

01/04/2023

Rapport de projet

INFO0601 – INFO0604



DARVILLE Killian, Sinet Théo

UNIVERSITE DE REIMS CHAMPAGNE-ARDENNE

Sommaire

| | |
|------------------------------------------------------|----|
| Introduction..... | 2 |
| I- Éditeur | 3 |
| Interface de l'éditeur..... | 3 |
| Outils | 3 |
| Structures | 5 |
| Fichier binaire..... | 6 |
| Table d'adressage et table de vide..... | 6 |
| Map | 7 |
| Fonction..... | 8 |
| Lancement de l'éditeur | 10 |
| II- Client..... | 11 |
| Lancement du client..... | 11 |
| Interface | 12 |
| Règle..... | 13 |
| Structures | 16 |
| Fonctions | 16 |
| III- Serveur | 18 |
| Lancement du serveur..... | 18 |
| Structure..... | 18 |
| Fonctions et Threads..... | 20 |
| IV- Communication entre le client et le serveur..... | 24 |
| Structures de communication | 24 |
| Création d'une partie | 24 |
| Rejoindre une partie..... | 24 |
| Les différents échanges..... | 25 |
| Le protocole UDP..... | 25 |
| Le protocole TCP..... | 27 |
| V- Remarques sur le projet | 28 |
| Difficultés..... | 28 |
| Améliorations | 28 |

Introduction

Le projet commun d'INFO0601 et INFO0604 consiste à créer un jeu multi-joueurs en réseau et multi-threadé ! Le but de ce projet est de concevoir un jeu de plateformes passionnant dans lequel chaque joueur contrôle un personnage et le déplace dans un monde rempli de défis et d'obstacles. Le jeu se déroule dans plusieurs niveaux et l'objectif est d'atteindre la sortie avant les autres joueurs. Pour cela, il faudra récupérer des clés pour passer les portails, éviter les ennemis, les pièges et les bombes des adversaires. Ce projet promet des heures de divertissement et de compétition en ligne avec vos amis ou d'autres joueurs sur le même réseau local.

L'histoire du jeu prenant contexte dans la capture de notre vaisseau spatial puis dans notre tentative de nous échapper du terrible labyrinthe parsemé de ses nombreux pièges, nous avons décidé que ce jeu s'intitulera : **Rocket Space Invaders.**

Pour réaliser ce projet, il a fallu créer un éditeur permettant de créer des cartes. Aussi, il a fallu concevoir un serveur et un programme pour les clients.

I- Éditeur

Dans cette partie, nous allons nous concentrer sur la partie éditeur de mondes. L'éditeur de mondes est un programme qui permet de concevoir et de sauvegarder des mondes sous forme de fichier binaire.

Interface de l'éditeur

Pour embellir le visuel dans le terminal, nous utilisons *ncurses*, qui est une bibliothèque de programmation en langage C qui facilite la création d'interfaces utilisateur textuelles sur des terminaux, en offrant des fonctionnalités avancées telles que la gestion des couleurs et des fenêtres.

Sur cette application, nous avons créé trois fenêtres :

- « **Level** », qui correspond au plateau de jeu sur lequel nous pouvons placer des éléments. Il peut y avoir jusqu'à 100 plateaux différents dans un même fichier, avec obligatoirement l'élément 'Start' qui correspond au spawn des joueurs et l'élément 'Exit' qui correspond à l'arrivée.
- « **Tools** », qui permet de sélectionner l'outil que l'on veut positionner sur le plateau. Les différents outils seront présentés dans la prochaine section.
- « **Informations** », qui permet d'afficher des informations à l'écran et des précisions sur l'utilisation des outils sélectionnés.












Voici un exemple d'affichage de l'éditeur de monde :



Outils

Dans ce programme, il y a plusieurs outils disponibles qui permettent de placer ou de supprimer des éléments sur le plateau. L'utilisateur doit cliquer sur l'outil qui veut poser, puis cliquer sur la fenêtre 'Level' pour poser l'élément. Voici la liste des éléments et les explications :

| Nom | Description | Image | Commentaire |
|-------|-------------------------------------------------------------------------------|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Block | Permet de bloquer les Joueurs/Robots/Probes. Fait office de mur ou de sol. | | Lors du premier clique sur le plateau, une croix rouge apparaît . L'utilisateur doit ensuite cliquer sur l'axe des abscisses ou l'axe des ordonnées pour faire apparaître une ligne de Block (qui commence sur la croix rouge et qui va jusqu'au 2 ^e clique). Si |

| | | | |
|--------------------------|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | | | l'utilisateur clique 2 fois sur la même case ou s'il clique sur un point qui n'est pas sur l'axe des abscisses ou des ordonnées, alors un simple Block apparaît au niveau de la croix rouge. |
| Ladder | Permet au joueur de monter ou descendre. |  | Plusieurs Ladder peuvent être combinées pour faire de grandes échelles. |
| Trap | Prends l'apparence d'un Block, mais apparaît et disparaît à intervalles réguliers. |  | Un joueur peut passer à travers le trap si celui-ci disparaît. |
| Gate | Permet de bloquer les joueurs s'ils n'ont pas la clé correspondante. |  | Les Gates peuvent être de 4 couleurs différentes (Magenta, Vert, Jaune ou Bleu). Pour changer la couleur d'un Gate, l'utilisateur doit cliquer sur une des 4 couleurs puis placer l'élément. |
| Key | Permet de faire disparaître les Gates. |  | Le joueur doit ramasser la clé de la même couleur du Gate pour pouvoir traverser les Gate. Pour changer la couleur d'une Key, l'utilisateur doit cliquer sur une des 4 couleurs puis placer l'élément. |
| Door | Permet au joueur de passer d'une Door à une autre. |  | Les Doors sont numérotées de 1 à 99 et fonctionnent par paire. Lorsque le Joueur entre par la première, il ressort par la deuxième. |
| Exit | Permet au joueur de finir le jeu. |  | Le premier joueur qui franchit cette porte gagne la partie. |
| Start | Correspond au point d'apparition du joueur. |  | Si un joueur meurt, il réapparaît à cette porte. |
| Robot | Le Robot est un monstre qui se déplace de gauche à droite. |  | Le Robot peut être tué par un joueur, mais peut aussi tuer le joueur. |
| Probe | Le Probe est un monstre volant qui se déplace dans toutes les directions. |  | Le Probe peut être tué par un joueur, mais peut aussi tuer le joueur. |
| Heart | Permet de recharger les cœurs du joueur. |  | Un joueur peut avoir au maximum 5 vies. |
| Bomb | Permet de recharger les bombes du joueur. |  | Un joueur peut avoir au maximum 5 bombes. Le joueur peut poser une bombe pour paralyser les monstres pour leur passer à travers ou pour paralyser des autres joueurs. |
| Delete (Haut de page) | Permet de supprimer un élément du plateau. | | L'utilisateur doit supprimer les cases une par une. |
| DELETE (Bas de page) | Permet de supprimer tout le plateau. | | L'utilisation de cet outil supprime l'intégralité du plateau. |

Un level est donc composé d'un ensemble d'éléments posés par l'utilisateur sur le plateau. Il est important de noter que pour le bon fonctionnement de la map, l'utilisateur doit respecter plusieurs règles :

- Afin d'éviter le lag, il peut y avoir au **MAXIMUM** 10 robots et 10 probes par level. Si l'utilisateur décide d'en mettre plus, le surplus ne sera **pas pris en compte** par le serveur lors du lancement de la partie.
- La map doit **OBLIGATOIREMENT** contenir un 'Start' pour faire apparaître les joueurs et un 'Exit' pour finir la partie.
- Si une porte est insérée, il doit y avoir une deuxième porte avec le même numéro pour permettre au joueur de passer de salle en salle. Si plus de deux portes avec les mêmes numéros sont insérées, uniquement les deux premières portes trouvées par le serveur seront prises en compte lors de la game. Si une porte possède un numéro, mais qu'il n'y a pas de deuxième porte possédant le même numéro, alors la porte devient inutilisable (ne pose pas de problème côté serveur).

Bien sûr, l'utilisateur doit utiliser son cerveau pour faire des maps cohérentes, et terminables.

Structures

Pour mettre en place l'éditeur de mondes, nous avons utilisé 4 structures qui sont les suivantes :

```
1 typedef struct {
2     int type;
3     int couleur;
4     int numero;
5 } case_t;
```

Cette structure représente une case du plateau. Chaque plateau est une map_t de 60 par 20 cases_t, il y a donc 1200 cases sur chaque plateau. Cette structure est composée de 3 champs, 'type' qui correspond au type de case (*Block*, *Ladder*, *Door* ...), chaque élément à un entier qui lui est associé et défini grâce au '#define'. Le champ 'couleur' correspond à la couleur de l'élément choisi. Ce champ est

uniquement utilisé si l'élément est une *Gate*, une *Key* ou une *trap* (les seuls éléments avec une couleur ou apparence variable). Le champ 'numero' correspond au numéro de porte. Ce champ est uniquement utilisé si l'élément est une *Door*.

```
1 typedef struct {
2     case_t matrice[NB_LIGNE][NB_COLONE];
3 } map_t;
```

Cette structure représente le plateau de jeu (un level). Elle est composée d'un champ 'matrice' qui est un tableau de case_t (structure expliquée au-dessus) de 60 par 20. Cette structure nous

permet de regrouper toutes les informations contenues sur un level et donc de pouvoir facilement modifier les modifications.

```
1 typedef struct {
2     int numeroMap;
3     off_t position;
4     off_t taille;
5 } adresse_t;
```

Cette structure représente une entrée de la table d'adressage que nous expliquerons ultérieurement. Le champ 'numeroMap' correspond au numéro (identifiant) de la map, à qui nous y associons le champ 'position' qui est la position de la map dans le fichier. Le champ 'taille' correspond à la taille de la map dans le fichier, qui est fixe dans notre cas, car nous

utilisons un tableau à deux dimensions initialisées de manière statique. Nous utilisons dans cette structure des off_t pour la taille et la position de la map. Le type off_t est un type de données entier signé qui permet de représenter les offsets du fichier.

```
1 typedef struct{
2     adresse_t tableAdressage[TABLE_MAX_SIZE];
3     int prochaine_table_existe;
4 }adresse_complete_t;
```

Cette structure représente la table d'adressage qui sera écrite dans le fichier binaire. Elle contient 2 champs qui sont 'tableAdressage' qui est un tableau d'adresse_t de 10 cases (le nombre d'entrées maximum dans la table d'adressage) et le champ 'prochaine_table_existe' qui spécifie si une autre table d'adressage existe.

