c'était trop tôt d'implémenter une liste chaînée seulement pour la mana.
Mais dans le rendu final on a une liste chaînée qui récupère les cases sur lesquelles le joueur marche. On stocke ces cases et quand le joueur utilise son pouvoir cela choisit une case aléatoire **parmi** les cases de la liste chainées et donc les cases déjà vide.

Note : Le pouvoir de rapidité consomme 2 de mana mais la fonction qui place la mana sur le terrain ne pose que 1 case de mana sur le terrain nous au courant mais on a pas jugé que ce soit un problème pour le bon fonctionnement du jeu et donc laisser cela.

2 - Pourquoi une liste chaînée ?

Concernant le placement de la mana on avait 2 problématiques, stocker les cases vides dans une liste sans connaître la taille (on aurait pu créer une liste de Case de taille 100) mais surtout la suppression à tout endroit de la liste. Pour cela on a cherché dans le cours et trouvé cela.

- Suppression à tout endroit, parcours dans un sens : liste simplement chaînée

Cours : Bibliothèques, et comment écrire du bon code

Compétence du joueur

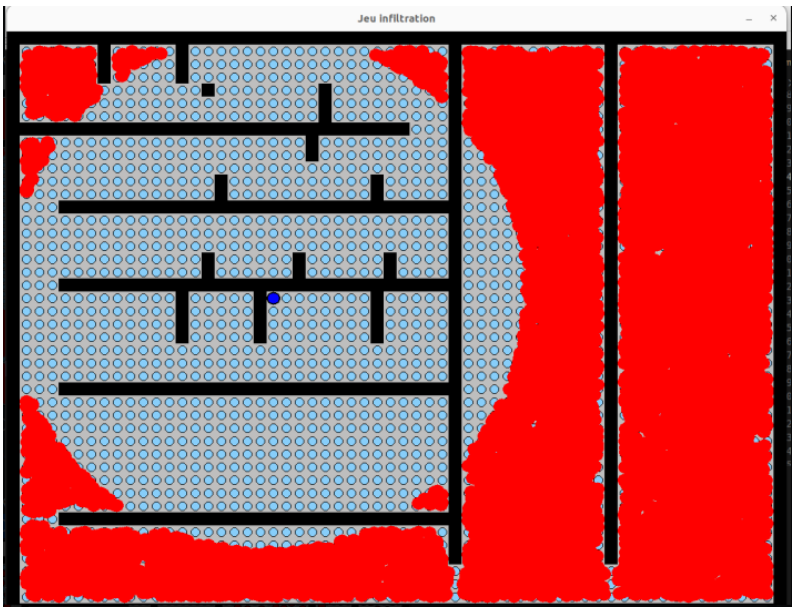
1 - Active le pouvoir

Pour le choix de pouvoir on a fait une structure Power qui contient deux int qu'on utilise comme booléens. On aurait pu faire un Enum avec invisible et speed mais ça ne nous permettait pas d'utiliser les deux pouvoirs en même temps. Donc on a opté pour la première solution.

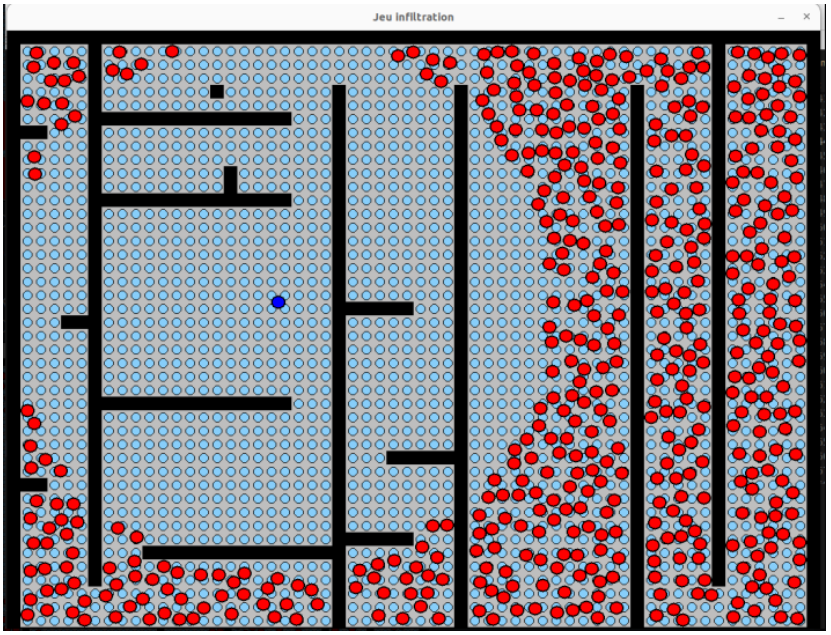
Placer gardiens et reliques sur le terrain

