Rapport Projet

INFO0601 / INFO0604

Jeux de plateformes multijoueurs avec éditeurs de niveaux

LEVALET Corentin et COLLIGNON Alexandre

2022-2023 Licence 3 INFO S6

Bonne lecture 🙂

Sommaire

Présentation

Ce rapport présente les différentes étapes de notre travail effectué sur le projet d'INFO0601/0604. Le projet consiste en 3 exécutables :

- L'éditeur de niveau.
- Le client joueur
- Le serveur de jeu

Pour simplifier la compilation de notre projet, nous avons utilisé la compilation séparée grâce à un Makefile de notre fabrication basé sur le Makefile multi-répertoire de M. Rabat. À la différence du Makefile de M. Rabat, le nôtre récupère lui-même le nom des ".o" en scannant les répertoires du projet, cela permettant donc d'organiser les fichiers du projet en sous-répertoire pour séparer les fichiers en commun du projet et les fichiers spécifiques à un exécutable (client, serveur, éditeur).

Pour commencer ce rapport, nous allons dans un premier temps nous pencher sur la première partie produite : l'éditeur.
Nous verrons dans cette partie les premiers choix d'orientation importants effectués sur la manière d'enregistrer les niveaux.

Dans une seconde partie, nous verrons la mise en place de notre client, dont une importante partie du code est très semblable à celle de l'éditeur. La partie qui diffère étant l'ajout du menu de choix d'une partie et création d'une partie ainsi que l'ajout de la gestion de la communication réseau à la fois en UDP pour une partie et en TCP pour une autre.

En dernière partie, nous analyserons le serveur étant une application multi-processus et multi-threads gérant le fonctionnement des parties (déroulement du jeu), la création des parties et donnant la possibilité aux joueurs de rejoindre les parties.

Bien sûr, pour aider à la compréhension des différents systèmes et échanges mis en œuvre, des schémas, des échanges réseaux (UDP et TCP), entre processus et entre threads, accompagneront les différentes parties du rapport.

Finalement, pour conclure ce rapport, nous nous pencherons sur les difficultés rencontrées durant ce projet.

L’éditeur

L’éditeur est composé de 4 grands systèmes :

- La sauvegarde, chargement des niveaux.
- Le contrôle d’éditions d’un niveau.
- L’affichage d’un niveau.
- L’interface graphique ncurses.

La sauvegarde, chargement des niveaux

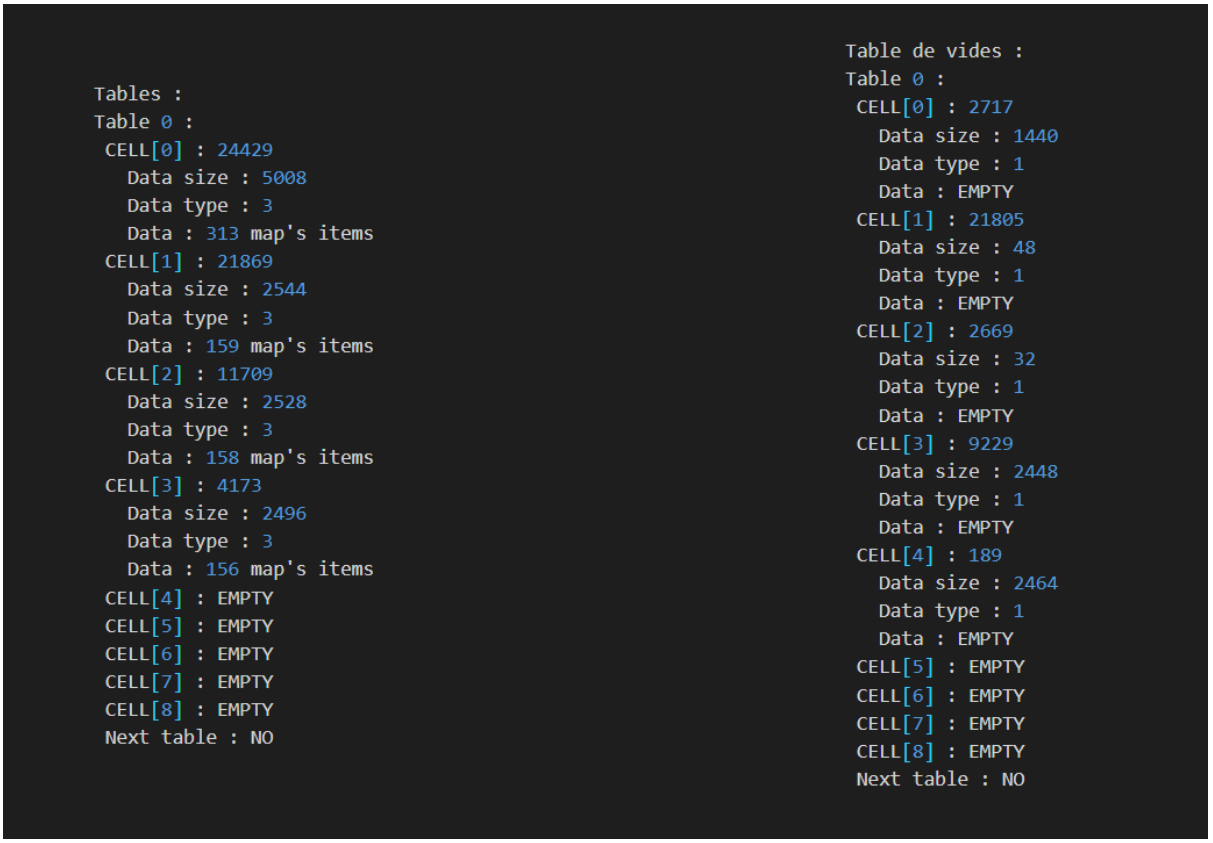
Le système de sauvegarde est décomposé en plusieurs méthodes pour réduire les répétitions, ainsi, nous disposons de méthode telle que : read_data, write_data, update_empty, get_empty, write_table, get_table, write_data_info, get_data_info...

Ces différentes méthodes sont ainsi appelées au sein de 2 méthodes principales : get_level, save_level.

La méthode **save_level** vérifie si un niveau est déjà enregistré dans le fichier, s'il ne l’est pas alors, il l’ajoute. Si le niveau est déjà présent, il le supprime, ce qui aura pour effet de créer un vide qui sera enregistré, ainsi ensuite si le niveau rentre de nouveau dans le trou, celui-ci sera enregistré de nouveau dedans ou bien dans un autre trou suffisamment grand et à défaut à la fin du fichier.

La méthode **get_level**, elle, vient parcourir les tables du fichier jusqu’à l’entrée absolue équivalente au niveau demandé. Si cette entrée existe, elle est lue et le niveau est chargé en mémoire sur l’espace mémoire ou pointe le pointeur passé en argument, au contraire le cas échéant une valeur -1 est retourné par la méthode.

Voici une capture d'écran d'une schématisation de la table d'un fichier de niveaux fait dans notre fichier log.