Chacune de ces structures à un rôle précis dans notre code, pour permettre de simplifier l'enregistrement des données dans le fichier binaire.

Fichier binaire

Table d'adressage et table de vide

Fonctionnement

La table d'adressage est une structure de données utilisée pour stocker les informations sur la position et la taille des données stockées (des *map_t* dans notre cas) dans un fichier binaire. La table d'adressage est généralement utilisée pour faciliter l'accès et la lecture des données dans les fichiers de grande taille ou avec un nombre d'information variable.

Une table de vide est une structure de données utilisée pour connaître la position des trous (les espaces vides) dans un fichier. Elles contiennent généralement 2 champs qui sont la position du trou et la taille du trou.

Dans notre cas, nous utilisons une matrice de 60 par 20 *case_t* pour stocker tous les éléments de la map. Nous avons donc des maps qui font tous la même taille qui est de 14400o (la structure *case_t* contient 3 int, soit 12o et la table d'adressage fait 60*20 *case_t* soit 60*20*12 = 14400o).

Nous avons donc choisi de ne pas utiliser de table de vide, en sachant que toutes nos *map_t* ont une taille de 14400o et cette taille ne changera jamais.

Utilisation

| Index | n° map | Position | Taille |
|-------|--------|----------|--------|
| 0 | -1 | 248 | 14400 |
| 1 | -1 | 14648 | 14400 |
| 2 | -1 | 29048 | 14400 |
| 3 | -1 | 43448 | 14400 |
| 4 | -1 | 57848 | 14400 |
| 5 | -1 | 72248 | 14400 |
| 6 | -1 | 86648 | 14400 |
| 7 | -1 | 101048 | 14400 |
| 8 | -1 | 115448 | 14400 |
| 9 | -1 | 129848 | 14400 |

Concernant l'utilisation de la table d'adressage, nous avons la structure *adresse_complete_t* (vue précédemment) qui contient 10 entrées. Lors de la création d'une table, le numéro du level est initialisé à -1 pour toutes les entrées, la taille est calculée (mais reste fixe, car toute les levels sont des matrices de 60*20) et la position des levels dans le fichier est également calculer en fonction du nombre de level enregistrer et du nombre de table d'adressage existante.

Pour sauvegarder un nouveau level, le programme parcourt la table de vide et recherche le numéro de level. Si le numéro est de -1, alors il y a un trou de la taille de 14400o (taille fixe) et le nouveau level (qui fera également 14400o) pourras alors y être inscrite à la position correspondante.

Si les 10 entrées de la table ont un numéro de level différent de -1, c'est que la table d'adressage est pleine. Dans ce cas, une nouvelle table d'adressage est créée et ajoutée à la fin du fichier pour pouvoir ajouter des nouveaux levels. Il y a au maximum 10 tables d'adresses, soit 100 level au maximum au sein d'une map, c'est-à-dire 100 levels maximum dans un fichier.

Voici un exemple d'exécution :

| Étape | Description | Image |
|-------|-------------------------------------------------------------------|-------|
| 1 | Ajout de 10 levels. | |
| 2 | Ajout de 5 levels donc création d'une deuxième table d'adressage. | |
| 3 | Suppression du level n°7. | |
| 4 | Ajout du level n°16. | |

Après avoir vu le fonctionnement des tables d'adressages, voyons maintenant comment la map est enregistrer.

Map

Lorsqu'un utilisateur arrive sur un level (chaque level est associé à un numéro (id)), le fichier binaire va être ouvert (*open*), puis nous allons nous positionner avant la table d'adressage (*lseek*) pour pouvoir la lire (*read*). Nous parcourons ensuite la table d'adressage et si le numéro du level est trouvé dans la table, alors nous pouvons récupérer la position du level dans le fichier pour nous positionner (*lseek*) avant le level et le lire (*read*) pour l'afficher à l'écran. A noter que le nombre de level maximum par map est de 100.

Voici un court algorithme afin de comprendre le fonctionnement global du programme :


```

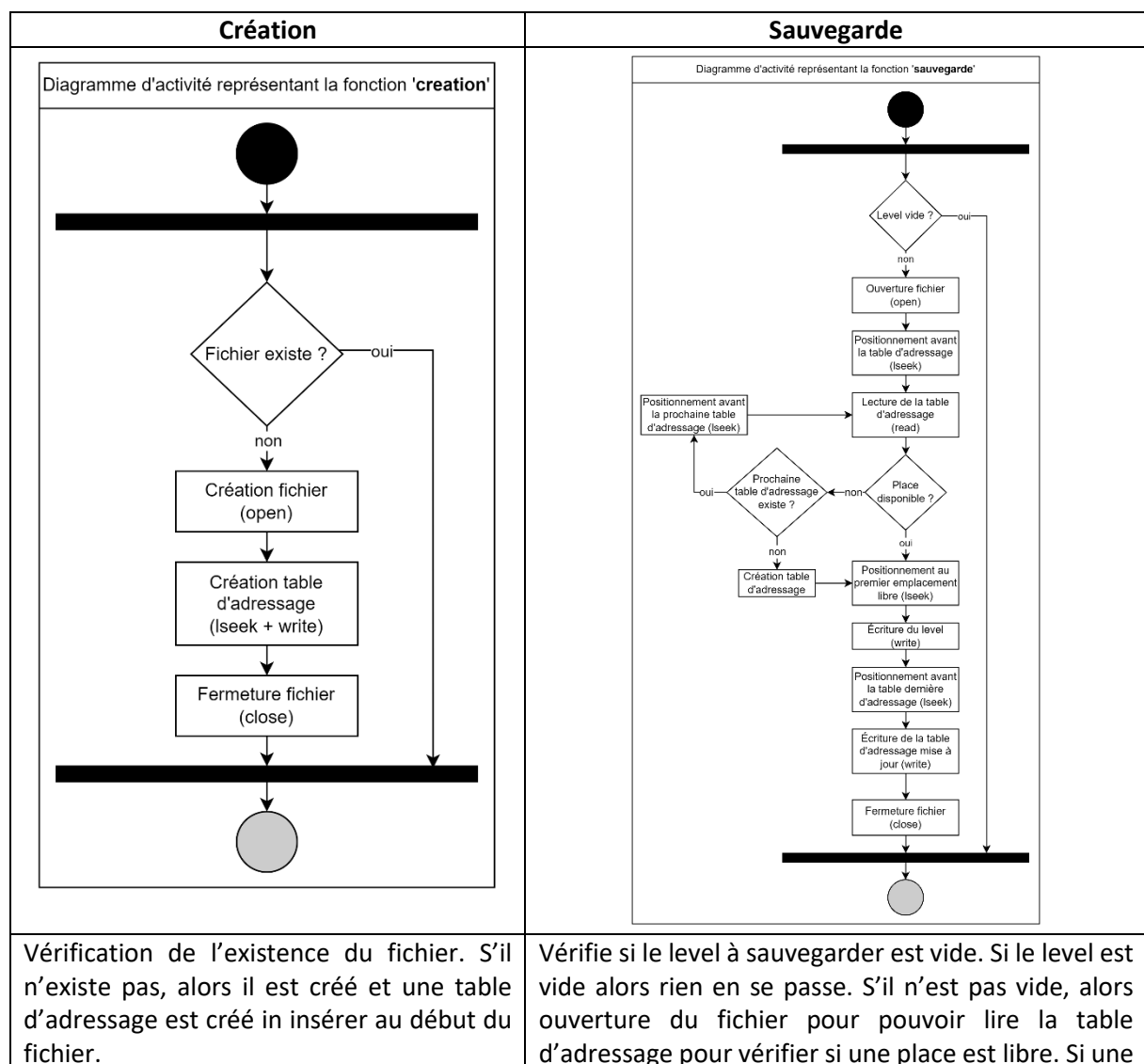
1  Creation()                                // Création du fichier binaire (+ table d'adressage)
2  Tant que non quitter
3      Chargement(idMap)                     // Chargement de la map depuis le fichier binaire
4      /*
5      * Modification de la map l'utilisateur
6      */
7      Si map non vide Alors
8          Sauvegarde(map)                   //Sauvegarde la map dans le fichier binaire
9      Si bouton 'DELETE' est cliqué Alors
10         Suppression(idMap)                // Suppression de la map dans le fichier binaire