Dans le sujet il était écrit qu'on devait placer les gardiens à une distance de 20 cases du joueur. Pour cela on calcule simplement la distance euclidienne entre la position du gardien et du joueur. Par contre, il faut pouvoir éviter de faire apparaître un gardien sur un mur, ou en contacte avec un mur et même sur une ouverture de porte (qu'on a géré avec le enum DOOR précédemment expliqué).

Pour cela on a créé une fonction **has_wall_around()** et aussi on ajouté une fonction **has_guardian_around()** même si ce n'était pas demandé qui permet de poser les gardiens a une certaines distance entre eux. Pour éviter qu'il y ait des gardiens au même endroit.



9000 gardiens posés à une distance de 20 cases du joueur et évite les murs.



500 Gardiens toujours à une distance de 20 cases du joueur mais avec une certaine distance entre eux pour éviter une superposition.
L'implémentation pour éviter de mettre un gardien (lorsque son "corps" dépasse) sur une porte n'est pas présente dans ce screen qui est un ancienne version

Le placement de reliques est similaire aux placements de gardiens. La seule différence est que les reliques sont placées aléatoirement sur le terrain sans minimum de case entre elles.

La détection du joueur par les gardiens

1 - Premières idées

Lorsque le gardien a une vision à 360 degré on se dit qu'il suffit de calculer la distance entre le gardien et le joueur et si cette distance est inférieur ou égal au rayon du cercle qui représente le champ de vision du gardien alors le joueur est détecté.

On voit bien que cette méthode ne suffit pas lorsque le joueur est caché derrière un mur.

2 - Lecture de l'énoncé et première implémentation

On a trouvé l'explication de l'énoncé assez compliqué quand on a vu les formules mathématiques. Au final c'est devenu plus clair en implémentant ces formules. Au début on avait tracé plusieurs points régulièrement sur le segment qui sépare le joueur du gardien et on regardait si parmi ces points, un seul d'entre eux était sur un mur. On savait que ce n'était pas ce que demandait l'énoncé mais on voulait déjà comprendre les choses de manière simple

3 - Quelques problèmes

Ensuite on a décidé de suivre l'énoncé. Le problème c'est que l'énoncé nous montre comment nous déplacer en x mais pas en y. On s'est vite rendu compte que ça revenait au même mais cela nous a un peu ralentis.

On a aussi eu du mal à savoir si les formules fonctionnaient peu importe l'emplacement du joueur par rapport à celui du gardien.

Au final on a testé plusieurs positions et il y a avait quelques erreurs mais on a compris que la valeur de x0 et x1 étaient importantes car inverser les deux pouvait tout changer dans les calculs.

4 - Etape finale

Une fois ces quelques problèmes réglés, qui nous ont permis de mieux appréhender les formules. Nous avons réussi à tracer tous les points d'intersections du segment reliant le joueur aux gardiens. Après quelques ajustements on a réussi à trouver les bonnes cases associées aux points d'intersections et nous avons fini par réussir à savoir si un mur séparait ou non le joueur d'un gardien.

Tableau de classement

1 - Quelles informations stockées ?

Cette partie n'était pas particulièrement difficile, il fallait au début bien lire et comprendre ce qu'on attendait de nous. On a compris qu'il fallait stocker un temps, un nombre de mana et un pseudo. Cela nous faisait alors une structure qu'on a nommé **Score.** On a donné un maximum de caractère au pseudo pour simplifier sa représentation sur l'écran de fin et aussi pour simplifier son utilisation.

2 - Comment récupérer le pseudo ?

Pour récupérer le pseudo du joueur il fallait un peu d'astuce car ML possède des fonctions permettant de recueillir les touches du joueur. Mais il faut convertir ses touches en String qu'on convertis ensuite en char pour ensuite la stocker dans une variable char* pseudo. Une fois le pseudo écrit on pourra alors l'écrire dans un fichier binaire. C'est la première fois qu'on utilisait un fichier binaire dans un projet on a donc dû se documenter car nos habitudes d'utiliser fprintf() pour écrire dans un fichier a été notre premier réflexe. Mais en utilisant fwrite() avec notre structure Score cela écrit bien en binaire comme on veut.