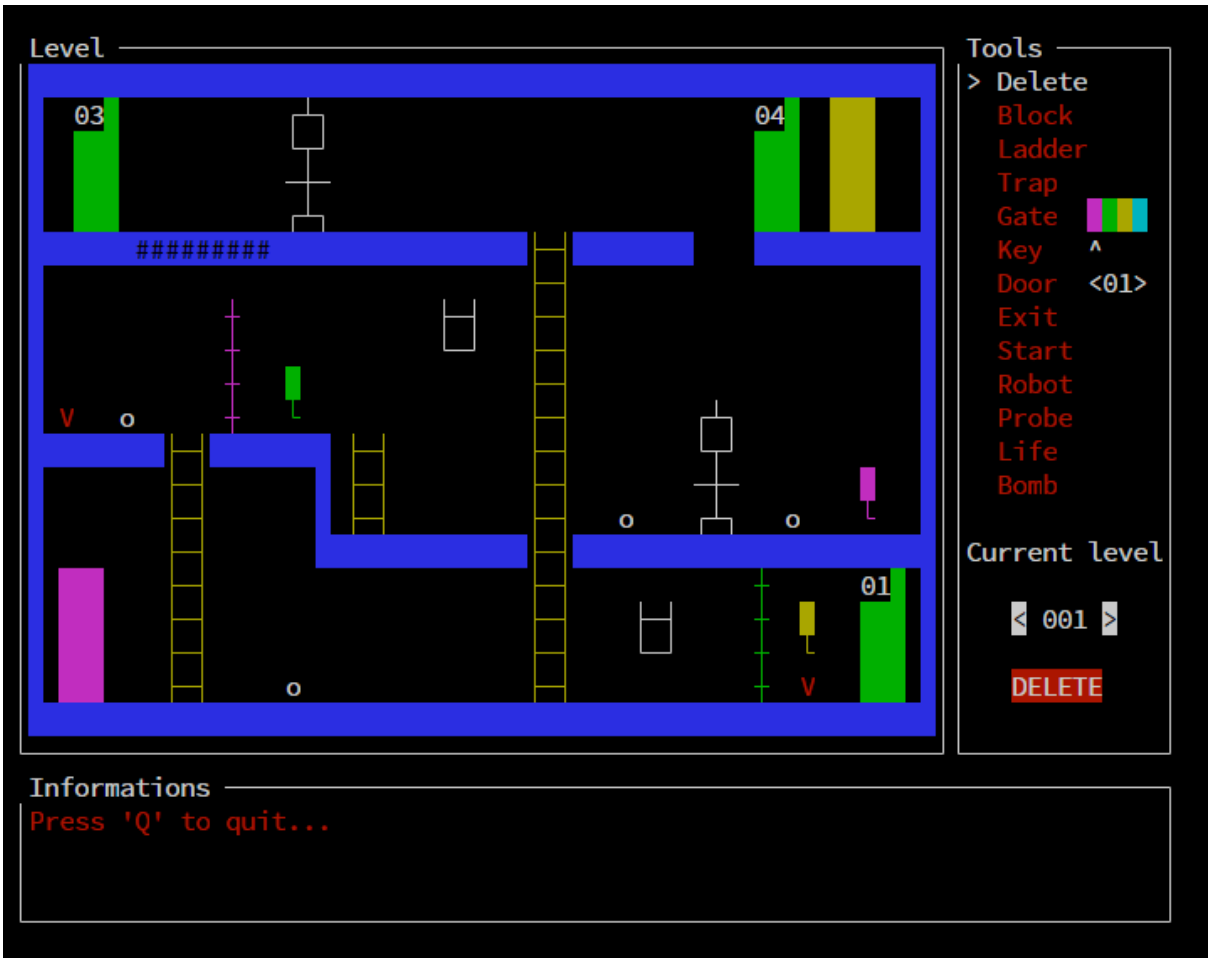


Le contrôle d’éditions d’un niveau

Pour permettre l’ajout d’éléments sur la map d’un niveau, nous avons construit une méthode “checkHitbox” qui nous retourne ainsi si des éléments entrent en collision avec un objet que l’on souhaiterait ajouter sur la map. Nous utilisons cette fonction dans chaque fonction pour poser des éléments qui sont les fonctions appelées quand l’on clique sur l’espace d’édition de l’éditeur.

Suivant la sélection dans le menu de droite, la fonction appelée sera par exemple “poserBloc” quand bloc est sélectionné, “poserRobot” quand robot est sélectionné…

Quand l’outil de “delete” est sélectionné, nous appelons une autre méthode utilisant aussi “checkHitbox” qui supprime l’élément dont la position renseignée est dans la hitbox de celui-ci.



L’affichage d’un niveau

L’affichage d’un niveau est effectué en parcourant la liste des éléments d’un niveau. Ainsi les éléments en dernier dans la liste seront la couche la plus haute.

Nous utilisons un “switch case” qui, suivant la valeur type de notre Objet, permet de différencier ce qu’il faut dessiner et nous dessinons ainsi les éléments avec les variations qu’il nécessite, par exemple la couleur pour une clef en lisant les informations de l’union que l’objet contient.

Quand nous lisons le type joueur par exemple, nous accédons à ses informations de variations via la valeur d’union “.player” ou bien pour les clefs la valeur de “.key”, etc.

Cette méthode d’affichage qui est commune au client permet donc au serveur de choisir l’ordre des sprites suivant l'ordre des éléments dans la liste qu’il envoie au client.

L’interface graphique ncurses

L’interface graphique “ncurses” est composée de 6 fenêtres, car chaque fenêtre dispose d’une sous fenêtre, pour éviter d’effacer la box et devoir la remettre à chaque fois, nous dessinons la box dans la fenêtre et dessinons le “niveau” dans une sous-fenêtre. Ce modèle est ré-utilisé sur le client avec des variations sur le menu de choix de la partie.

Le client

Le client est composé de 3 grandes parties :

- Le système réseau UDP (Choix partie)
- Le système réseau TCP (Fonctionnement partie)
- Le GUI composé du “MENU GUI” et du “GAME GUI”.

Comme déjà expliqué plus haut, une partie du client est basé sur des fonctionnalités déjà écrites pour l’éditeur telles que l’affichage d’un niveau ou bien la construction de l’interface ncurses.

Même si la forme de l’interface ncurses a été reprise, une grosse refonte de son système d’initialisation a été faite pour permettre de simplifier le passage de l’interface du menu de choix, création de partie et l’interface de jeux qui est la partie la plus proche de l’éditeur.

Nous allons désormais voir les différents mécanismes du client dans les parties ci-dessous.

Le système réseau UDP (Choix partie)

La demande de liste de partie

Pour permettre d’afficher la liste des parties sur le menu, nous avons mis en place une requête de liste des parties. Les informations des parties devant passer sur le réseau, cela demande d’utiliser des messages de taille fixe. Nous avons donc opté pour un système de pagination des parties.

Pour obtenir la liste des parties, le client envoie un premier **NetMessage** de type : UDP_REQ_PARTIE_LISTE, dans ce message le **payload** spécifie le numéro de page demandé, le nombre de parties reçu ainsi qu’un tableau de Partie contenant pour chaque partie son état, son nombre de joueurs, le maximum de joueurs et la carte utilisée.

La seule information remplie par le client est bien sûr le numéro de page. En recevant le message, le serveur parcourt sa liste de parties et enregistre les bonnes informations dans le message, soit les parties correspondantes à cette page si celle-ci existe, le nombre d’éléments sur la page, soit un nombre entre 0 et 4.

La façon de savoir si une page est la dernière pour le client est d’envoyer une requête est si la requête retourne avec un nombre de

partie de 0 alors que la page précédente en possède, cela lui indique qu'il était déjà sur la dernière page.

Cette requête est donc exécutée lors de l'appui sur les boutons de pages suivantes, ou bien de pages précédentes (ou les flèches gauche et droite du clavier), ou bien de rafraîchissement. À la différence que le numéro de pages demandé est différent suivant le bouton appuyé.

La demande de création de partie

Pour permettre de créer une partie, nous utilisons un **NetMessage** de type UDP_REQ_PARTIECREATE avec un **payload** contenant le nombre de joueurs maximum choisi, et le numéro de la carte choisie, si la requête a été un succès et le numéro attribué à la partie. (Le détail de la requête est précisé dans la partie serveur)

Le client lui bien sûr ne remplit que le nombre maximum de joueurs , le numéro de la carte choisie. Le serveur en répondant au message celui-ci complète l'autre partie avant de renvoyer le message permettant au client d'afficher correctement dans la liste des parties la partie qu'il vient de créer et la marquer comme "en attente de démarrage" pour permettre au joueur de quitter l'attente de la partie s'il le souhaite.

La demande pour rejoindre/quitter l'attente d'une partie

Quand un joueur n'est pas responsable de la création d'une partie, il peut décider de rejoindre une partie en attente en cliquant sur le bouton rejoindre dans la liste.

Cela envoie une requête au serveur (Le détail précisé dans la partie serveur) qui ajoute, si possible, le joueur à la liste d'attente de la partie. Une fois la réponse positive du serveur reçue, le client déclenche un thread parallèle d'écoute en UDP pour attendre le message du serveur provisionnant le port du serveur TCP du jeu lorsque celui-ci démarrera.

Le joueur peut bien entendu appuyer sur le bouton rejoindre qui affiche désormais annuler s'il souhaite quitter la liste d'attente de la partie. (Cette requête est aussi explicitée dans un schéma dans la partie serveur).

Lors de l'attente d'une partie, seul l'action de rafraîchissement et pour quitter l'attente sont possibles.

Comme les messages UDP sont écoutés par un autre thread lors de l'attente d'une partie, la réponse est reçue par l'autre thread et est ensuite stockée dans une structure de donnée commune entre les threads de l'application client. Le thread principal se met en pause et c'est le thread d'écoute de l'UDP qui se charge de réveiller le thread principal une fois le message reçu est stocké dans la mémoire.

La demande de la liste des cartes de jeu du serveur.