```

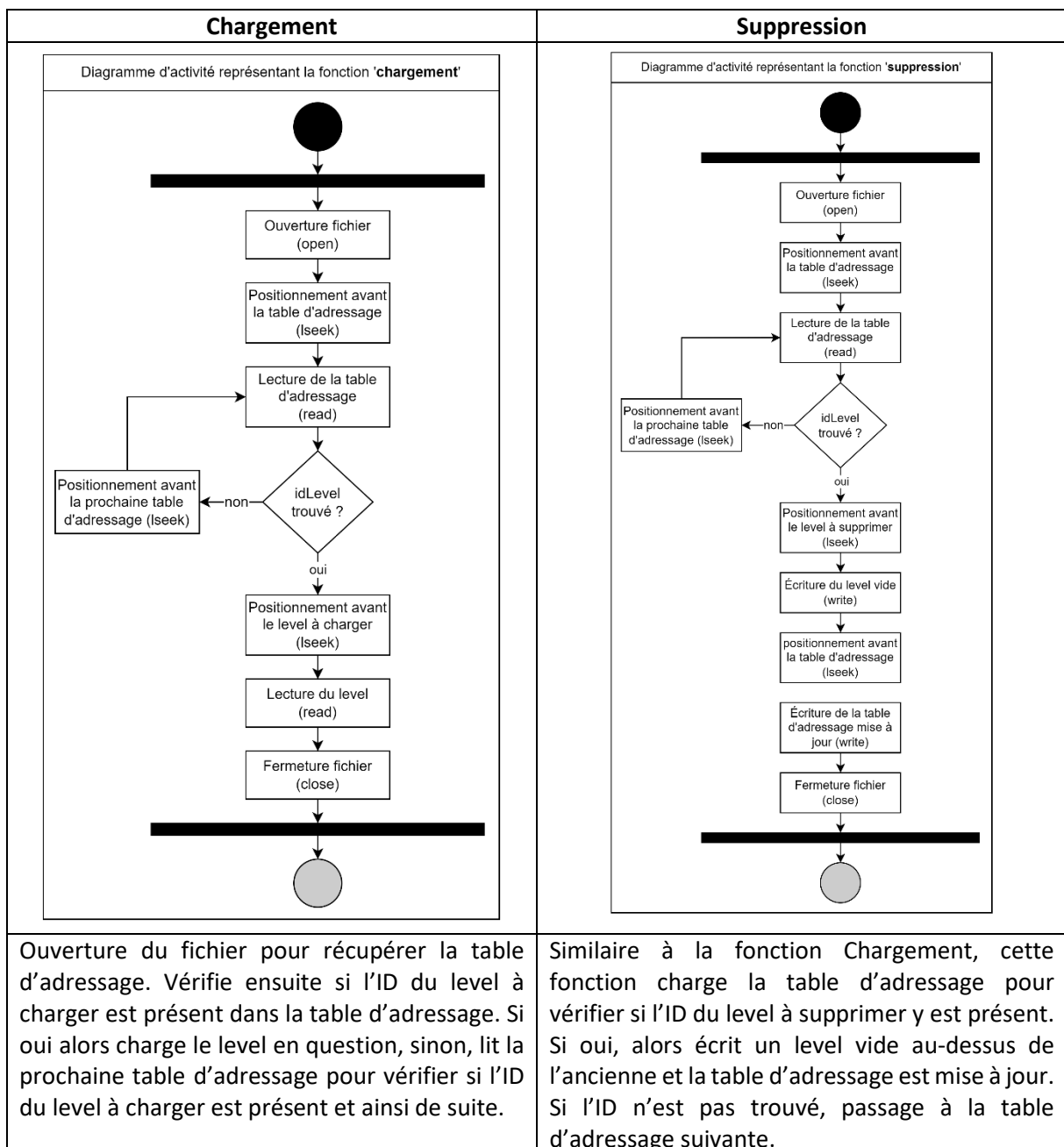
Fonction

Afin de simplifier le code, nous avons écrit plusieurs fonctions pour faciliter le code. Nous avons une dizaine de fonctions pour gérer l'affichage des boutons, des informations, de la map... Mais c'est fonction étant assez simple, elles ne nécessitent pas une explication approfondie.

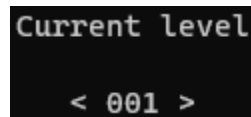
Pour apporter les modifications au fichier binaire, nous avons 4 fonctions principales : Création, Sauvegarde, Chargement et Suppression.



| | |
|--|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | place est libre alors écriture de lu level dans l'emplacement vide. S'il n'y a plus de place dans la table d'adressage, alors il y a création d'une nouvelle table d'adressage (maximum 10 tables) et sauvegarde dans cette nouvelle table. |
|--|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|



La fonction '**creation**' est appelé au lancement du programme afin de crée le fichier s'il n'existe pas. La fonction '**sauvegarder**' est appelé lorsque l'utilisateur clique sur la touche 'q' pour quitter le programme et lors du changement du level (grâce au chevron gauche et droite dans la section '*Current level*' dans la fenêtre *Tools*).

A screenshot of a terminal window with a black background and white text. The text 'Current level' is on the first line, and '< 001 >' is on the second line.

La fonction 'chargement' est appelée au lancement du programme pour charger le premier level s'il existe et lors du changement d'un level (grâce au chevron gauche et droite de la section '*Current level*' dans la fenêtre **Tools**).

La fonction 'suppression' est appelée lors du clique sur le bouton 'DELETE' en bas de la fenêtre *Tools*.

Lancement de l'éditeur

Pour lancer l'éditeur, vous devez vous placer au niveau du dossier 'Code' et commencer en faisant la commande 'make' (pour compiler les fichiers) puis vous devez écrire la commande './bin/editeur FICHIER' avec FICHIER qui est le nom du fichier binaire. Si aucun fichier n'est spécifié, alors le programme ne se lance pas. Si le fichier n'existe pas, alors il sera automatiquement créé.

La taille du terminal doit également être d'une taille de 27 par 77 au minimum. Si la taille du terminal est trop petite pour afficher toute l'interface, un message d'erreur apparaît et le programme se ferme.

II- Client

Dans cette section, nous allons aborder tous les principes d'affichage que le client utilise. Bien qu'il communique également avec le serveur, nous aborderons ce principe dans la section 'Communication entre le client et le serveur'.

Lancement du client

Lors du téléchargement du dossier, avant de pouvoir jouer, le client va devoir se positionner au niveau du dossier 'Code' et faire la commande 'make' dans le terminal, afin de rendre le code exécutable. Une fois la compilation réalisée, l'utilisateur va pouvoir lancer le programme client. Pour ce faire, l'utilisateur doit écrire la commande './bin/client ADRESSE_IP NUMERO_PORT', avec ADRESSE_IP qui correspond à l'adresse IP du serveur (127.0.0.1 dans notre cas) et NUMERO_PORT qui correspond au numéro de port de la socket (12345 dans notre cas) pour pouvoir communiquer en UDP avec le serveur.

La taille du terminal doit être de 27 par 77 au minimum. Si la taille du terminal est trop petite pour afficher toute l'interface, un message d'erreur apparaît et le programme se ferme.

Le serveur doit être démarré avant le client, sinon lors du premier envoi de donnée vers le serveur, celui-ci ne recevra rien et l'utilisateur sera bloqué (*recvfrom* qui ne recevra jamais rien).

Si le serveur est lancé, l'utilisateur va pouvoir lancer le client. Il va ensuite devoir faire un choix entre créer une partie, rejoindre une partie ou quitter le programme.

```
1. Créer une partie
2. Rejoindre une partie
0. Quitter
Votre choix : |
```

Choix 1 : L'utilisateur crée une partie, il doit ensuite choisir la map sur laquelle il veut jouer, ainsi que le nombre de joueurs maximum dans la partie. Pour finir l'utilisateur choisit s'il veut rejoindre la partie qu'il vient de créer ou non.

```
Veuillez choisir un monde parmi les suivants :
0. map1.bin
1. map2.bin
2. map3.bin
Votre Map : 1
Choisissez le nombre de joueur maximum (1-10) : 2

Voulez-vous rejoindre cette partie ?
1. Oui
0. Non
Votre choix : 1

La partie est en attente de joueurs
Merci de patienter quelques instants
```

Choix 2 : L'utilisateur veut rejoindre une partie, il va regarder dans la liste des parties disponibles et va en choisir une.

```
Voici les différentes parties disponibles :
0. Aucune
1. map1.bin      0/3
2. map2.bin      0/5
3. map3.bin      0/6
4. map1.bin      1/2
Choisissez une partie à rejoindre : |
```

S'il manque un seul joueur dans la partie (exemple : 2/3) et que l'utilisateur rejoint, alors la partie se lance. S'il manque plus d'un joueur dans la partie (exemple : 3/8) et que l'utilisateur rejoint, alors il est mis en attente, le temps que le nombre de joueurs requis soit atteint (exemple : 8/8).

La partie est en attente de joueurs
Merci de patienter quelques instants

Choix 3 : L'utilisateur quitte le programme.

Pour que la partie commence, il faut donc qu'elle soit créée et que le nombre de joueurs requis soit valide.

Interface

Une fois sur la partie commencée, l'interface ncurses apparaît pour l'utilisateur :



Tout comme l'interface de l'éditeur de monde, l'interface du client possède 3 fenêtres : la fenêtre « **Level** » qui correspond au visuel de jeu. La fenêtre « **Informations** » qui permet d'afficher des informations pour l'utilisateur et une troisième fenêtre sur la droite qui permet d'afficher les caractéristiques du joueur. Cette dernière fenêtre nous permet de voir notre nombre de clé (par défaut, tout est au rouge, c'est-à-dire aucune clé possédée), le nombre de vie (par défaut à 5), le nombre de bombes disponible (par défaut à 0 afin d'éviter le spawn kill) et le numéro du level.

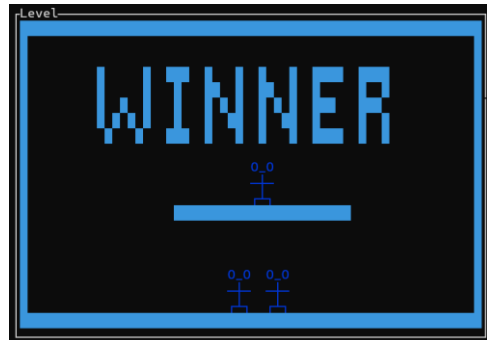
La fenêtre **Level** va varier selon le level dans lequel est le joueur. Il peut y avoir 100 levels maximum par map et il peut y avoir un nombre de map infinie. Mais il y a deux levels qui reviendront à chaque partie :

Le level *Respawn* qui apparaît lorsqu'un joueur meurt :



Pour quitter cette fenêtre, vous pouvez cliquer sur le bouton "YES" avec votre souris pour retourner au point spawn de la map, ou sur le bouton "NO" pour être renvoyé au menu principal (ragequit ???).

Le level *Winner* qui apparaît lorsqu'un joueur gagne :



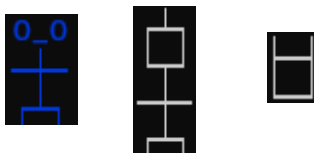
Lorsqu'un joueur gagne, tous les joueurs de la partie sont téléportés sur ce level. Le vainqueur est mis en avant sur la plateforme en tant que "WINNER", tandis que les perdants, eux, sont relégués en bas et ne peuvent qu'observer le succès du vainqueur.

Règle

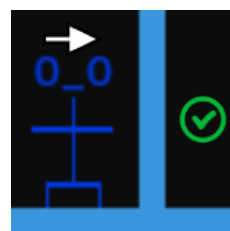
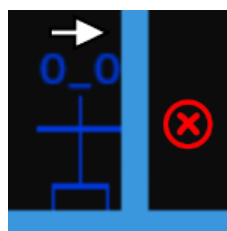
Dans cette partie, nous nous concentrerons sur les règles du jeu que l'utilisateur doit respecter. Cependant, il convient de noter que la gestion effective de ces règles est **exclusivement** assurée par le code du serveur, que nous verrons dans une prochaine partie.

