

Objectifs : Développer un jeu massivement multi-joueurs *War of Circles*

Notions abordées : Protocoles UDP / TCP - Programmation sockets en C

1 Préparation

L'objectif de ce TP est de développer un jeu multi-joueurs qui est une version (très simplifiée) du jeu Agar.io. Il s'agit de développer un serveur acceptant les connexions des joueurs et gérant le jeu, ainsi que le code client pour jouer.

Afin de réduire le temps de développement (qui fait appel à la librairie graphique SDL2), nous vous fournissons une version initiale du code avec l'interface utilisateur déjà implémentée, ainsi qu'un ensemble de fonctions utiles.

Commencez par télécharger ce code initial sur Moodle : [TP3-WoC-start.zip](#)

Une description du contenu de ce code est donnée à la fin du sujet.

2 Travail à réaliser

2.1 Attente de messages côté serveur

La première étape consiste à préparer le serveur à attendre des connexions des joueurs (clients) et réceptionner leurs messages.

Pour cela, dans la fonction `main()` du serveur, ajoutez le code permettant de créer un socket UDP (utilisez la variable globale `sock` déjà présente), le lier au port entré en paramètre de la commande `server` puis lancer un thread exécutant la boucle principale d'attente de messages (fonction `server_listen_loop()`). Le code permettant de lier le socket à un port est à écrire dans la fonction `bind_socket()` présente dans le fichier source `network.c`.

Dans la fonction `server_listen_loop()`, codez la boucle (infinie) de réception des messages des clients. Pour le moment, on affiche simplement le message reçu et notamment la valeur du premier octet reçu qui peut prendre une des trois valeurs : `REQUEST_ID`, `RELEASE_ID` ou `PLAYER_MOVE`.

2.2 Création socket client

Côté client, il s'agit maintenant d'établir une connexion au jeu en envoyant une demande au serveur. Pour cela, dans la fonction `main()` du client, écrire le code permettant d'obtenir les informations sur le serveur dont le nom (ou l'adresse IP) et le port ont été donnés en paramètre et créer le socket pour la communication des messages.

2.3 Demande d'identifiant

Côté client Dans la fonction `get_player_id()`, ajoutez le code envoyant la demande d'identifiant au serveur (message de type `REQUEST_ID`) et attendant la réponse. Lorsque la réponse du serveur arrive, afficher l'identifiant reçu.

Côté serveur Dans la fonction `server_listen_loop()`, ajoutez le code gérant la réception d'un message de type `REQUEST_ID`. Le serveur cherche l'identifiant disponible avec la fonction

```
| uint8_t next_available_id(GameData* g)
```

de `game.c`), stocke les informations liées à ce nouveau client dans la liste des clients connectés

```
| struct sockaddr_in clients_info[MAX_PLAYERS]
```

Le serveur initialise alors le nouveau client avec la fonction

```
| void init_player(Player* p)
```

puis envoie la réponse au client, formée d'une suite de deux octets : REQUEST_ID + identifiant

2.4 Échange des informations de jeu entre clients et serveur

Les données de jeu (liste des joueurs avec toutes leurs caractéristiques : position, vitesse, taille, etc.) sont stockées dans la structure `GameData` (cf. section ??). Le serveur centralise et calcule constamment les nouvelles positions puis envoie l'ensemble des informations aux clients pour qu'ils mettent à jour leur interface graphique. Pour cela, un échange de messages de type `GAME_DATA` est mis en place.

Côté client Dans la fonction `main()`, ajouter le code lançant un thread exécutant la fonction `client_listen_loop()`. Dans celle-ci, ajouter la boucle infinie d'attente de messages du serveur.





À réception d'un message de type `GAME_DATA`, le client copie les données reçues dans sa structure `GameData` locale et positionne la variable `redraw` à 1 afin d'indiquer que l'interface graphique doit être rafraîchie.