Cette requête est complètement similaire à celle de la liste de partie à la différence que nous récupérons le nom des cartes de jeu, leur numéro et nom des parties. Cela permet au joueur de choisir la carte lors de la création d'une partie.

Nous disposons ici par compte que d'un bouton de page suivante et précédente, car l'action de rafraîchissement n'est pas vraiment utile dans ce cas.

Le système réseau TCP (Fonctionnement partie)

Une fois une partie rejoint, une connexion TCP est établie, ainsi le client se retrouve décomposé en 2 threads, un thread de réception de message TCP du serveur et un thread pour envoyer les inputs du client au serveur.

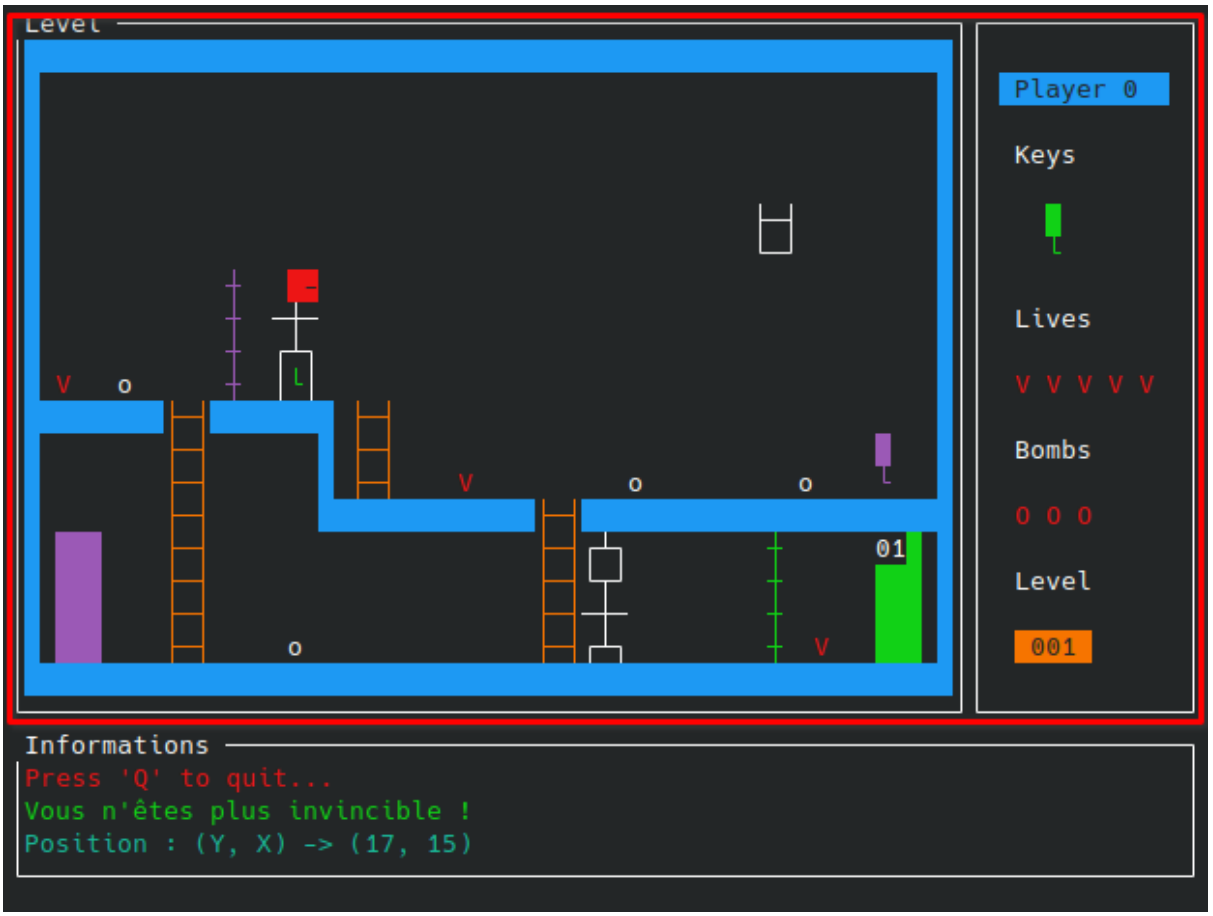
Nous allons voir ci-dessous les 3 types de messages. Soit 2 types de messages recevables et 1 type de message adressable au serveur.

Réception messages de mise à jour niveau et infos joueur

La réception de ce message se fait par la méthode de routine du thread de réception TCP. Au sein de cette méthode, suivant le type du **NetMessage** nous différencions le type de **payload** à traiter et l'entrons en arguments à la méthode de mise à jour des infos de partie. Cette méthode convertit le niveau qui est stocké en un tableau vers un objet "Level" compréhensible pour notre méthode d'affichage de niveau. La méthode remplace aussi les informations de la structure des informations joueurs par celle reçue dans le message et elle finit en lançant les méthodes de rafraîchissement des interfaces (L'interface du jeu et celle d'informations joueur à droite).

Après cela, le traitement est terminé et le thread boucle un tour de boucle pour attendre de nouveau, un nouveau message.

Capture d’écran de la zone mise à jour :



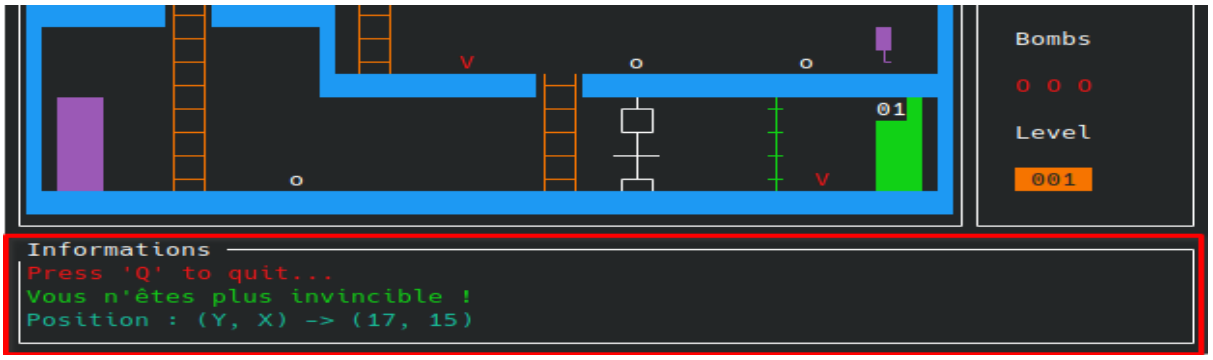
La zone en rouge est la zone mise à jour par ce type de message.

Réception des messages d'informations

La réception de ce message se fait comme pour celui précédent à la différence que celui-ci dispose d'un type différent dans le **NetMessage** ce qui a pour effet de le faire traiter par une autre fonction qui met à jour la ligne indiquée par le message dans son payload, par le texte fourni et la couleur. NB : seule la dernière ligne est mise à jour par le client (pas par le serveur), car elle nous sert de débogage.

Après cela, le traitement est terminé et le thread boucle de nouveau pour attendre un nouveau message à traiter.

Capture d'écran de la zone mise à jour (Zone en rouge) :



Envoie des entrées utilisateurs du joueur

À chaque fois que le joueur presse une touche de son clavier, nous envoyons celle-ci dans un **NetMessage** au serveur avec un type pour valeur la constante "TCP_REQ_INPUT_PLAYER".

Ainsi, il ne s'agit donc que d'une boucle sur le retour de getch() et nous envoyons directement le message avec la touche en **payload**.

Nous avons aussi ajouté un petit "usleep" de 0.1 seconde après l'envoi de la requête pour temporiser l'entrée utilisateur et éviter de spammer le serveur. Mais l'utilisation d'un "sleep" demande de dump le cache du "getch()" sinon la touche envoyée au prochain tour de boucle ne sera pas la bonne, car "getch()" dispose d'un buffer.

Le reste du traitement de l'entrée de l'utilisateur s'effectuant sur le serveur, nous en reparlerons dans sa partie dédiée.

Mais à ce niveau, nous pouvons ainsi constater que nous avons décidé de baser nos échanges sur des échanges sans réponse ni demande du serveur ou du client, car en utilisant le TCP, la garantie d'arrivée de l'information est gérée par une couche système plus haute que nous.

L’Interface graphique utilisateur

Le GAME GUI : affichage du niveau et des infos joueurs

L’affichage du GUI du jeu est composé en deux parties étant donné que la partie informations du bas et commune au deux GUI (Game et Menu).