Dans ce jeu, chaque joueur contrôle un personnage et le déplace dans un monde contenant plusieurs tableaux (les levels). L'objectif est d'atteindre la sortie avant les autres, en récupérant les clés pour passer les portails, tout en évitant les ennemis (robots et sondes), les pièges et les bombes des adversaires.

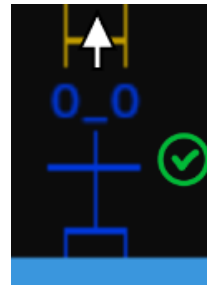
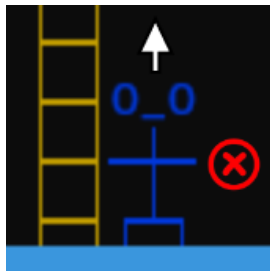
Voici le joueur, le robot et le probe :



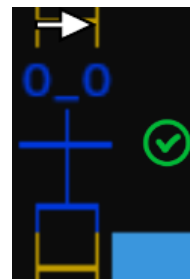
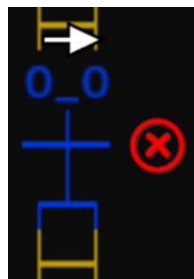
Concernant le déroulement de la partie, une fois que le nombre de joueurs nécessaire pour lancer la partie est atteint, tous les joueurs apparaissent sur une porte de couleur magenta nommée Start (les joueurs occupent la même place). L'objectif des joueurs est d'être le premier à finir la partie, c'est-à-dire passer la porte Exit en premier. Pour ce faire, ils vont pouvoir se déplacer de gauche à droite à l'aide des flèches directionnelles, jusqu'à rencontrer un mur ou une GATE.



Les joueurs peuvent également monter ou descendre (flèche directionnelle haute et basse), à condition d'être sur une échelle.



Si le joueur est sur une échelle, alors il ne peut aller ni à gauche, ni à droite, il est obligé de monter ou de descendre **sauf** s'il y a un block qui lui permet de marcher dessus.

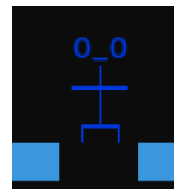
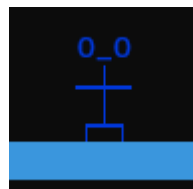


Si une *GATE* bloque le passage du joueur, celui-ci va devoir trouver la clé de la même couleur que la *GATE* pour pouvoir la traverser. Les clés récupérées sont affichées en haut à droite de l'interface. Elles sont par défaut rouges, c'est-à-dire non posséder et prennent une couleur lorsque l'utilisateur ramasse une clé. Il y a 4 couleurs possibles (Magenta, Vert, Jaune et Bleu). Pour ramasser une clé, il suffit de la traverser.



Lors de son aventure, le joueur peut rencontrer des *traps*. Ces éléments prennent l'apparence des *blocks*, mais disparaissent et réapparaissent à intervalle régulier.

Si le joueur est sur un *trap* lorsqu'il disparaît, alors le joueur tombe et perd une vie.



Si le joueur se trouve sur la case d'un trap lors de sa réapparition, alors il meurt instantanément.

Concernant la mort d'un joueur, il y a deux moyens pour lui de mourir. Le premier étant d'avoir un nombre de vie de 0 et le deuxième est lors de la réapparition du trap comme dit précédemment.

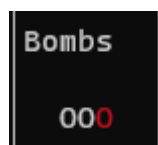
Le joueur possède par défaut 5 vies. Afin de ne pas perdre de vie, le joueur doit faire attention à plusieurs choses : les dégâts de chute, même si la chute ne fait qu'un seul block de haut, le joueur perd une vie. Rentrer en contact avec un monstre fera également perdre une vie au joueur, même si c'est l'antenne du robot qui vient toucher le pied du joueur, car les contacts avec les monstres sont vérifiés

grâce aux dimensions des entités (3*3 pour le joueur, 3*4 pour le robot et 3*2 pour le probe). Être dans le rayon d'explosion d'une bombe fait également perdre une vie au joueur et en plus, le paralyse pendant 5 secondes. Si le joueur atteint le nombre de vie de 0, alors il meurt et est envoyé dans la map respawn présenté précédemment. Lors de la perte d'une vie, le joueur passe en état 'GOD MODE', c'est-à-dire qu'il est immunisé contre tous les dégâts et contre les paralysies pendant trois secondes. Pendant cet état, afin d'avoir un marquage visuel des trois secondes, le joueur va devenir multicolore (comme l'étoile dans MarioKart) et ses expressions faciales vont changer aléatoirement. Le joueur redeviendra normal après les trois secondes.



Afin d'explorer la map à la recherche de l'*Exit* (une porte de couleur jaune) et les différentes clés pour ouvrir les portails, le joueur va devoir passer de niveau en niveau en utilisant des portes de couleur verte, numérotée de 1 à 100. Passer dans une porte vous fera ressortir à la deuxième porte portant le même numéro.

Sur la route du joueur, il va également se trouver des monstres : les robots et les probes. Ces monstres ne possèdent pas de vie, ils ne sont donc pas tuables, mais possèdent un point faible : les bombes. Chaque joueur a la possibilité de poser des bombes s'il en a en réserve (la réserve de bombe est visible sur la droite de l'interface). Un 'O' affiché en blanc signifie que la bombe est disponible.

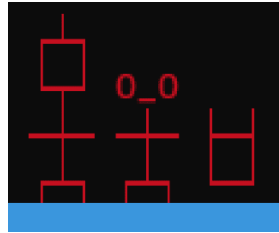


Pour poser une bombe, le joueur doit appuyer sur la touche 'b' de son clavier. Les bombes ont un rayon de cinq caractères. En réalité, nous avons choisi de faire une explosion de cinq caractères à gauche et à droite, et deux caractères en haut et en bas, car dans ncurses, les caractères ne font pas la même taille en hauteur quand largeur. Nous avons donc essayé de faire l'explosion la plus ronde possible et nous avons décidé de choisir cinq caractères en abscisse et deux en ordonnée, ce qui nous donne le résultat suivant :



À noter que pour plus de complexité dans le jeu, l'explosion d'une bombe passe à travers les murs, c'est-à-dire qu'il n'est pas possible de se cacher, le seul moyen pour ne pas perdre de vie et de s'éloigner.

Lorsqu'une entité est touchée par le rayon d'explosion d'une bombe, elle est paralysée pendant 5 secondes (et perd une vie si l'entité est un joueur), ce qui signifie que l'entité ne peut plus se déplacer. Pour avoir un indicateur visuel du temps de paralysie, les entités touchées deviennent rouges :



Si un joueur est en état 'PARALYSER', et qu'il se fait toucher par un robot ou un probe, alors le joueur perd une vie, perd son état 'PARALYSER' et passe en état 'GOD MODE' pendant 3 secondes.

Lorsqu'un robot ou un probe est paralysé, les joueurs peuvent le traverser sans perdre de vie.

Une partie se finit lorsqu'un joueur passe la porte *Exit*.

Structures

Dans cette partie, nous allons uniquement expliquer les structures qui sont utilisées dans le client. Les structures servant à faire passer des informations du serveur vers le client seront expliquées dans une prochaine partie.

Nous avons donc plusieurs structures tel que :

La structure *map_t*, qui contient une matrice de *case_t*, qui a déjà été expliquée dans la partie éditeur.

```
1 typedef struct thread_affichage_client_t{
2     map_t map;
3     joueur_t tab_joueur[NB_MAX_PLAYER];
4     robot_t tab_robot[NB_MAX_MOB];
5     probe_t tab_probe[NB_MAX_MOB];
6     int nb_joueur;
7     int id_joueur_actuel;
8 } thread_affichage_client_t;
```

Cette structure, déclarée comme variable globale dans le client, nous sert à passer la map, le tableau de joueur, le tableau de robot et le tableau de probe au thread afficheur. Le client va recevoir cette structure du serveur à chaque tour de boucle, ce qui va nous permettre de pouvoir afficher les informations mises à jour.

Fonctions

Comme pour l'éditeur de monde, plusieurs fonctions d'affichage sont utilisées, mais elles ne seront pas expliquées car elles contiennent uniquement des *mvwprintw* ou des *mvaddch* pour afficher des caractères. La seule chose à dire sur ces fonctions est la priorité d'affichage. Le joueur passe toujours au premier plan, suivi des monstres, puis des bombes, et pour finir, la map. Le choix d'afficher ces éléments indépendamment les uns des autres, s'explique par le fait que nous avons choisi d'avoir des levels qui ne change jamais, c'est-à-dire que tous les robots, probes, joueurs et bombes ne sont pas inscrits dans la map. Nous avons en conséquence, un tableau de joueur, un tableau de probe, un tableau de robot et un tableau de trap (pour chaque partie créée), que l'on vient parcourir et superposer à la map lors de l'affichage, grâce à l'attribut 'position' de chacun de ces éléments.

La fonction *main* permet de faire les différents choix pour créer ou rejoindre une partie. Puisqu'elle sert principalement à communiquer avec le serveur, cette fonction sera plus détaillée dans notre section 'Communication entre le client et le serveur'.

Notre fonction principale est la fonction '*jouer*' qui est appelée lorsque le joueur a rejoint une partie. La fonction commence en créant une socket TCP, suivie de l'adressage et de la connexion au serveur.

grâce au port TCP que le serveur a envoyé. Nous créons ensuite un thread dédié à l’affichage sur lequel nous reviendrons un peu plus tard. Une fois la connexion établie avec le serveur, le client va pouvoir lire la map et le tableau de joueur que lui envoie le serveur grâce à l’appel ‘*read*’. Nous pouvons ensuite initialiser *ncurses* avec les différentes fonctions ‘*ncurses_init*’, ‘*ncurses_init_mouse*’, ‘*ncurses_colors*’... Nous pouvons ensuite créer l’interface composée de 3 fenêtres comme expliquer dans la partie ‘Interface’. Il y a par la suite une boucle *while*, qui va venir récupérer la touche sur lequel à cliquer l’utilisateur avec la fonction ‘*getch*’. Nous avons utilisé un timeout de 0.1 seconde pour la fonction *getch*, qui permet de rendre non bloquante l’attente d’une touche. Grâce au timeout, le joueur peut lâcher le clavier, ce qui le fera rester immobile, mais la map quant à elle, continuera de s’actualiser. Une fois une touche pressée (GAUCHE, DROITE, BAS, HAUT, Q, B), celle-ci va être enregistrer dans une variable de type *deplacement_t* avec l’identifiant du joueur (structure utilisée pour communiquer avec le serveur, elle sera expliquée dans la prochaine partie).