Côté serveur Dans la fonction `main`, le serveur lance un second thread dédié à l'envoi des messages `GAME_DATA` via la fonction `server_game_loop()`. Dans cette fonction, le serveur lance une boucle infinie qui calcule les nouvelles informations de jeu (*i.e.* les déplacements des joueurs, les collisions, etc.) puis envoie ces nouvelles données aux clients. Afin de régler un temps de rafraîchissement constant (x fps) le serveur doit envoyer les nouvelles données de jeu à tous les joueurs régulièrement toutes les y millisecondes. Pour cela, on demande l'heure système en début de boucle (commande `gettimeofday()` pour avoir une précision suffisante). Ensuite, les déplacements des joueurs et les résolutions d'éventuelles collisions sont calculés (fonction `move_players()`). Pour garantir un temps fixe de rafraîchissement, on redemande l'heure système après calcul et on calcule le temps à attendre (`usleep()`) avant d'émettre les nouvelles données (on suppose que le temps de calcul est inférieur au temps de rafraîchissement).

Exemple : on souhaite avoir un rafraîchissement à 20 fps. Il faut envoyer les données toutes les 50 millisecondes. Le serveur demande l'heure système : t_0 puis lance les calculs des déplacements et des collisions. Ce dernier dure t_{dc} millisecondes. Le processus doit donc attendre $t_{\text{timeToSleep}} = 50 - t_{dc}$ millisecondes avant d'émettre les données. Si $t_{\text{timeToSleep}}$ est négatif, on n'attend pas et on envoie les données immédiatement.

2.5 Envoi des déplacements et accélérations des clients au serveur

Quatre actions peuvent être effectuées par les joueurs :

-  virer à droite - GO_RIGHT
-  virer à gauche - GO_LEFT
-  accélération - GO_ACC
-  décélération - GO_DECC

Côté client Les clients envoient, lors d'une frappe d'un caractère de direction au clavier, un message de type `PLAYER_MOVE` au serveur pour indiquer le changement de direction ou d'accélération. Ce message se compose des trois octets :

`PLAYER_MOVE + identifiant + {GO_LEFT | GO_RIGHT | GO_ACC | GO_DECC}`

Côté serveur À la réception d'un message `PLAYER_MOVE` le serveur appelle simplement la fonction `update_player_direction` qui s'occupe de modifier les données de jeu en conséquence.

2.6 Gestion des déconnexions

Côté client Lorsqu'un client souhaite se déconnecter (fermeture de la fenêtre de jeu ou `ctrl + C`), l'événement `SDL_QUIT` est déclenché. Ajoutez le code afin de gérer proprement les déconnexions :

- arrêt du thread `client_listen_loop()`
- envoi d'un message de type `RELEASE_ID` au serveur (`RELEASE_ID + identifiant`)
- fermeture du socket

Côté serveur À la réception d'un message de type `RELEASE_ID`, appeler la fonction `release_player()`.

2.7 Pour les braves...

- ajouter l'utilisation d'un mutex pour éviter l'accès concurrent aux données `GameData`
- modifier le `gameplay`
- ajouter un chat afin que les joueurs puissent s'échanger des commentaires pendant la partie
- ...

3 Description du code

Structure représentation un point dans l'espace de jeu :

```
struct point2D
{
    float x;
    float y;
};
```

Structure représentant un joueur :

```
typedef
struct Player
{
    uint8_t valid;           // joueur valide ou non
    struct point2D position; // position sur le champ de bataille
    uint16_t size;           // taille du cercle
    uint16_t direction;      // direction de déplacement (en degrés)
    uint8_t speed;           // vitesse de déplacement (en pixels/unité de temps)
    uint8_t score;           // score
}
Player;

void init_player(Player* p);
void release_player(Player* p);
void update_player_direction(Player* p, uint8_t action);
void move_player(Player* p);
float distance(Player* p1, Player* p2);
```

Structure représentant les données du jeu, à savoir une liste de joueurs :

```
typedef
struct GameData
{
    Player players[MAX_PLAYERS]; // les joueurs de cette partie
}
GameData;

void init_gamedata(GameData* g);
uint8_t next_available_id(GameData* g);
void move_players(GameData* g);
```