Nous disposons d’une méthode pour initialiser les données affichées dans les GUI, soit une pour la partie jeu (la carte, le niveau) et une pour la partie de droite (les informations du joueur). Nous disposons aussi de méthodes pour rafraîchir les GUI, soit ainsi redessiner les deux parties avec les informations dans les structures de données. Ces méthodes sont celles utilisées par le thread réseau après la mise à jour des structures effectuée.

Nous disposons aussi d’une fonction permettant de détruire correctement chaque partie du GUI pour libérer correctement le mémoire alloué dynamiquement. Ces fonctions sont appelées à la fermeture du jeu, mais aussi lors de l’appel de la fonction “switch_gui” permettant de passer du GUI MENU au GUI GAME.

Le MENU GUI : choix, création de partie

Ce GUI est composé, de 2 sous GUI, le choix d’une partie et celui de création d’une partie. Ces deux GUI disposent, à la différence des autres GUIs de fenêtres ncurses, en plus soit 4 pour les 4 onglets maximum de choix de partie sur une page et 6 pour les 6 cartes maximum affichages par page sur la création de partie.

Ayant plus de fenêtre, nous avons donc une fonction spécifique pour libérer correctement la mémoire de ce GUI et nous différencions les deux GUI grâce à un booléen stocké dans notre structure de données principal du jeu. Cela permet ainsi à notre fonction de fermeture principale du jeu de savoir quelle méthode de libération de mémoire appeler.

Comme déjà vu, lors de l’explication des requêtes UDP, nous avons ainsi un bouton de rafraîchissement, un système de pagination, un bouton pour accéder à la création de partie. Et lors de l’affichage du menu de création, nous disposons aussi d’un bouton d’annulation permettant de retourner sur le menu de choix sans valider de création de partie.

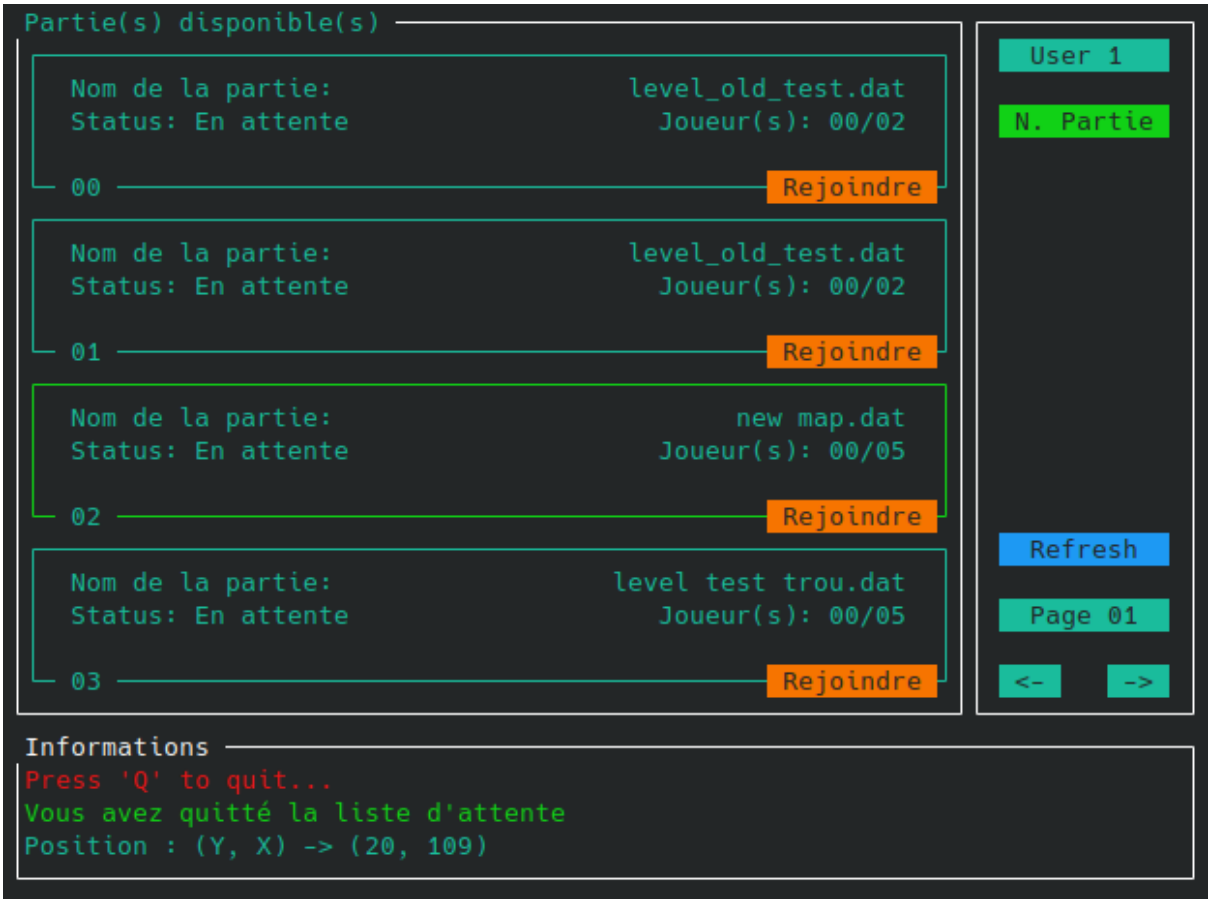
Le choix d'une carte sur le menu de création se fait grâce aux flèches, UpArrow, DownArrow et le changement de page peut être effectué par les flèches LeftArrow, RightArrow. (Changement de pages vaut pour le choix de partie et la création de partie).

Le choix du nombre de joueurs maximum dans une partie se fait en cliquant sur les boutons flèches sous le nombre de joueurs max affiché sur le menu de droite.

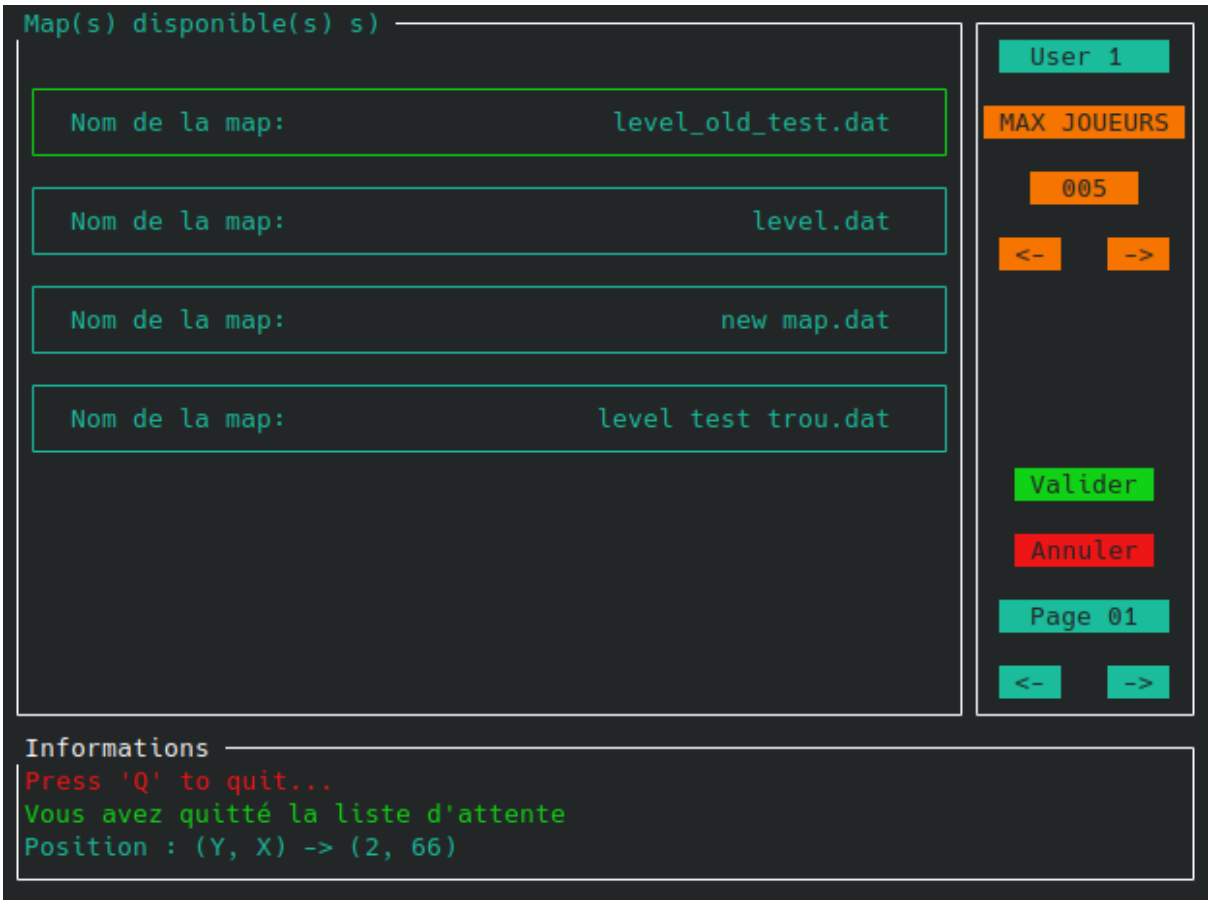
Le choix d'une partie peut se faire par entrée clavier et entrée souris. Pour valider un choix via le clavier, il suffit de presser la touche "ENTRER" et via la souris de cliquer sur le bouton "rejoindre" ou "annuler" lorsque nous sommes déjà en attente.