Le client va ensuite pouvoir envoyer la structure au serveur grâce à un ‘*write*’. Le serveur va récupérer la touche et faire la gestion des règles que nous verrons dans une prochaine partie. Une fois toutes les vérifications réalisées dans le serveur, celui-ci va envoyer au client la structure ‘*thread_affichage_client_t*’, contenant la map, le tableau de joueur, le tableau de robot et le tableau de probe mise à jour. Les informations vont ensuite pouvoir être récupérer par le thread dédié à l’affichage grâce la structure ‘*thread_affichage_client_t*’ déclarer en tant que variable global. La boucle *while* recommence jusqu’à ce que l’utilisateur presse la touche ‘Q’ pour quitter et que la partie se finisse pour lui.

Concernant le thread dédié à l’affichage, il est associé à une routine qui contient une boucle *while*, qui va attendre un signal avec un *pthread_cond_wait*. Ce signal va être envoyé (*pthread_cond_signal*) depuis la fonction jouer, une fois que toutes les informations auront été reçues depuis le serveur.

Voici un brief résumer des fonctions ‘jouer’ et ‘*routine_thread_affichage*’ au format algorithmique :

```
1  fonction jouer:
2      tant que non quitter :
3          saisir(touche) OU timeout()    // Récupération de la touche du joueur ou timer
4          write(touche)                  // Envoie de la touche au serveur
5          read(structure_information)     // Réception des données mise à jour (map, joueur, robot, probe)
6          envoie_signal()                 // Envoie du signal grâce à pthread_cond_signal
7
8  fonction routine_thread_affichage :
9      tant que non quitter :
10         attente_signal()                // Attente du signal grâce à pthread_cond_wait
11         affichage()                     // Permet d'afficher la map, probes, robots, bombes et joueurs
```

Nous voyons qu’aucune règle n’est gérer dans le client, celui-ci a pour unique but de récupérer la touche de l’utilisateur, puis d’afficher la map.

III- Serveur

Le serveur est l'élément central de ce projet. Il a pour but de gérer la création de parties, ainsi que le bon déroulement de ces dernières, en comprenant la gestion des règles du jeu et des différents threads.

Lancement du serveur

Une fois le 'make' réalisé une première fois, l'utilisateur va pouvoir écrire la commande './bin/serveur', en étant dans le dossier *Code* afin de lancer le serveur.

Une fois le serveur démarré, l'utilisateur n'a plus besoin d'interagir avec lui, c'est le client, depuis le terminal du client, qui va choisir ce qu'il souhaite faire, et tout sera fait automatiquement. Le serveur a donc uniquement besoin d'être démarré et va tourner seul en arrière-plan. Le serveur peut éventuellement être redémarré afin de mettre à jour la liste des maps disponibles.

Structure

Les structures utilisés côté serveur servent principalement à représenter des données telles que des parties ou encore servent pour passer des informations en arguments aux threads.

```
1 typedef struct partie_t{
2     int id;
3     int nbJoueurMax;
4     int nbJoueurConnecte;
5     char nomMonde[MAX_CHAR];
6     int *tab_adresses_clients;
7 }partie_t;
```

La représentation d'une partie se fait grâce à cette structure. Afin d'être identifiable, elle utilise un *id*. Une partie est caractérisée par son nombre de joueur requis (champ *nbJoueurMax*), son nombre de joueur actuellement connecté et par le nom de la carte sur laquelle les joueurs jouent. Aussi, dans le but de communiquer avec les joueurs, leurs adresses sont conservées dans le champ *tab_adresses_clients* et seront réutilisées au moment de

l'établissement de la connexion TCP.

```
1 typedef struct bomb_t{
2     position_t pos;
3     int etat;
4     int range_gauche;
5     int range_droite;
6     int id_level;
7 } bomb_t;
```

Cette structure représente la bombe, elle permet de stocker plusieurs informations telles que la position de la bombe (attribue *pos*), l'identifiant du level dans lequel est la bombe (attribut *id_level*), *range_gauche* et *range_droite* qui représente la portée de l'explosion à gauche et à droite de celle-ci, et l'état qui peut être 'EXPLOSION' si la bombe est en train d'exploser, 'ATTENTE' si la bombe a été posée mais n'a pas encore exploser et 'INEXISTANTE' si la bombe est dans l'inventaire du joueur.

```
1 typedef struct joueur_t{
2     int id;
3     position_t pos;
4     int id_level;
5     int nb_life;
6     int nb_Bomb;
7     int got_key[NB_MAX_KEY];
8     int dernier_mouvement;
9     int etat;
10    int statut;
11    bomb_t tab_bomb[NB_MAX_BOMB];
12    pthread_mutex_t mutex_joueur;
13 } joueur_t;
```

La structure *joueur_t* représente le joueur, elle contient différents attributs pour l'identifier, connaître son emplacement et son inventaire. L'attribut *id* représente l'identifiant du joueur, il permet de différencier les joueurs. L'attribut *pos* permet de connaître la position en X et Y du joueur. Cet attribut va avec l'*id_level* qui est égal à l'id de la map dans lequel est le joueur. *Nb_life* correspond au nombre de vie du joueur (max 5) et *nb_Bomb* correspond au nombre de bombe dans l'inventaire du joueur. Le tableau *got_key* permet de stocker les clés que l'utilisateur à ramasser. L'attribut *dernier_mouvement* est égal à la

dernière touche de l'utilisateur, il permet notamment d'empêcher le joueur de poser plusieurs bombes d'affilé involontairement (en restant appuyer trop longtemps sur la touche 'b'). L'état du joueur peut prendre les valeurs de 'NORMAL' qui est l'état par défaut, 'FALLING' qui permet d'identifier si le joueur est en train de tomber ou 'TRANSITION_PORTE' qui intervient quand le joueur passe une porte. L'état

est donc plus porter vers le déplacement, alors que l'attribut statut, se concentre sur l'aspect vie du joueur. Il peut prendre la valeur de 'NORMAL' s'il ne se passe rien, 'PARALYSER' si le joueur est paralysé ou 'GOD_MODE' s'il vient de prendre des dégâts. Pour finir, le joueur possède un tableau de bombe qui correspond aux bombes que le joueur peut poser sur le plateau.

Les structures *robot_t* et *probe_t* représentent les deux types de monstres : les robots et les probes. Ils possèdent tous les deux les mêmes attributs : un id pour les identifier, une position, une direction qui est GAUCHE ou DROITE pour le robot et GAUCHE, DROITE, HAUT ou BAS pour le probe. L'état peut prendre la valeur 'SPAWN' s'ils viennent d'apparaître sur le plateau, 'PARALYSER' s'ils sont paralysés à cause d'une bombe, sinon 'NORMAL'.

```
1 typedef struct {
2     int id_partie;
3     int nombre_map_total;
4     int adresse_joueur;
5 }info_thread_player_t;
```

Cette structure permet de passer l'identifiant de la partie, le nombre de map et l'adresse d'un joueur, en paramètre d'un thread qui permet de gérer un joueur.

```
1 typedef struct {
2     int id_partie;
3     bomb_t* bombe;
4 }info_thread_bomb_t;
```

Cette structure permet de mettre la bombe que le joueur vient de poser dans un thread, elle fait office de timer avant de faire exploser la bombe. Elle prend donc l'identifiant de la partie dans lequel est le joueur, et l'adresse de la bombe qui vient d'être posé, pour pouvoir changer son état.

```
1 typedef struct thread_map_robot_t{
2     int id_partie;
3     robot_t* robot;
4     map_t* map;
5 } thread_map_robot_t;
6
7 typedef struct thread_map_probe_t{
8     int id_partie;
9     probe_t* probe;
10    map_t* map;
11 } thread_map_probe_t;
```

Tout comme les structures du dessus, ces deux structures sont utilisées pour envoyer des informations au thread. *Thread_map_robot_t* permet d'envoyer l'identifiant de la partie, la map sur lequel est le robot, et le robot en question. De même pour *Thread_map_probe_t* mais qui envoie un probe à la place du robot. Le robot et le probe sont des pointeurs, ce qui nous permet de modifier leur attribue dans le thread, tel que leur position une fois leur déplacement effectuer.

```
1 typedef struct thread_trap_t{
2     int id_partie;
3     case_t** tab_trap;
4     int nb_trap;
5     int *tab_id_level_trap;
6 } thread_trap_t;
```

La structure *thread_trap_t*, contenant l'identifiant de la partie, le nombre de traps total et un tableau de pointeur de trap, est une structure envoyée en paramètre du thread. L'utilisation d'un tableau de pointeur permet de modifier les attributs de tous les traps (VISIBLE ou INVISIBLE). L'attribut *tab_id_level_trap* nous permet de

savoir dans quel level sont les traps, ce qui nous est utile pour comparer le level des joueurs et celui des traps, ce qui permet de ne pas charger les traps d'un level si aucun joueur n'est présent sur ce level. En sachant qu'il n'y a pas de nombre maximal de trap, ne pas charger les traps qui ne sont visibles pour aucun joueur, nous permet d'économiser des ressources et de diminuer la latence.

```
1 typedef struct game_info_t{
2     probe_t tab_probe[NB_LEVELS_MAX][NB_MAX_MOB];
3     robot_t tab_robot[NB_LEVELS_MAX][NB_MAX_MOB];
4     joueur_t tab_joueur[NB_MAX_PLAYER];
5     int nb_joueur;
6
7     map_t tab_map[NB_LEVELS_MAX];
8     position_t pos_start;
9     int id_map_start;
10 } game_info_t;
```

Cette dernière structure sert pour une variable globale. En effet, puisqu'elle comprend les tableaux des joueurs, sondes et robots ainsi que la carte les threads peuvent consulter la positions des entités et appliqué les différentes règles du jeu. Pour le besoin de certaines fonctions ou certains threads, le nombre de joueur est aussi présent dans la structure. La position de la porte de départ et le

numéro de level dans lequel elle se trouve sont aussi inscrits dans la structure notamment pour permettre au joueur de recommencer sa partie s'il meurt.