Pour aller plus loin

Nous avons ajouté tout un tas d'options à notre jeu, notamment un champ de vision réaliste pour les gardiens(dragon).

Paladin - Détecte n'importe quel point du joueur

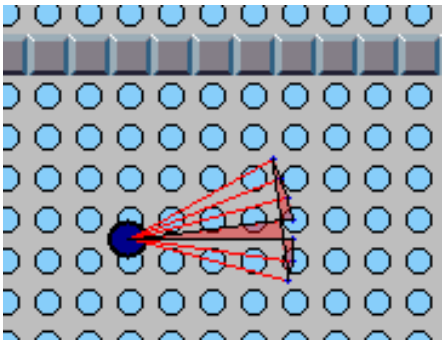
Pour cela c'était assez simple, au lieu de comparer si un gardien détecte un joueur en son centre il fallait comparer avec tous les points sur les bords du joueur. En utilisant la fonction **create_points()** qu'oOn avait déjà utilisé cette fonction dans un précédent projet il suffisait de la modifier pour qu'elle fonctionne en C.
cela crée de nombreux points autour d'un point.

Dragon 2 - Rayon de détection

Une fois la détection du joueur implémenter nous avions de quoi déformé le champ de vision des gardiens. Nous nous sommes empressés de le faire cependant beaucoup d'erreurs se sont présentées. On pensait alors que c'était parce que nous n'étions pas encore prêts à travailler sur les options du jeu, que c'était trop difficile pour nous.

Finalement, on a compris que la détection du joueur n'était pas totalement aboutie. Nous avons réussi à trouver les points d'intersections mais c'était plus compliqué pour les cases du plateaux associé à ces points. Nous avons donc fait une refonte de ce travail et sommes revenus sur le dragon et on a finalement réussi à donner aux gardiens une vue réaliste.

Dragon 1 - Champs de vision limité (Pas implémentés)



On a essayé d'implémenter le premier dragon. La création des points s'est correctement déroulée. Il suffisait de modifier la fonction avec une condition et de rajouter en paramètre une **direction** et un **angle**.
Le problème qu'on a rencontré c'est de dessiner le polygon correctement. En fait lorsque la (direction + angle) > 360 alors l'affichage du polygon avec MLV ne se passait pas correctement.

Le bug lorsque la (direction + angle) dépasse 360

On a des pistes d'où viennent les problèmes mais cela prend trop de temps pour une option. Donc on a laissé cela. Avec un peu plus de temps, on pense réussir et pourquoi pas le finir plus tard. Pour implémenter les deux dragons il aurait aussi fallu créer une fonction **is_in_polygon()** pour détecter si le joueur se trouve entre les points du gardiens et le gardien lui-même et ça aurait été bon.

Mode pause

On a ajouté un mode pause dans le pause. Pour l'activer, il appuie sur la touche Echap. Cette option n'est pas demandée mais on a trouvé cela assez fun à implémenter et c'est très pratique.

Sons, musique de jeu et du panic mode

Lorsque le joueur clique sur des boutons ou qu'il se passe certains événements des sons expliquant les événements sont lancés. cette option a était la plus fun a ajouté en plus cela crée un effet de "vrai" jeu. On peut parler d'un problème lors de cette implémentation est la gestion du **MLV_free_music_()**. On avait pas bien compris comment cela fonctionne lorsqu' on utilisait plusieurs musiques. On a finalement réussi à libérer la mémoire lors de la fin du jeu pour les musiques.

De plus les sons provoquent un arrêt du jeu tant qu'il ne sont pas terminés, on a cherché comment appeler un son pendant que le reste du programme est lancé mais on a pas trouvé. Pour régler ce problème, on attend que le son finisse pour continuer dans le programme. Cela ne dérange pas le déroulement du jeu car les sons sont courts donc ce n'est pas grave.