Pour conclure cette partie d'explication du client, voici une capture d'écran de chacun des menus. Le choix de partie et la création de partie :

Le menu de choix de partie :



Menu de création d’une partie :



Le serveur

Le serveur est la partie la plus complexe de notre projet, celle-ci est composée de plusieurs processus et threads permettant de gérer harmonieusement les différents mécanismes de notre jeu allant du système de gestion des parties à la gestion des mécanismes de jeu au sein d'une partie.

Gestion des parties

La première partie du serveur est un gestionnaire de partie appelable par réseaux via Datagramme UDP. Les différentes requêtes auxquelles le gestionnaire de partie peut répondre sont :

- L'obtention d'une page de la liste des parties.
- L'obtention d'une page de la liste des cartes disponibles.
- La création d'une partie.
- Rejoindre ou quitter une liste d'attente d'une partie.
- Simple ping/pong avec le serveur.

Échange UDP

Ces différentes requêtes sont utilisables en envoyant une structure NetMessage via UDP à l'adresse et port renseigné au client depuis les arguments du programme (Exemple ./client -p 25565 -h 127.0.0.1). De plus, côté serveur, les mêmes arguments peuvent être utilisés pour choisir le port et l'adresse d'écoute pour les échanges UDP. (Exemple ./serveur -p 25565 -h 0.0.0.0). Notons que l'argument -h n'est pas obligatoire.

Un **NetMessage** est composé d'un attribut "type" et d'une union comportant les différents **payload** permettant au serveur de distinguer quelle entrée de l'union utiliser, les entrées de l'union étant au même nombre que le nombre de requêtes différentes possibles (cf. liste au-dessus) à l'exception du ping ne nécessitant aucun **payload**.

Dans l'optique de permettre l'administration de multiple partie, le serveur se compose de plusieurs processus. Nous distinguons 3 types de processus :

- Processus d'entrée utilisatrice (Console de commandes)
- Processus d'écoute UDP (Le gestionnaire de parties)
- Processus de gestion d'une partie. (Gestion d'une partie)

Processus d’écoute UDP

Celui-ci permet la réception et le traitement des requêtes UDP de gestion des parties vues ci-dessus. Dans ce processus, nous exécutons une méthode spécifique pour chaque requête suivant le type renseigné dans le **NetMessage**. La fonction s’occupe d’effectuer les actions nécessaires telles que la récupération de la liste de parties, de la liste des cartes, ou bien encore la création d’une partie.

La fonction pour rejoindre une partie se distingue des autres dans le sens où elle est capable d'envoyer plusieurs messages en plus de la réponse à sa requête initiale, car celle-ci est chargée d’envoyer la réponse UDP aux différents clients en attente du début de la partie.

Dans le cas d’une partie avec seulement 1 joueur, c’est la fonction de création qui appellera la création du processus de la partie, mais dans tous les autres cas cette tâche incombe à la fonction gérant les requêtes de “JoinWaitListPartie” car comme expliqué dans les consignes, la création du socket TCP d’une partie est fait que lorsque tous les joueurs nécessaires selon la configuration d’une partie est atteint.

Pour les réponses aux requêtes, les différentes fonctions du serveur complètes le type **payload** du **NetMessage** reçu et retourne le nouveau afin d’être envoyé, parce que nous utilisons la même structure pour une requête et sa réponse, la différence entre le client et le serveur se trouve seulement dans les champs que l’un et l’autre remplissent tel un formulaire que l’on se passerait.
(aka. Nos conventions de stage =))

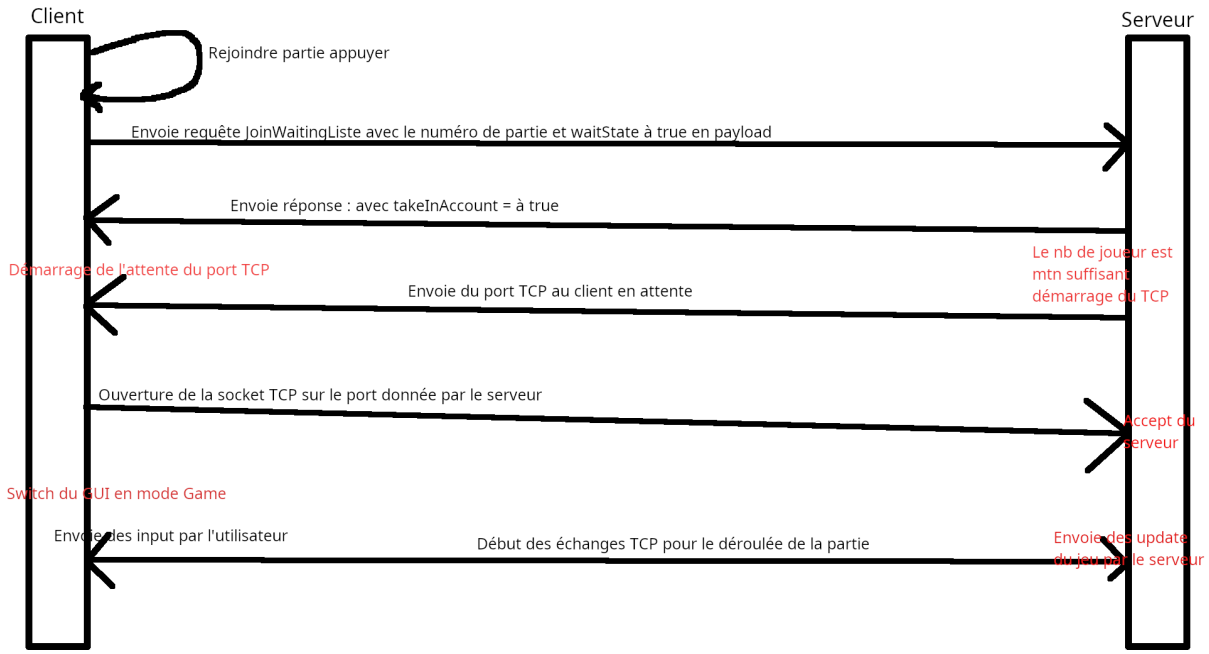
Cela est à peu près tout pour la partie UDP du serveur. Voici ci-dessous quelque schéma d’échanges réseau entre le client et le serveur pour permettre une meilleure compréhension du déroulement d’une requête.

Schémas de déroulement des requêtes UDP

Retrouvez sur les pages suivantes deux exemples de déroulement de requête UDP, de la réception de la requête via l'appel système "recvfrom" sur le socket UDP de notre handler UDP, au renvoi de la réponse complétée via l'appel système "sendto" dans le même handler.