Fonctions et Threads

Lors du lancement du serveur, celui-ci va récupérer le nom de toutes les cartes disponibles grâce à la fonction '*recuperer_nom_map*' qui va ouvrir le sous-dossier «Map» (*opendir*), parcourir tous les fichiers (*readdir*) se finissant par l'extension '.bin', stocker leur nom dans un tableau puis fermer le dossier (*closedir*).

Le serveur met en place deux gestionnaires de signaux, l'un pour s'occuper des signaux SIGCHLD et l'autre des SIGINT. Ainsi, le gestionnaire *handler_SIGCHLD* permet d'attendre la fin d'un fils lorsque ce dernier se termine. Dans le cas de *handler*, qui s'occupe des SIGINT, il permet de changer la variable globale *stop* dans le but d'arrêter la boucle présente dans le *main* qui sert à gérer la création et l'accès aux parties. De plus, le tableau global permettant d'arrêter les threads des monstres et des pièges est lui aussi modifié pour stopper ces threads. Enfin, ce gestionnaire attend la fin de tous les fils en cours avant de terminer le programme.

Ensuite, le serveur crée une socket UDP pour communiquer avec les clients, mais nous développerons cette partie ultérieurement. Nous allons parler ici uniquement du code qui touche seulement le serveur.

Une fois la partie créée et tous les joueurs connectés, la partie va pouvoir se lancer. Le serveur va lire le fichier binaire grâce à la fonction '*lecture_fichier_map*'. Cette fonction reprend les mêmes bases que la fonction 'chargement' de l'éditeur, qui permet de charger un seul level, mais cette fois ci, notre nouvelle fonction va parcourir toute la map pour charger tous les levels d'un coup. Notre fonction lit donc la table d'adressage, la parcourt, charge les différents levels et charge la prochaine table d'adressage si elle existe et ainsi de suite.

Après avoir chargé tous les levels, le serveur va pouvoir préparer la partie. La fonction '*recherche_start_map*' va être appelé et va permettre de trouver le level sur lequel est le start et la position de ce start. Le serveur va ensuite préparer toutes les entités. Il va commencer en appelant la fonction '*compte_entite*' qui va permettre de créer tous les robots, les probes et la fonction '*initialiser_trap*' qui va permettre de créer les traps. Toutes ces entités vont être insérées dans des tableaux qui sont contenus dans la structure '*game_info_t*'. Nous avons choisi de mettre ces entités dans des tableaux différents et de ne pas les inscrire sur la map pour des questions d'efficacité. Selon nous, il est préférable de faire ce choix, ce qui permet notamment d'avoir plusieurs entités sur la même case, sans poser de problème de gestion d'accès à une case. Notre stratégie va par exemple nous permettre de simplement gérer le cas où un joueur pose une bombe sur un trap qui sera en état invisible avec en même temps un robot et un probe sur cette même case. En ayant les entités inscrit directement dans la case, nous aurions eu une case d'un level qui aurait dû contenir dans ce cas : un trap, une liste de joueurs, une bombe, une liste de robots ainsi qu'une liste de probes. Grâce à notre technique, nous pouvons avoir beaucoup d'entités sur les mêmes cases, ce qui ne nous pose aucun problème puisque nos entités sont indépendantes les unes des autres.

Une fois toutes les entités récupérées, le serveur va créer un thread pour chaque robot (fonction '*routine_thread_robot*'), un thread pour chaque probe (fonction '*routine_thread_probe*'), et un thread pour l'ensemble des pièges (fonction '*routine_thread_trap*'). Nous estimons qu'il est important que tous les traps soit synchronisées (qu'ils changent tous d'état en même temps). Tandis que les robots ou les probes peuvent être désynchronisé, dans le cas où l'un d'eux devient paralysé. Cela nous permet également de pouvoir changer le temps entre chaque déplacement d'un monstre, dans le cas où nous souhaitons complexifier le jeu.

Dans la fonction '*routine_thread_trap*', afin de réduire la latence, uniquement les traps qui sont sur un level contenant au moins un joueur vont changer d'état. Toutes les 2 secondes, tous les traps dans un level contenant au moins un joueur vont passer de l'état VISIBLE à INVISIBLE ou inversement.

```
1 fonction routine_thread_trap :
2   Tant que partie non fini :
3     Si le trap est dans le même level que un joueur Alors
4       Si trap.couleur == VISIBLE Alors
5         trap.couleur = INVISIBLE
6       Sinon
7         trap.couleur = VISIBLE
8     sleep(2)
```

Dans la fonction '*routine_thread_robot*', le premier déplacement du robot est choisi aléatoirement parmi GAUCHE ou DROITE. Le robot va donc aller dans une direction (GAUCHE ou DROITE), jusqu'à rencontrer un mur ou du vide au niveau du sol, puis changera de direction pour aller dans l'autre sens. À noter que les robots peuvent se croiser.

```
1 fonction routine_thread_robot :
2   Tant que partie non fini :
3     Si le robot n'est pas paralysé Alors
4       deplacement_robot(robot)
5       nanosleep(500000000) // = 0.5s
```

Dans la fonction '*routine_thread_probe*', les déplacements du probe sont choisi aléatoirement parmi HAUT, DROITE, BAS ou GAUCHE. Le probe ne peut pas traverser les murs et peuvent se croiser.

```
1 fonction routine_thread_probe :
2   Tant que partie non fini :
3     Si le probe n'est pas paralysé Alors
4       deplacement_probe(probe)
5       nanosleep(500000000) // = 0.5s
```

Après avoir chargé la map et initialiser les entités, le jeu va pouvoir commencer. Le serveur va créer un thread (fonction '*thread_joueur*') pour chaque joueur de la partie, et ce sont ces threads qui vont gérer les **déplacements du joueur** et la **gestion des règles du jeu**.

Les threads des joueurs commence en créant le level de respawn expliqué précédemment, grâce à la fonction '*create_map_respawn*'.

Par la suite, nous trouvons une boucle while qui va tourner pendant toute la durée de la partie. La première instruction est un *read*, qui permet de lire la touche de l'utilisateur. Si le joueur n'est pas en état 'PARALYSER', alors il peut se déplacer. Selon le déplacement de l'utilisateur, le serveur va appeler différentes fonctions :

- Si le joueur souhaite se déplacer vers la gauche, la fonction '*aller_a_gauche*' va être appelée et va vérifier plusieurs conditions : que le joueur n'est pas sur une échelle, que le joueur n'est pas bloquer par une GATE (s'il ne possède pas la clé pour traverser la GATE), que le joueur n'est pas bloqué par un mur ou des traps ou s'il traverse une porte.
- Même chose si le joueur se déplace vers la droite, mais les coordonnées de vérification vont changer (les coordonnées du joueur sont en bas à gauche des dimensions du joueur).

- Si l'utilisateur veut monter ou descendre, le serveur vérifie s'il peut le faire grâce aux fonctions '*peut_monter*' ou '*peut_descendre*' qui vérifie simplement si les cases au-dessus ou en dessous du joueur sont des échelles.
- Si la touche vaut 'B', pour la bombe, la fonction '*poser_bomb*' est appelée, elle permet de calculer la portée de l'explosion à gauche et à droite, jusqu'à rencontrer un bloc, avec une portée de cinq caractères maximum. Suite au calcul de la portée, un thread est créé pour la bombe du joueur, avec la routine '*routine_thread_bombe*' qui attend 5 secondes, puis vérifie à l'aide de la fonction '*verif_explosion*', s'il y a des joueurs, robots ou probes dans le rayon de l'explosion. Si une entité est trouvée, alors la fonction '*paralyser_entite*' est appelée, elle change l'état de l'entité à PARALYSER (ce qui rend l'entité immobile) et crée un thread, prenant l'adresse vers la variable état de l'entité. Après 5 secondes, l'état de l'entité repasse à NORMAL.

Ensuite, vient la vérification des autres règles du jeu, comme la fonction '*player_falling*' qui va vérifier, après le déplacement du joueur, si celui-ci est dans le vide, c'est-à-dire en train de tomber. On vérifie ensuite si le joueur touche un item avec la fonction '*verif_recup_item*'. S'il touche un item LIFE, alors il voit son nombre de vie remonté à cinq et le thread '*routine_thread_life*' est lancé, ce qui permet de faire disparaître l'item qui vient d'être ramassé, et va le faire réapparaître 10 secondes plus tard. Même chose pour l'item BOMB.

La fonction '*verif_changer_map*' qui vient vérifier avec les coordonnées du joueur, s'il traverse une porte suivi de la fonction '*verif_perdre_vie*' qui vérifie si le joueur se trouve à la même position qu'un TRAP quand celui-ci réapparaît. Si c'est le cas, alors le joueur meurt instantanément. Elle compare également la position du joueur avec celle du robot et probe, afin de savoir s'ils se touchent. Dans ce cas, le joueur perd une vie, la fonction '*activation_GOD_MODE*' est appelée, ce qui fait passer le joueur en statut GOD_MODE, et qui crée un thread qui fait guise de timer pour faire passer le joueur en statut NORMAL après 3 secondes.

À la suite de ces fonctions, '*verif_joueur_mort*' permet de vérifier si le joueur a un nombre de vie de 0. Si c'est le cas, alors le serveur envoie et envoie les informations au client, mais en passant la map de respawn au joueur. De plus, la structure étant accompagnée d'un tableau de joueur et de ceux des monstres, ces trois tableaux sont remplacés par des tableaux vides qui font disparaître toutes les entités. Le joueur devra cliquer sur le YES s'il souhaite respawn ou le NO s'il ne veut pas. Si le joueur souhaite respawn, alors il est téléporté à la porte start et son inventaire est réinitialisé.