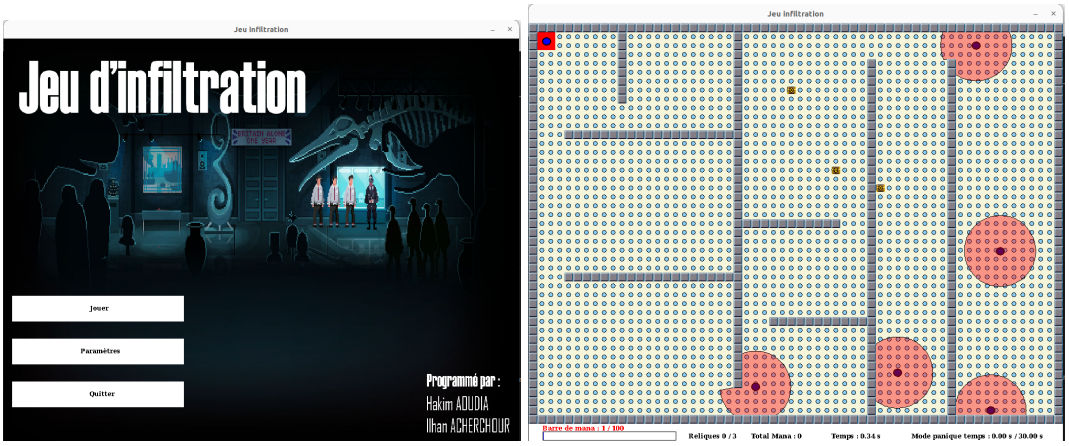
Menu paramètre

Un menu paramètre qui permet de modifier plusieurs paramètres. Cela était fastidieux à implémenter car il fallait gérer les cliques de la souris correctement. Et la fonction est plutôt longue, on ne voit pas trop comment simplifier la fonction. Cependant elle fonctionne correctement, les clics de souris sont précis.

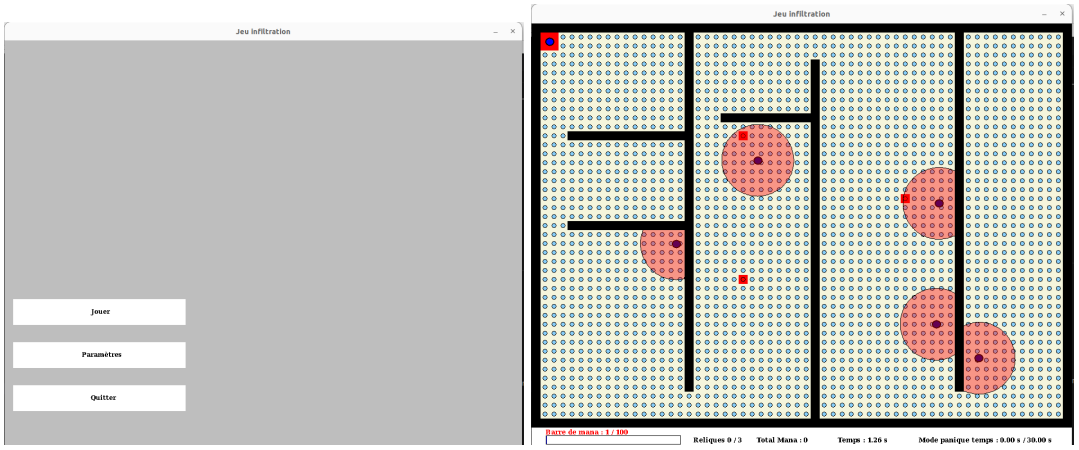
On a aussi ajouté une fonction **clamp_setting()** qui permet de mettre des valeurs minimales et maximales dans chaque option.

Images, sons

Comme les images et les sons ne sont pas obligatoires au bon déroulement du jeu, on a testé si leurs valeurs était nul à chaque fois avant leur utilisation. Cela permet au jeu d'être jouable même si le répertoire /img ou /sound venait à disparaître ou être inaccessible.



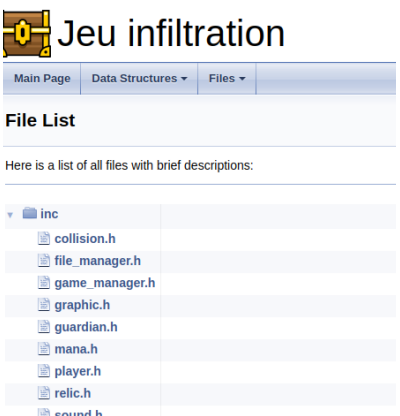
Version avec images (et sons)



Version sans images (et sons)

On précise cela car précisément on renvoyer un message d'erreur pour indiquer qu'il manquait un fichier. Cependant ce n'était pas pertinent étant donné que les images et les sons sont non essentiels pour jouer. Voilà pourquoi on fait des tests dorénavant avant d'utiliser un fichier extérieur sans stopper le programme.