Requête "JoinAndLeaveWaitingList"

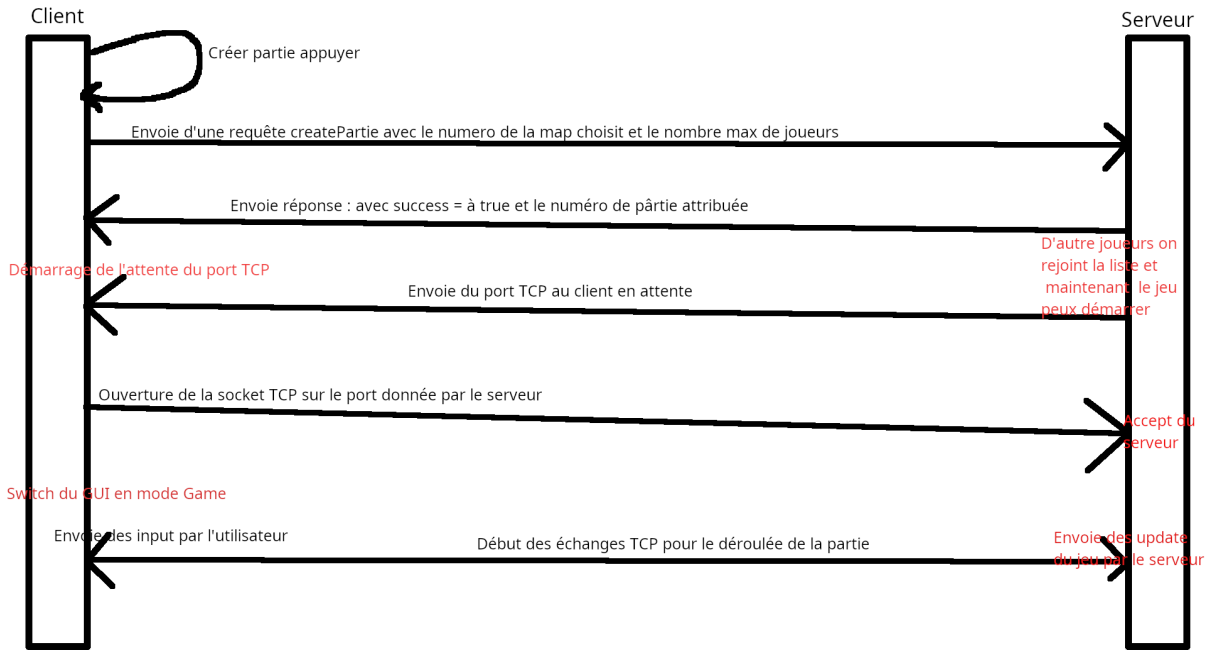
Voici un schéma du déroulé interne entre le client et le serveur lorsqu'un joueur rejoint une liste d'attente d'une partie. (Oui veuillez zoomer un peu désolé, nous avons des problèmes pour créer une image lisible)



Comme nous pouvons voir, le client envoie une requête pour rejoindre une partie, comme le serveur n'est pas encore allumé, le serveur lui retourne un message de bonne réception et ajoute le joueur dans sa liste d'attente. Une fois la liste d'attente pleine, le serveur démarre la partie et retourne une réponse à tous les clients s'étant mis en attente. Les clients en réponse du port lancent une connexion TCP sur ce port et le serveur l'accepte. Cela Switch le client en mode "Game" et les échanges ne se font plus qu'en TCP. Ces échanges sont vus dans la suite du rapport.

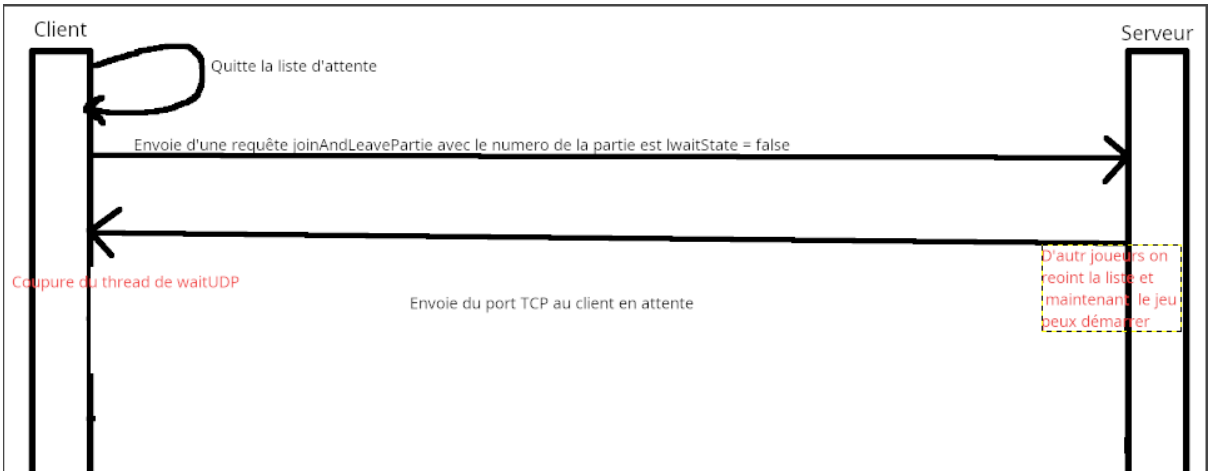
Requête “CreatePartie”

Voici le schéma de création d’une partie :



Comme nous pouvons voir, ce schéma est similaire au schéma précédent sauf pour la première requête où le client précise la map choisie et le nombre max de joueurs voulu. En réponse, le serveur donne en plus le numéro de partie attribué.

Voici pour finir la requête quand un joueur quitte une liste d’attente :



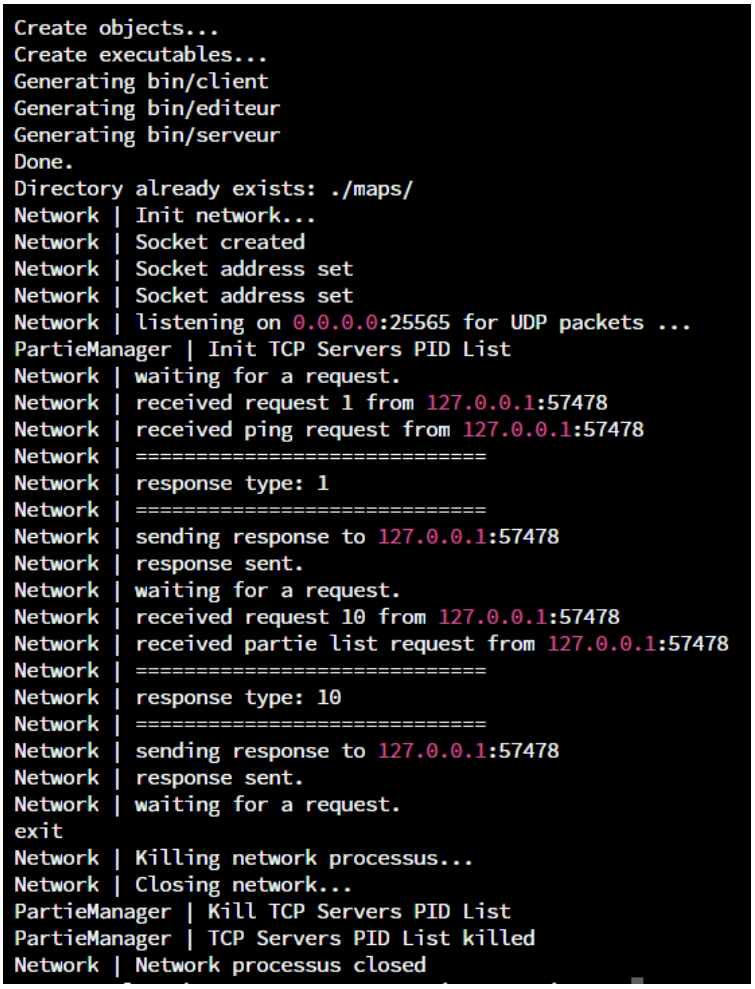
Processus d’entrée utilisatrice

Ce processus permet actuellement d’exécuter l’arrêt du serveur, mais dans l’optique d’amélioration future, nous pouvons imaginer d'autres commandes de maintenance telle que la visualisation de la liste des parties depuis la console ou bien d'autres.

Celui-ci consiste donc en l’utilisation d’une simple boucle “while” qui répète la demande d’une commande et exécute celle-ci. Ayant seulement la commande “exit” pour le moment, celle-ci change donc juste la condition de boucle du serveur à “faux” et utilise l’appel système “kill” sur les autres processus puis “waitpid” sur le processus de gestion de partie qui dispose d’un “sigaction” enregistrée gérant la fermeture des différents processus de partie déjà ouverts en leur propageant le signal “SIGINT”.

En plus de la commande “exit” le “CTRL + C” est utilisables proprement grâce aux “sigaction” enregistrée sur les différents processus.

Capture d’écran de la console du serveur de jeu :



Processus d’une partie

Le 3ème type de processus est lui divisée en plusieurs threads, car il s’agit du processus de gestion d’une partie comportant donc plusieurs mécaniques de jeu pouvant être mis en parallèle, mais nécessitant un partage de mémoire, ainsi l’utilisation de threads permet un partage de mémoire plus facile qu’avec l’utilisation du segment de mémoire partagée, voici les différents threads présent :

- Thread principal d'envoi de mise à jour de la carte de jeu
- Threads de gestion des mobs, comme demandée dans les consignes chaque mob dispose de son propre thread.
- Thread de gestion des pièges.
- Threads de gestion d’input d’utilisateurs. (Un par joueur).
- Threads de réapparition des loots (Vie, Bombe).
- Threads des bombes lâchées par les joueurs (Un par bombe, car considérée comme toute autre mob).