La dernière fonction appelée dans la gestion des règles du jeu est la fonction '*verif_exit*' qui vérifie si un joueur passe la porte exit. Si c'est le cas, alors tous les joueurs sont téléportés sur un level de fin de partie. Ce thread étant un peu complexe, voici un résumé au format algorithmique :


```

1  fonction thread_joueur :
2      creation_map_respawn()           // Crée le level pour le respawn
3      Tant que partie non fini :
4          recvfrom(touche)             // Reçoit la touche du joueur
5
6          Si touche == GAUCHE Alors    // Si le joueur presse la touche flèche gauche
7              aller_a_gauche(map, joueur)
8          Sinon Si touche == DROITE Alors // Si le joueur presse la touche flèche droite
9              aller_a_droite(map, joueur)
10         Sinon Si touche == HAUT Alors // Si le joueur presse la touche flèche du haut
11             peut_monter(map, joueur)
12         Sinon Si touche == BAS Alors // Si le joueur presse la touche flèche du bas
13             peut_descendre(joueur)
14         Sinon Si touche == BOMB Alors // Si le joueur presse la touche 'b'
15             poser_bomb(map, joueur)
16         Sinon Si touche == QUITTER Alors // Si le joueur presse la touche 'q'
17             quitter_partie()
18
19         player_falling(map, joueur)    // Vérifie si le joueur est en train de tomber
20         verif_recup_item(map, joueur)  // Vérifie si le joueur récupère un item
21         verif_changer_map(map, joueur) // Vérifie si le joueur passe dans une Door
22         verif_perdre_vie(map, joueur, tab_robot, tab_probe) // Vérifie si le joueur perd de la vie
23         verif_joueur_mort(joueur)      // Vérifie si le joueur est mort
24         verif_exit(map, joueur)        // Vérifie si le joueur passe l'Exit
25
26         // Envoie un structure contenant le tableau de joueur, la map, le tableau de robot et la tableau de probe
27         write(thread_affichage_client)

```

Concernant la concurrence du programme, chaque modification ou accès à une variable est exécuté en exclusion mutuel par des mutex (`pthread_mutex_lock` et `pthread_mutex_unlock`). En effet, chaque joueur ou monstre a un mutex. Ainsi, lorsque le serveur effectue une vérification ou une modification sur un joueur, un robot ou un probe, on verrouille le mutex de l'entité concerné avant d'effectuer les vérifications ou traitements. Donc, pour chacune des fonctions décrites précédemment pour le joueur, le mutex de ce joueur est verrouillé, des vérifications sont effectués puis si les conditions sont remplies, une modification d'une donnée du joueur est faite avant de déverrouiller le mutex. Ce processus intervient aussi pour les monstres.

Lorsqu'un joueur décide de quitter la partie, son thread s'arrête. Une fois que tous les joueurs ont quitté une partie, tous les threads des robots et des probes s'arrêtent également en étant attendu par des `pthread_join`. Toutefois, nous n'avons pas réussi à faire l'annulation retardé pour les joueurs et les monstres.

IV- Communication entre le client et le serveur

Que ce soit pour créer et rejoindre des parties ou encore pour le bon déroulement du jeu, le client et le serveur ont besoin de communiquer. Pour ce faire, nous avons utilisé deux protocoles réseaux, UDP et TCP ainsi que des structures que vont s'échanger les deux interlocuteurs. Afin de bien comprendre les échanges, voyons d'abord les structures utilisées lors de la communication.

Structures de communication

Les structures diffèrent selon si le client décide de créer une partie ou d'en rejoindre une. Qu'importe son choix, il utilisera principalement la structure '*game_request_t*' pour communiquer ses intentions. En effet, le client recevra différentes structures de la part du serveur, néanmoins, il ne répondra qu'à travers celle mentionnée précédemment.

```
1 typedef struct {
2     int type;
3     int idMap;
4     int nbJoueurMax;
5     int game_choice;
6     int join_game;
7 }game_request_t;
```

Cette structure comprend une premier champ *type* qui va recueillir le choix de l'utilisateur concernant sa volonté de créer ou rejoindre une partie. Dans le cadre de la création, il devra choisir une carte sur laquelle jouer ainsi que le nombre de joueurs attendus pour que la partie se lance. Les champs *idMap* et *nbJoueurMax* servent à conserver ces choix pour les transmettre au serveur. Le client créant la partie a le droit de rejoindre sa partie directement ou il peut revenir au menu principal. Dans cette optique, le champ *join_game* contiendra le booléen de sa décision. Enfin, *game_choice* est utilisé dans le but d'identifier la partie que le joueur souhaite rejoindre.

Création d'une partie

Comme indiqué ci-avant, lors de la création d'une partie, le client doit choisir la carte sur laquelle il veut jouer. En outre, le serveur lui envoie une structure *info_map_t* pour lui permettre de faire son choix.

```
1 typedef struct {
2     int nbMap;
3     char nomsMaps[NB_MAP_MAX][MAX_CHAR];
4 }info_map_t;
```

Cette structure comprend simplement le nombre de cartes disponibles ainsi qu'un tableau avec le nom de ces dernières.

Rejoindre une partie

Pour terminer cette partie détaillant les structures, voyons la dernière que le serveur utilise dans le but d'informer le client sur les parties existantes grâce à *info_game_t*.

```
1 typedef struct {
2     int nbPartie;
3     partie_t tabPartie[NB_PARTIES_MAX];
4 }info_game_t;
```

Ici encore, la structure fournit des informations sur le nombre de parties actuelles ainsi qu'un tableau contenant des informations sur ces parties. Ces informations provenant de la structure *partie_t* aillant été

décrites dans la section concernant le serveur, nous n'en reparlerons pas ici.

```
1 typedef struct {
2     int id_joueur;
3     int touche;
4 }deplacement_t;
```

Pour recueillir les déplacements du joueur, la structure *deplacement_t* comprend la touche sur laquelle il a appuyé ainsi que son identifiant afin que le serveur effectuer les opérations et vérifications nécessaires au bon fonctionnement du jeu.

Maintenant que nous savons quelles sont les données qui transitent entre le client et le serveur, nous pouvons nous pencher sur la communication elle-même à l'aide des protocoles UDP et TCP.

Les différents échanges

Dans ce projet, nous utilisons deux protocoles différents et chacun à son rôle à jouer dans la mise en place de la communication entre le client et le serveur.

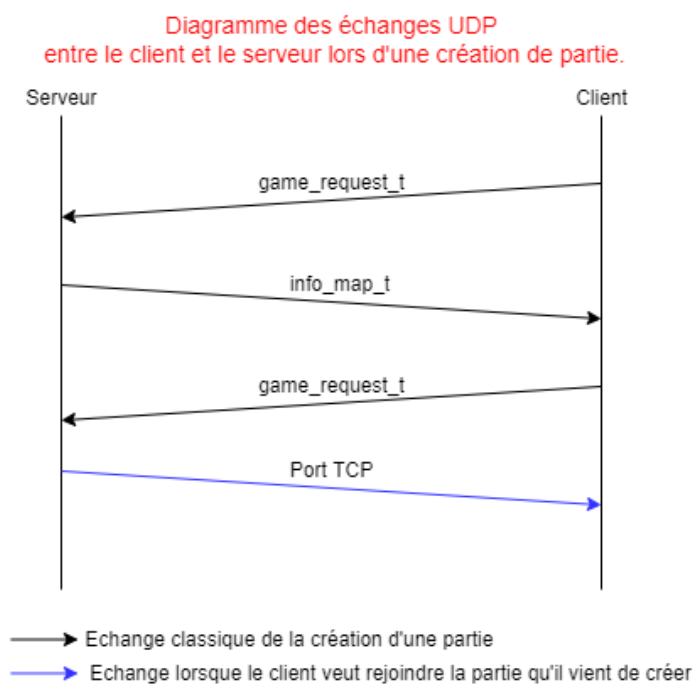
Le protocole UDP

L'utilisation du protocole UDP a pour but de permettre au joueur d'échanger avec le serveur concernant ses choix dans le menu principal. De cette manière, le client peut aisément envoyer ou recevoir des données. Le serveur peut quant à lui fournir des informations à l'utilisateur sans avoir à gérer de multiples connexions grâce à ce protocole.

Etant donné que le client a la possibilité de créer ou rejoindre une partie lorsqu'il est dans le menu, nous ferons un diagramme pour chacune de ces options. Toutefois, le protocole TCP faisant partie intégrante du processus d'accès à une partie, les échanges liés à ce protocole ne seront pas détaillés dans cette sous-section mais le sera dans celle lui correspondant.

Avant de pouvoir échanger, les deux interlocuteurs doivent créer une socket en mode non-connecté sur le même port d'écoute. Le serveur crée une socket qu'il va nommer à l'aide de l'appel système *bind* après avoir définie son adresse. De son côté le client n'a besoin que de créer une socket et de s'associer à l'adresse du serveur. L'adresse du serveur et son port sont passés en arguments lors du lancement du clients. La communication peut ensuite avoir lieu.

Création de partie

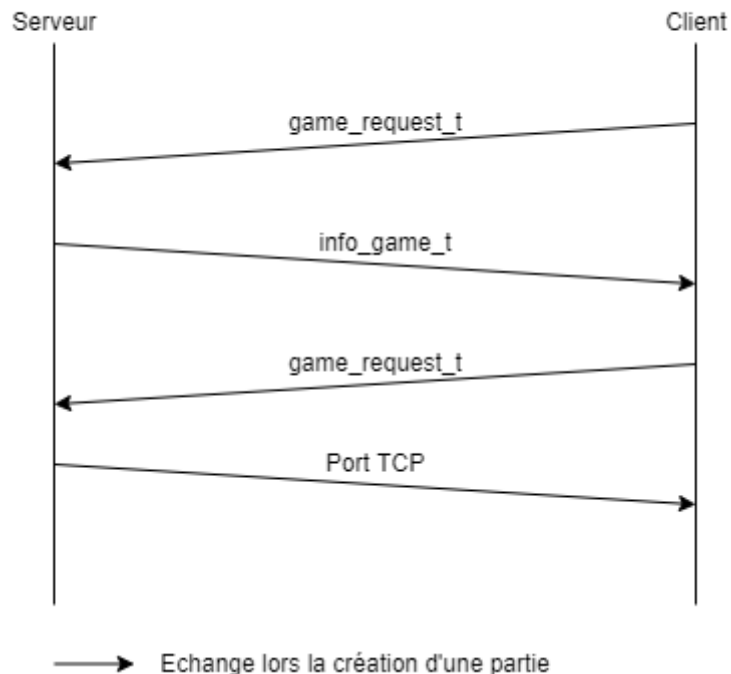


Le client est le premier à envoyer un message qui contient son choix par rapport à s'il veut rejoindre ou créer une partie. Cela passe par la structure *game_request_t* dans laquelle il va remplir le champ *game_type* et tous les autres sont initialisés à -1. De cette façon le serveur peut savoir qu'il doit envoyer des informations sur le nombre de map dont il dispose ainsi que leurs noms. Ces informations sont renseignées dans la structure *info_map_t* avant d'être envoyé à l'utilisateur. Lorsque ce dernier va revoir la structure il aura la possibilité de choisir la carte sur laquelle il veut jouer et aussi le nombre de joueur que contiendra la partie. Ses choix seront enregistrés dans les champs *idMap* et *nbJoueurMax* de la structure *game_request_t*. Avant de renvoyer la structure, le joueur a la possibilité de rejoindre sa partie sans repartir dans le menu principal. Sa décision est conservée dans le champ *join_game* qui permet au serveur d'entamer directement la procédure pour que le client rejoigne sa partie en lui

envoyant le port TCP associé à sa socket qui permettront de communiquer durant la partie. Cette particularité n'existe que si le client choisit de rejoindre sa partie sans retourner dans le menu principal.