Découverte de Doxygen



C'est bien la première fois qu'on utilise Doxygen. Et on a compris son importance. En effet un projet avec des fonctions documentées peut suffir dans certains cas mais

dans notre cas par exemple ou il y commence a y avoir de **nombreux fichiers** et ainsi de nombreuses structures, enum cela est pratique pour mettre en ordre tout cela.

Pour utiliser doxygen on a documenter votre code d'une façon précise avec des **mots clés**. Par exemple :

- @brief : Description de la fonction
- @param : Paramètre de la fonction
- @return : La valeur de retour de la fonction

Cela a pour effet de faciliter la création de documentation et donc de lisibilité du projet pour des lecteurs extérieurs au projet ou même pour nous-mêmes lorsqu' on voudra relire ce qu'on a fait.

Utilisation de git

L'utilisation de git dans un projet et même en général a était une première pour nous et cela n'a pas été simple de s'habituer. On a pas encore cette habitude de faire un commit a chaque fois qu'on fini de programmer.

De plus on a dû d'abord connaître les commandes essentielles comme git clone, add ., commit -m, push et d'autres. On n'a pas réellement trouvé d'intérêt dans ce projet car nous programmons certes chacun de notre côté mais aussi en vocal sur discord ce qui nous permet de s'aider mutuellement directement.

Le makefile

1 - ansi, Wall, pedantic

Au début du projet on a voulu utiliser les flags **-ansi -Wall -pedantics** pour voir les warnings qu'ils nous donnent mais aucun warnings ne s'est affiché. On s'est longtemps demandé pourquoi alors qu'on savait qu'il y avait des warnings qui aurait pu apparaître par exemple pour des commentaires avec //. Au bout du compte on a remarqué que c'était seulement dans notre makefile ou on avait mal écrit le flags ce qui a eu pour effet de ne pas activé les flags et donc de donner de warnings. On a réglé nos warning -ansi -Wall -pedantics finalement une fois qu'on a trouvé le problème.

2 - Fuite mémoire

On a aussi utilisé le flag **-fsanitize=address** qui permet de détecter d'éventuelles fuite mémoire. Car en effet on avait oublié lors de la conception du projet de désalloué les mallocs créés comme pour la liste à deux dimensions du terrain ou une liste de points par exemple. Finalement on a géré cela grâce aux warnings du flags précédemment dit.

Ce qu’on aurait voulu faire

1 - Vitesse proportionnelle

On a voulu rendre la vitesse du joueur et des gardiens proportionnelle à la taille de la fenêtre mais cela signifie changer la constante **#define SPEED** en une variable mais cela nécessitait une refonte du projet et on ne trouvait pas que le temps passé à réimplanter valait la peine de cet ajout.

2 - Mode pleine écran

Le mode pleine écran était relativement simple à implémenter grâce à des fonctions MLV mais cela ne fonctionnait que sur 1 des ordis de notre binôme. On a pas su corriger le problème et on a alors pas fait ce mode.

3 - Boutons

On a aussi quelques boutons qui ne s’affiche pas bien en fonction de la taille des cases.

4 - Simplifier certains fonctions et programme

Il y a cette fonction qui dépasse les 20 lignes, on est au courant de cela et ça a été une problématique durant tout le projet. On a essayé le plus possible de respecter la règle “une fonction, un problème gérer” mais cela reste assez difficile dans certains cas. Notre programme **main.c** est aussi long.
On aurait pu mettre certaines parties du code dans le game manager pour diviser ce programme mais on a trouvé que cela est étrange de mettre des fonctions d’affichages, de sons, de calcul dans le game manager. Cela aurait servi comme un fourre tout sans logique. Au moins le main.c est le seul à posséder des fonctions d’affichages et de sons par exemple. Cela évite que le game manger doit s’en occuper. mais cela a un prix et c’est la qualité du code du **main.c** qui en pâtit.