Thread principal d’envoi de mise à jour de la carte de jeu

Ce thread est a pour objectif tout d’abord de charger les différentes informations nécessaires pour le déroulement de la partie, il démarre donc les autres différents threads tels que les threads des mobs, celui des pièges et les threads d’échanges avec les joueurs (Ouverture des sockets TCP : fonction “accept”).

C’est une fois tout paramétré que celui-ci déclenche une boucle attendant sur une variable de condition pour exécuter l’envoie de la carte mise à jour aux différents utilisateurs. En plus de charger les mobs, le thread récupère toutes les portes pour former les couples de portes correspondantes.

La boucle de mise à jour a pour condition d’arrêt l’état de la partie qui est mis à jour par la déconnexion des joueurs. Ainsi, lorsqu’il ne reste plus de joueurs en jeu, la boucle s’arrête et la fonction continue sur le code d’arrêt des différents threads, des mobs, des pièges. Ainsi que sur la libération de la mémoire allouée dynamiquement.

Exemple d’exécution de la boucle d'envoi de mise à jour de la carte :

- Réveil du thread par une variable de condition.
- Parcours par une boucle “while” de la liste des niveaux.
- Envoi de la liste des objets de la carte aux joueurs dans le niveau (donc les joueurs ne reçoivent que leur niveau).

- Mise en pause du thread sur la variable de condition après le parcours de tous les niveaux.

Threads de gestion des mobs

La gestion des mobs est effectuée par un thread pour chaque mob. Pour instancier les threads des mobs, nous créons au chargement de la partie une liste des mobs du jeu dans laquelle nous passons une structure qui contient un pointeur vers l'Objet de la carte, le pthread_t du thread du mob, ainsi qu'un pointeur vers la structure du niveau où est le mob. Depuis cette liste, nous lançons une méthode qui déclenche, suivant le type du mob, le thread de routine adéquate. Nous allons voir ci-dessous le fonctionnement de la routine des différents mobs.

Les robots

La routine d'un robot est composée d'une boucle s'exécutant toutes les 0.5 secondes pour éviter de rendre le mob trop rapides. Cette routine consiste pour le robot à vérifier qu'il n'y a pas devant lui d'objet non traversable (Pièges à ses pieds, blocs, ou bien un portail) Pour détecter tout cela, nous utilisons une méthode qui nous retourne les objets dans une zone rectangulaire définie, ainsi, nous utilisons cette méthode pour sonder la nouvelle position que le mob prendra (plus une rangée de plus sous le mob pour voir les trous et les pièges) et si nous trouvons en parcourant la liste un piège ou bien un bloc ou un portail, nous empêchons le robot d'avancer et nous changeons son sens de marche pour la prochaine itération de la boucle. Une fois le mouvement validé ou non, nous vérifions sur la hitbox du robot si un joueur s'y trouve. Si un joueur est trouvé dans la hitbox, nous faisons attaquer le robot et affligeant donc la vie en moins au joueur et appliquant l'invulnérabilité pour 3 secondes au joueur, si le joueur se retrouve sans vie alors l'on déclenche aussi la routine de mort du joueur. Quand la variable d'état de freeze du robot est mise à jour, celui-ci se met en pause (attente passive) jusqu'à la fin du freeze et redémarre sa routine normalement en se réveillant.

Les probes

La routine d'une probe est presque similaire au robot à la différence que nous tirons un nombre entre 0 et 3 aléatoirement pour choisir la nouvelle direction de la probe et nous sondons de la même façon que le robot pour infliger les dégâts au joueur et vérifier

aussi si la nouvelle position est correcte. Comme pour le robot, si une bombe touche la probe, celle-ci se met en pause pendant le temps du freeze.

Thread des pièges

Le thread des pièges déclenche les pièges toutes les 3 à 5 secondes et ainsi inflige des dégâts aux joueurs au-dessus et déclenche la routine de mort du joueur si le joueur se retrouve sans vie.

Threads de gestion d'input utilisateurs

Il y a un thread de gestion d'input utilisateurs par joueur pour écouter le socket TCP ouvert entre le serveur et celui-ci.

L'ouverture de ce socket est effectuée dans le thread de mise à jour, vue plutôt durant le paramétrage de la partie. Ce thread s'occupe ainsi de gérer 3 cas :

- Requête de déconnexion
- Requête d'input utilisateur
- Fermeture du socket par le client

Dans le cas de la déconnexion et de fermeture du socket par le client, le thread exécute la fonction de gestion de "leave" d'un joueur. Cette fonction supprime le joueur de la carte du jeu, prévient les autres joueurs et si la partie ne dispose plus d'aucun joueur, celle-ci déclenche la variable de conditions de mise à jour (celle sur laquelle le thread principal attend) après avoir passé l'état de la partie à "fini".

Dans le cas de la réception d'une requête d'input utilisateur, le serveur process la requête grâce à une méthode de gestion d'input.
Cette méthode exécute toute la logique de déplacement et d'action du joueur :

- Déplacement dans la carte.
- Utilisation de bombes.
- Utilisation des portes.

Avant d'exécuter, les 3 différentes possibilités d'action, on vérifie que le joueur est en vie, n'est pas freeze et n'est pas en chute.

Si toutes les conditions sont respectées, alors, nous déclenchons l'utilisation de la porte, si une est à proximité lors de l'appui sur la touche "VALIDATE". Pour trouver la porte distante, nous lisons un tableau de couple de portes. La case à lire étant celle du numéro de

la porte, ainsi, nous pouvons retrouver la porte correspondant via ce tableau. Si une touche de direction est appuyée, nous procédons au calcul de la potentielle nouvelle position et l'envoyons à une fonction qui vérifie avec les éléments de la carte si ce mouvement est légal ou non. Cette fonction s'occupe de déclencher les actions suivantes :

- Récupérer une clef s'il y en a une de disponible sur la nouvelle position.
- Afficher le message pour prévenir de l'utilisation possible d'une porte.
- Récupérer un loot (Vie ou Bombe) et déclencher la routine de respawn du loot.

Une autre action possible du joueur est l'utilisation des bombes, ainsi, c'est comme pour l'utilisation des portes que celle-ci est effectuée. Nous vérifions si l'utilisateur presse la touche "ESPACE" et si celui-ci dispose d'au moins une bombe dans son inventaire.

Si oui, alors nous créer un objet bombe explosif sur la carte et déclenchons sa routine dans un thread grâce à une fonction spécifique à laquelle nous renseignons le pointeur de notre bombe explosive récemment posée. Le reste du fonctionnement de la bombe est explicité dans une autre partie.

Une dernière chose à voir au niveau des échanges entre le client et le serveur est l'envoi de messages d'informations qui s'affichent dans la zone d'information sous le niveau.

Cela est géré par une fonction à laquelle nous précisons le client ID, le message, la ligne et la couleur du message. Cette méthode existe en deux versions, celle pour un client spécifique et une version broadcast envoyant à tous les utilisateurs.

Il est intéressant de noter que nous passons à la majorité de nos méthodes une structure comportant la mémoire générale de la partie nous permettant ainsi d'accéder aux informations utiles sans avoir recours à une variable globale.

Bien entendu, la lecture et l'écriture sur cet argument de nos fonctions ne se fait pas sans le verrouillage préalable du mutex spécifique à l'information que nous voulons lire ou bien modifier.

Pour conclure cette partie, une description des attributs d'un joueur peut être intéressant :

- Le nombre de vies, nombre de bombes, les clefs possédées
- Les états : en Vie, Freeze, en Chute, est Invincible
- Le niveau où est le joueur, le numéro du joueur (égal clientID)
- Le pointeur vers l'objet sur la carte du joueur.

Autres threads de routines

Les différents threads que nous avons vus précédemment ne sont instanciés que lors du début de la partie et survivent tout le long de celle-ci (à l'exception de celui d'un joueur si celui-ci quitte la partie). À la différence, les threads explicités ci-dessous quant à eux peuvent être instanciés plusieurs fois durant la partie.