Rejoindre une partie

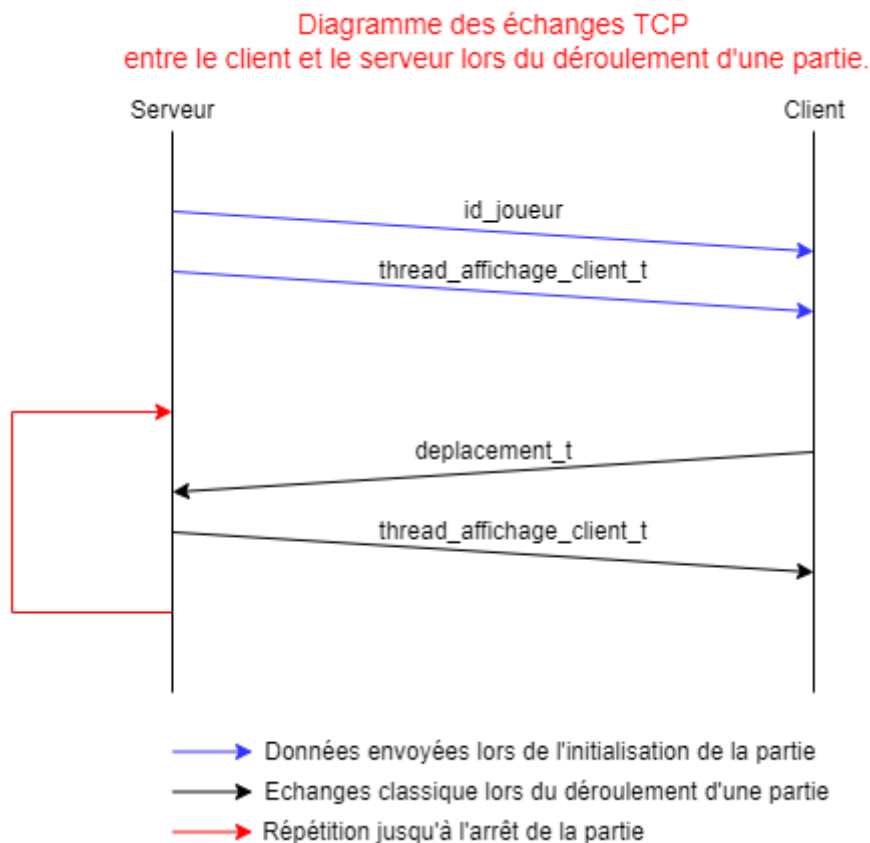
Diagramme des échanges UDP
entre le client et le serveur lors de l'accès à une partie.



Ce diagramme est sensiblement similaire aux échanges produits lors de la création d'une partie à quelques différences près. Le premier envoi du client est le même que dans le diagramme précédent, seule la valeur de *game_type* est différente pour correspondre à une requête d'accès à une partie. A la réception de cette requête, le serveur envoie une structure *info_game_t* contenant dans ses champs le tableau des parties ainsi que le nombre de parties actuellement en cours. L'utilisateur peut donc choisir une partie et cette décision sera enregistrer dans le champ *game_choice*. A la condition que l'identifiant de la partie soit valide ou que la partie ne soit pas déjà pleine, le champ *join_game* deviendra après que la structure ait été envoyé au serveur. Une fois la nouvelle requête reçu, le serveur récupère le port TCP lié à la partie puis l'envoie au client qui peut alors créer une socket TCP et se connecter à celle du serveur.

Le protocole TCP

L'utilisation de ce protocole est liée à la communication entre le client et le serveur lorsqu'une partie est en cours. La création de la socket TCP se fait au moment de la création de la partie. A ce moment-là, le serveur laisse le système d'exploitation choisir le numéro de port. Il y a donc une socket TCP par partie. Le diagramme suivant représente les échanges qui se produisent lors de l'initialisation d'une partie ainsi que ceux qui interviennent durant cette dernière :



Une fois qu'un joueur décide de se connecter à une partie, il crée une socket TCP comme expliqué précédemment. Dès que la socket est prête et que le nombre de joueurs requis est au complet, le serveur démarre l'initialisation de la partie en envoyant à chacun des clients l'identifiant dont ils disposeront via la variable *id_joueur*. Une structure est envoyée à la suite de l'identifiant pour que le client puisse recevoir la carte et le tableau de joueur. Lorsque l'initialisation est terminée, le client et le serveur vont communiquer constamment tant que la partie n'est pas terminée du côté du joueur. En effet, à chaque tour le joueur envoie le déplacement qu'il souhaite effectuer puis le serveur lui renvoie son action, ainsi que toutes les autres qui se sont passées au même moment. Ce déplacement est capturé dans le champ *touche* et les actions sont correspondantes à la réception de la carte et des tableaux de toutes les entités mouvantes. Lorsque le joueur souhaite quitter pour une quelconque raison, cela transite toujours grâce au champ *touche* de *deplacement_t* et le thread correspondant au joueur du côté serveur pourra donc lui aussi s'arrêter. Lorsque les deux s'arrêtent, ils referment leurs sockets TCP respectives.

En parallèle des différentes communications TCP qu'il peut y avoir selon les parties en cours, les communications UDP ne sont pas pour autant interrompues. En effet, puisque la socket TCP est rendue non bloquante lors de sa création, le serveur a toujours la main pour continuer à recevoir et traiter les requêtes de création ou d'accès à une partie par de multiples clients.

V- Remarques sur le projet

Pour réaliser ce projet, nous nous sommes organisés en séparant le projet en 3 parties : l'éditeur, les connexions entre les clients et le serveur, et l'interface client avec la gestion des règles du jeu.

Nous avons commencé notre projet à partir du TP1, en faisant l'interface de l'éditeur, ainsi que le placement des différents éléments sur un level. Nous avons par la suite, après le TP3, réalisé la partie de l'enregistrement/suppression des levels dans un fichier binaire, accompagner de la table d'adressage. Une fois l'éditeur fini, nous avons coupé le travail en deux pour pouvoir travailler en même temps sur le projet, mais sur des tâches différentes. Grâce à cette technique, nous avons pu travailler à la fois sur la connexion du client avec le serveur (pour la création des parties en l'envoi de donnée pendant la partie), et à la fois sur l'interface client et la gestion des règles du jeu. Cette organisation nous permet de chacun travailler sur une fonctionnalité sur sa branche, puis une fois la fonctionnalité fonctionnelle, nous pouvons l'implémenter au code de la branche principale qui contient le projet définitif, tout cela sans poser de problème à l'autre personne qui travaille sur sa fonctionnalité.

Tout le code est bien évidemment commenté, afin que comprendre le sens de chaque fonction et de modifier facilement le bon morceau de code si besoin. Les variables, fonctions et structures utilisées possèdent des noms intuitifs qui décrivent les données qu'elles contiennent ou une description de ce à quoi elles servent. Aussi, nous avons vérifié le retour de tous les appels systèmes et fait la gestion de erreurs pour chacun d'entre eux.

Difficultés

Les principales difficultés rencontrées se sont produites durant les phases de communication entre le client et le serveur. Tout d'abord, la première difficulté a été de réussir à structurer intelligemment le code du serveur et du client au moment de la communication UDP afin que le serveur puisse gérer plusieurs clients à la fois sans pour autant utiliser de threads ou de processus fils.

Lorsque nous avons commencé le code client/serveur, nous sommes parties vers l'idée de faire un tableau de joueur, de probe et de robot, ce qui nous permettait d'avoir des levels qui ne contenaient pas d'entité qui se déplaçait et donc de simplifier le code au niveau des accès à une case. Nous nous sommes rendu compte, qu'avec cette technique cela empêchait une parallélisation correcte de l'application. Le fait d'utiliser un tableau pour les robots nous oblige donc, pour la vérification de l'explosion de la bombe, de parcourir le tableau pour vérifier les positions de chaque robot. Le fait de parcourir un tableau signifie séquentialisation du programme. Par manque de temps et étant donné que tout le reste de notre projet est complet et fonctionnel, nous avons décidé de laisser cette petite erreur, qui pourrait être corrigé en insérant des listes de robots/probes/joueurs dans chacune des cases du level.

Améliorations

Afin d'améliorer notre jeu et de lui faire avoir le même succès que Fortnite en 2018, nous pourrions lui ajouter toutes sortes de fonctionnalités pour rendre l'expérience utilisateur plus amusante comme des nouveaux objets, des nouveaux éléments ou encore des nouveaux monstres. Actuellement, notre jeu est uniquement jouable en local, c'est-à-dire connecter sur le même réseau. Nous pourrions modifier cela pour le rendre jouable en multijoueur en ligne, mais nous ne possédons actuellement pas les connaissances nécessaires pour le mettre en place.

Au niveau de la gestion des parties, il serait possible de faire qu'un joueur aillant quitter une partie puisse la rejoindre à nouveau à condition que cette dernière soit toujours en cours. Par ailleurs, il n'est actuellement pas possible de lancer plus de vingt parties. Nous pourrions faire en sorte que plus de parties soient disponibles en même temps. Aussi les parties pourraient être supprimées du tableau de parties et donc de l'affichage une fois que celles-ci sont terminées.