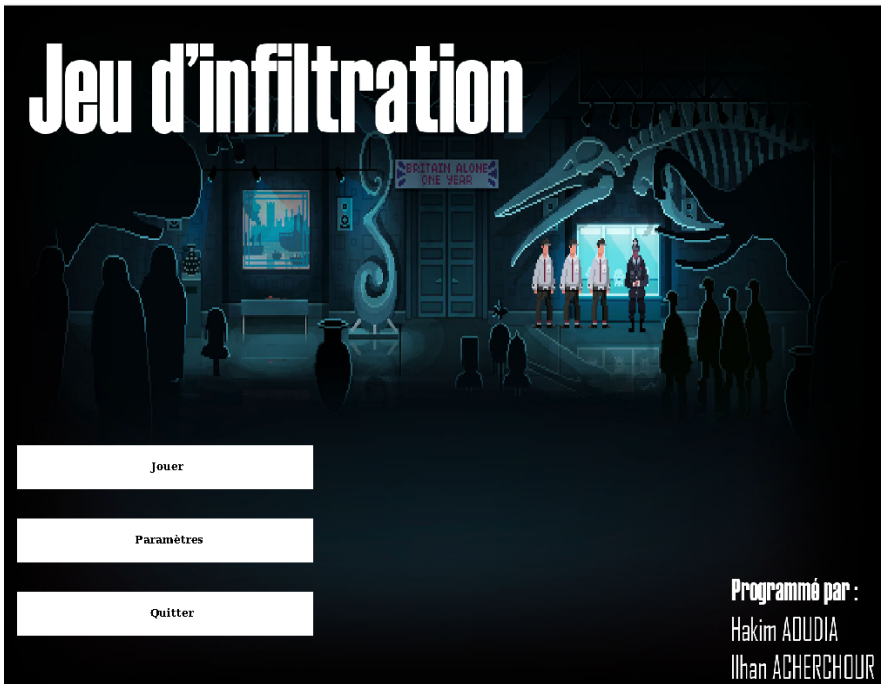
Au début dans le file manage on avait utilisé fprintf() on s'est rendu compte qu’il fallait plutôt utilisé fwrite et crée une structure Score pour simplifier l’écriture et la lecture

Guide utilisateur

1 - Début de partie

Une fois le programme lancé le jeu commence par l'écran d'accueil où nous sont proposé 3 boutons :

- **Jouer**
- **Paramètres**
- **Quitter**



a - Jouer

Le personnage du joueur commence en haut à gauche du terrain. On y voit les gardiens qui rodes et les reliques qui sont disposé un peu partout dans les pièces

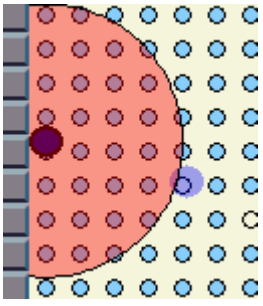
Pour **bouger** le personnage :

- **Z** pour **monter**
- **S** pour **descendre**
- **Q** pour **aller à gauche**
- **D** pour **aller à droite**

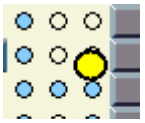
On peut aussi se déplacer diagonalement.

Pour utiliser les **pouvoirs** :

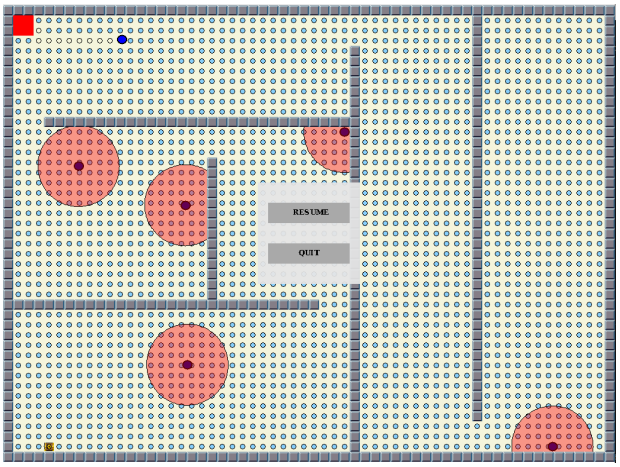
- Touche **ESPACE** pour rendre le personnage invisible



- Touche **SHIFT** pour déclencher le pouvoir d'accélération



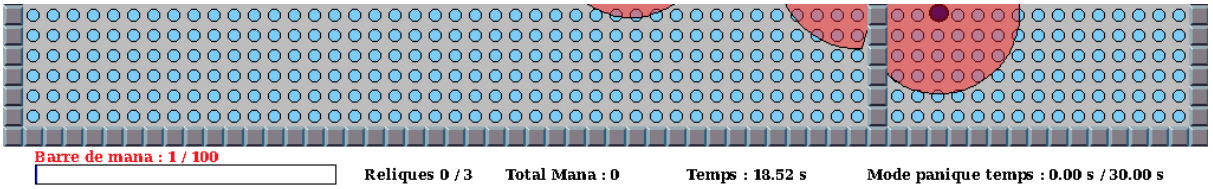
On peut mettre le jeu **en pause** en appuyant sur la touche **ECHAP**.