Les loots

Le thread des loots consiste en une routine à laquelle, nous passons le pointeur d'un objet venant d'être récupéré par un joueur. Et cette routine se charge de faire respawn le loot (Une vie ou bien une bombe) après le temps imparti de réapparition. Ainsi dans notre configuration, il s'agit donc d'un thread ne survivant que 5 secondes dont la majorité de son temps, celui-ci est en exécution passive en l'attente du temps imparti pour exécuter la réapparition du loot sur la carte de jeu.

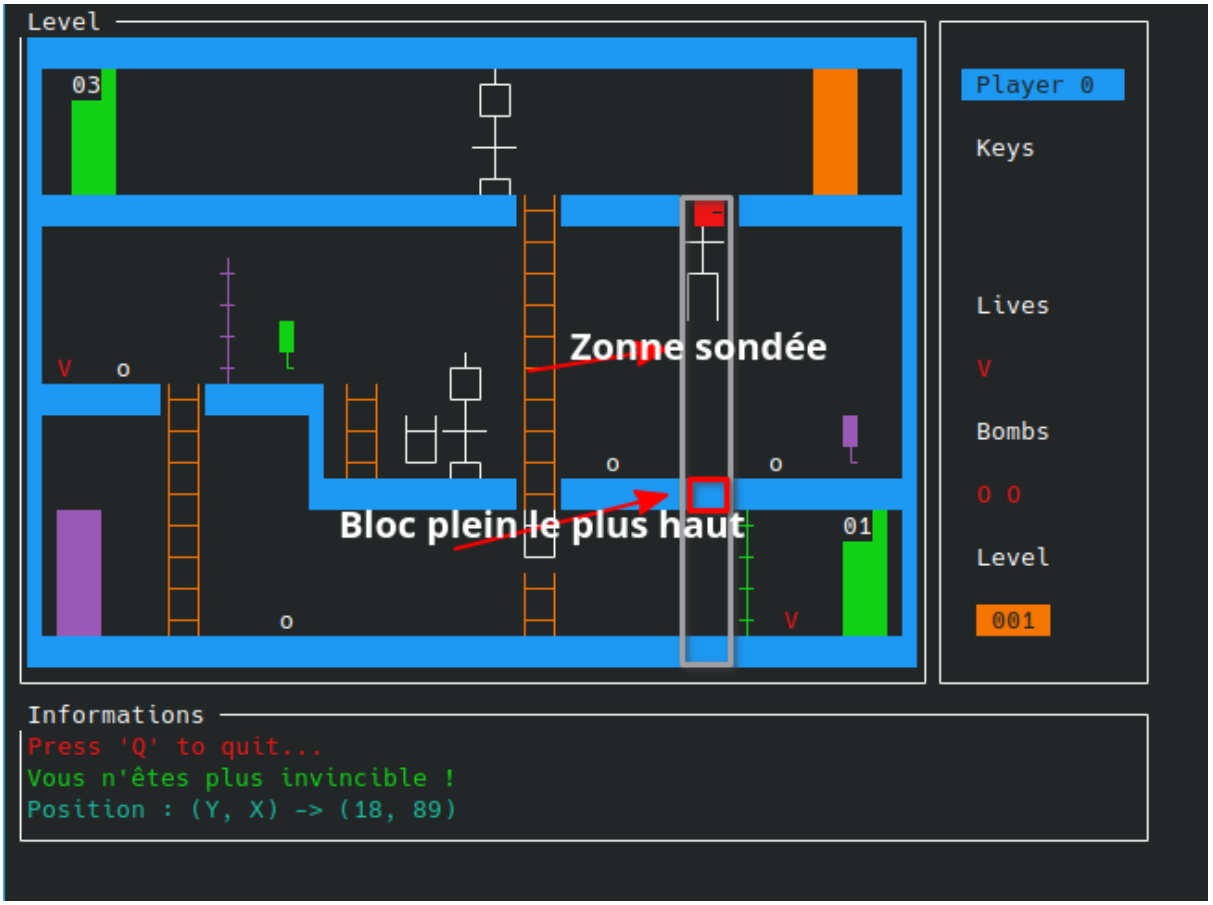
Les bombes

Les threads des bombes sont à ne pas confondre avec un thread de loot (type bombe) car ce thread est chargé de gérer la latence d'explosion d'une bombe déposée par un joueur et d'infliger les dégâts au joueur environnants ainsi que de les immobiliser, pareil pour les mobs aux alentours. "Les alentours" étant une zone sphérique de distance 5 autours de la bombe. Pour calculer les mobs et joueurs aux alentours, nous disposons de fonctions retournant les joueurs et/ou mobs dans une hitbox spécifier en arguments par sa taille et la coordonnée de son origine située localement en bas à gauche de l'hitbox. Ces fonctions retournent plus de mobs/joueurs étant formé d'une zone rectangulaire, ainsi, nous appliquons sur chacun des mobs/joueurs ressortis par ces fonctions un calcul de distance entre la hitbox des mobs/joueurs et de celle de la bombe. Une fois les mobs et joueurs impactés trouvés, nous pouvons leur appliquer la réduction de vie (seulement au joueur) et le freeze (Joueurs et mobs) ainsi que le déclenchement de la routine de mort dans le cas d'un joueur qui viendrait à perdre sa dernière vie.

La chute d’un utilisateur

Ce type de thread est déclenché par le thread d’input utilisateur lorsqu'un joueur se déplace à une position où il n’y a pas de blocs sous ses pieds, nous lançons donc ce thread qui se charge de freeze le joueur pour l’empêcher de bouger et le fait descendre d’un bloc toutes les 0.1 secondes jusqu’à atteindre l’objet solide le plus haut sous le joueur. Pour obtenir le nombre de niveaux de hauteur à faire descendre au joueur, nous utilisons une fonction ressortant tous les objets dans une hitbox défini, soit ici sous les pieds du joueur jusqu’au bas de la carte. Nous parcourons la liste des objets et retenons l’objet avec le y le plus bas (soit le plus haut, les y étant inversés) cela nous permet ainsi de trouver le point d’arrivée du joueur de façon optimale et d’obtenir par soustraction de la position du joueur de base le nombre de blocs à parcourir durant la chute pour retirer une vie si celui-ci tombe de plus de 2 de hauteur. Nous avons aussi ajouté par sécurité que si le joueur tombe hors de la carte due à un trou sur le plancher de la carte, nous effectuons un “GAME OVER” au joueur.

Schéma de détection de la chute d’un joueur :



Nos difficultés

Général

Nous avons choisi des listes chaînées pour stocker nos objets de carte de jeu.

Cela vient donc limiter l'optimisation de la parallélisation au niveau des niveaux, alors qu'en utilisant des matrices, nous aurions pu mettre des mutex par zone de la matrice d'un niveau permettant de réduire le déroulement séquentiel de certaines actions effectuées par les threads sur un même niveau.

Ainsi, cela serait une amélioration possible avec des heures supplémentaires sur le projet.

Éditeur

La découverte d'un "petit" bug de dernière minute durant le tournage de la vidéo de présentation.

Dans notre méthode de mise à jour d'un niveau, lorsque le niveau est plus grand, celui-ci est supprimé de sa zone actuelle dans le fichier et le trou est enregistré dans la table de vide.

Sauf que dans notre cas, la table vide était pleine. Ainsi notre système a alors créé une nouvelle table de vide en écrivant dans un des trous déjà enregistré, mettant donc à jour la taille d'un espace de la table de vide.

En finissant la fonction de création de la nouvelle table de vide, la fonction en cours d'exécution possédait déjà en mémoire la table de vide, mais sans les nouvelles modifications, en conséquence, le code de la fonction se basait sur une table de vide erronée causant des mutations de notre fichier non voulu.

Après 1 h 50 de lecture de notre fichier de logs, nous sommes enfin parvenus à comprendre la cause du problème et à le résoudre en ajoutant une mise à jour de la table en mémoire dans le cas où l'on crée une nouvelle table de vide ou table simple.

Client / Serveur

Durant le développement de la transition UDP vers TCP, au démarrage d'une partie, nous avons eu des soucis de débogage après quelques modifications effectuées sur le serveur.

Le problème était qu'au démarrage de la partie sur les logs du serveur, nous voyons bien le démarrage du socket TCP mais le client se coupait pour raison de "connection refused". Cela était dû à un "segmentation fault" sur le processus fils créé après l'ouverture du socket. Mais nous avons pris beaucoup de temps à comprendre, car le message de "segmentation fault" ne s'affichait pas. C'est en ajoutant des messages de logs via notre système de logs files que nous avons pu constater que le processus fils mourrait en silence sans alerte, provoquant donc logiquement un "connection refused" n'ayant aucun socket ouvert pour accepter la connexion TCP.