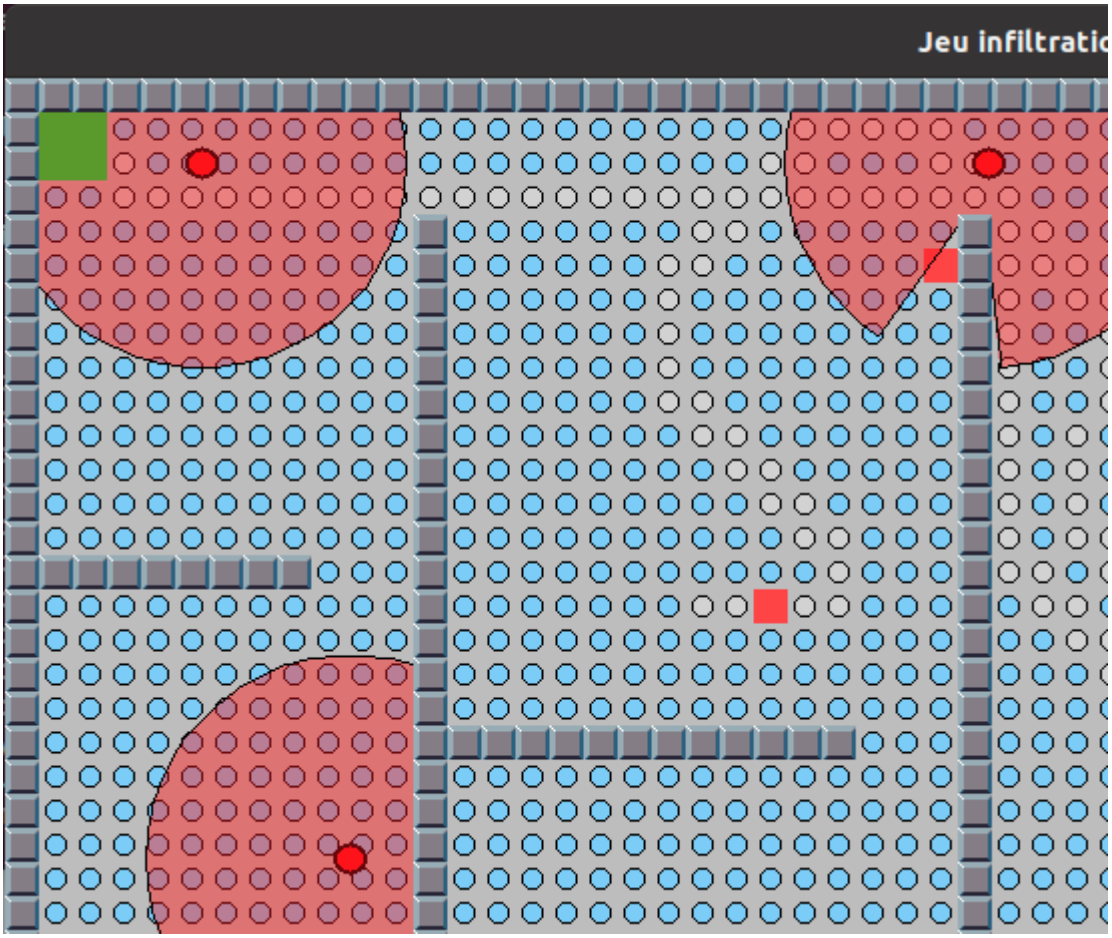
L'utilisateur peut ensuite utiliser les clics gauche de la **SOURIS** pour choisir soit **continuer** soit **quitter**.

Et puis **en bas de l'écran** on voit (en lisant de gauche à droite) :

- la barre de mana
- le nombre de reliques volé / le nombre total de relique sur le terrain
- le nombre de mana utilisé par le joueur
- le temps écoulé de la partie
- la durée du mode panique des gardiens
-



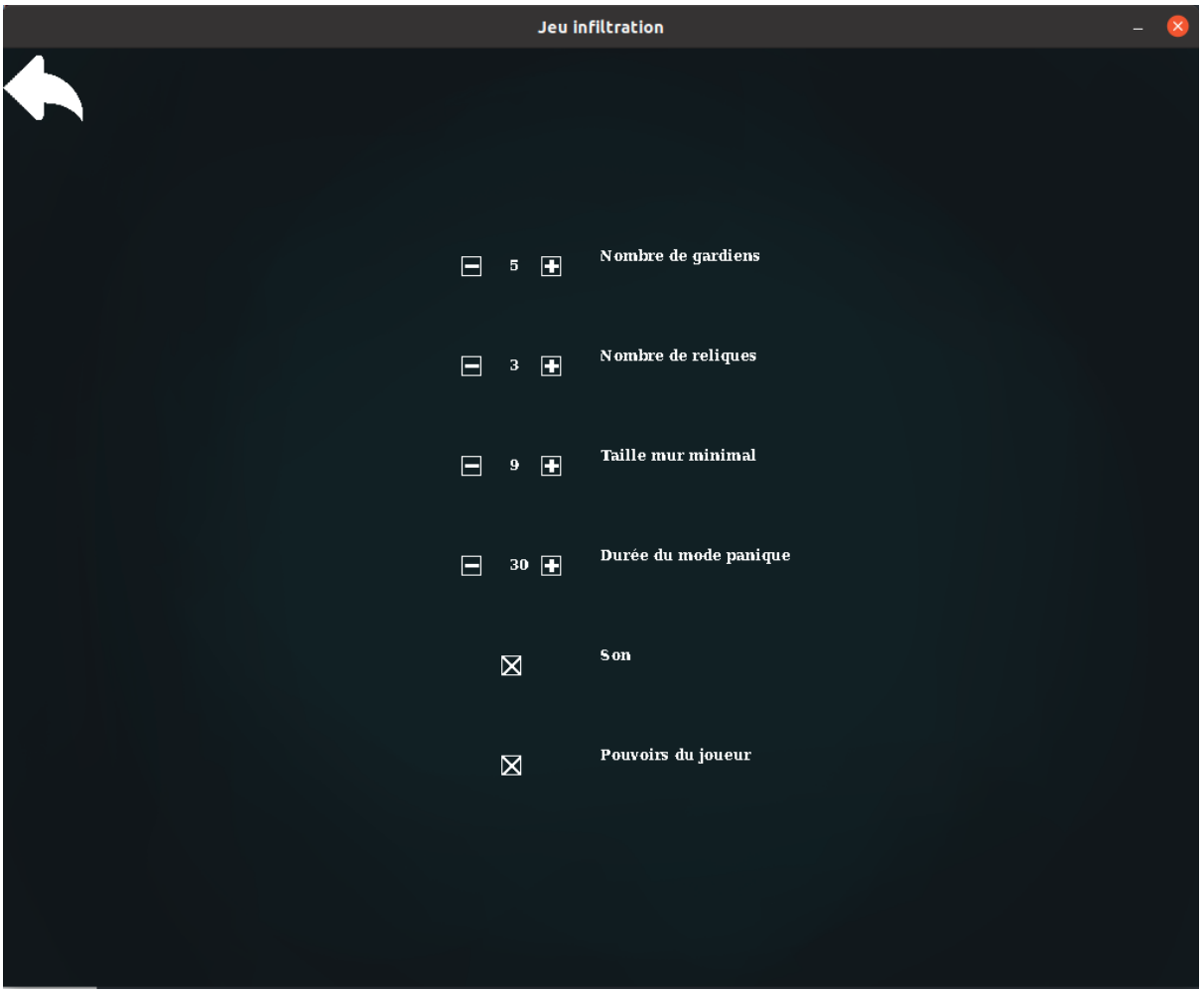
Une fois que le personnage à volé toute les reliques il doit revenir au coin en haut à gauche là où il a commencé la partie. On remarque que le carré rouge du point de départ devient vert une fois que toute les reliques ont été volé et donc que le joueur peut finir la partis une fois qu'il y sera retourné



b - Paramètres

On peut paramétrer plusieurs variables :

- Le nombre de gardiens (entre 0 - 15)
- Le nombre de reliques (entre 1 - 20)
- La taille du plus petit mur (entre 4 - 15)
- La durée du mode panique (entre 3 - 60)
- Activer ou non le son
- permettre au joueur ou non d'avoir des pouvoirs



2 - Fin de partie

A la fin de la partie le score du joueur est affiché c'est-à-dire la mana utilisée et le temps écoulé de la partie. Si le joueur gagne, il a la possibilité d'inscrire son nom dans la

table des scores. Enfin on peut choisir entre revenir à l'écran d'accueil ou quitter le programme.