Introduction

Le but de ce projet est de programmer un jeu de voitures multi-joueurs asynchrone. Le jeu est composé de plusieurs programmes :

- un programme maître qui gère la connexion des utilisateurs, la transmission du circuit et le calcul des positions des joueurs. Son rôle peut être assimilé à un serveur de jeu.
- un programme joueur (programme client), qui est lancé par chaque joueur et qui permet d'afficher le circuit et de demander un déplacement au maître.

1 Algorithmes

1.1 Déroulement normal maître-joueur

Dès la connexion d'un joueur, il s'initie un va-et-vient d'informations entre le programme maître et le programme joueur. (*Cf.* Tab. 1)

Programme maître	**Programme joueur**	
Processus maître	**Processus joueur**	**Processus affiche**
Envoi du numéro du joueur	Récéption	Attente taille du circuit
Envoi de la taille du circuit	Attente compte à rebours	Récéption
Envoi du circuit	Attente compte à rebours	Récéption
Envoi de la position de départ du joueur	Attente compte à rebours	Récéption
Envoi des coordonnées des autres joueurs	Attente compte à rebours	Récéption
Compte à rebours	Récéption et transmission au proc. affiche	Récéption et affichage
Départ	Récéption des touches saisies	Affichage du circuit
Réception + traitement du déplacement	Envoi d'un déplacement	Attente de nouvelles coordonnées
Envoi des nouvelles coordonnées	Récéption des touches saisies	Mise à jour de l'affichage

Tab. 1 – Description des échanges d'informations entre le maître et les joueurs

1.2 Détection des collisions

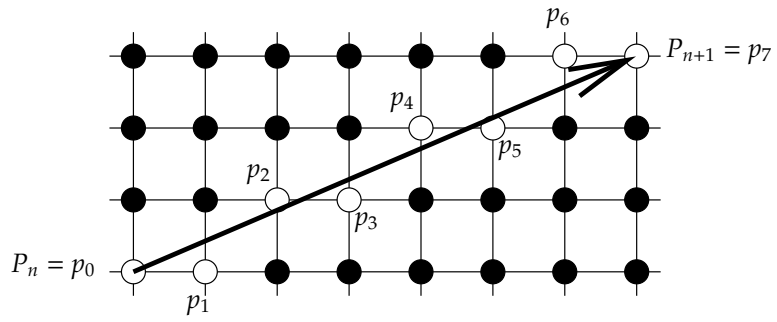

Fig. 1 – Points de $T_{(n,n+1)}$.

Si l'un des points blancs de la Fig. 1 est un mur, il faut forcer les coordonnées du joueur au point précédent et mettre sa vitesse à 0.

Le problème est d'obtenir ces fameux points proches pour vérifier une éventuelle collision. On utilisera de bien vieilles notions de mathématiques pour y parvenir. On calculera l'équation de la

droite passant par les points P_n et P_{n+1}, puis, on pourra calculer la distance des points à la droite, contenus dans le rectangle de diagonale $[P_nP_{n+1}]$.
Une droite s'exprime de la manière suivante :

$$y = ax + b,\ a \text{ étant le coefficient directeur et } b \text{ l'ordonnée à l'origine}$$

Le coefficient directeur est donné par la formule suivante :

$$a = \frac{y_{P_n} - y_{P_{n+1}}}{x_{P_n} - x_{P_{n+1}}}$$

On peut remarquer que lorsque $x_{P_n} = x_{P_{n+1}}$, on a $x_{P_n} - x_{P_{n+1}} = 0$. Cela correspond à une droite de type $x = y$. On traitera donc ce cas à part puisqu'une droite de ce type n'a pas de coefficient directeur (division par 0 dans le calcul de a).

Pour stocker une droite, on utilisera la structure suivante :

Structure droite
    coefDir : **Flottant**
    cte : **Flottant**
fin structure

La difficulté suivante est de parcourir tous les x et tous les y du rectangle dans *le bon sens*. En effet, suivant la direction du vecteur $\overrightarrow{P_nP_{n+1}}$, le parcours ne sera pas le même puisque le point précédent doit correspondre au point le plus proche du *point mur*, mais aussi du *point joueur* (*Cf.* Tab. 2). Pour cela, il faut parcourir les points proches dans l'ordre croissant par rapport à la trajectoire du joueur pour être sûr que le point précédent soit le bon.

$x_{P_n} < x_{P_{n+1}}$ et $y_{P_n} < y_{P_{n+1}}$	$x_{P_n} > x_{P_{n+1}}$ et $y_{P_n} > y_{P_{n+1}}$
$x_{P_n} < x_{P_{n+1}}$ et $y_{P_n} > y_{P_{n+1}}$	$x_{P_n} > x_{P_{n+1}}$ et $y_{P_n} = y_{P_{n+1}}$
$x_{P_n} > x_{P_{n+1}}$ et $y_{P_n} < y_{P_{n+1}}$	$x_{P_n} > x_{P_{n+1}}$ et $y_{P_n} = y_{P_{n+1}}$

Tab. 2 – Les six directions possibles d'une droite caractérisée par deux points

Pour effectuer le parcours dans le bon sens, il faut effectuer deux boucles **tant que** imbriquées. L'une s'occupera des ordonnées des points, l'autre des abscisses. On doit partir du point P_n et aller vers le point P_{n+1}.

Nom: parcours
Role: Effectue un parcours du point P_n au point P_{n+1}
Entrée: x_{P_n} : **Entier**, $x_{P_{n+1}}$: **Entier**, y_{P_n} : **Entier**, $y_{P_{n+1}}$: **Entier**
Variables: x, y, incx, incy : **Entiers**
début
    si($x_{P_n} < x_{P_{n+1}}$ **) alors**
        incx ← 1
    sinon
        incx ← -1
    finsi
    si($y_{P_n} < y_{P_{n+1}}$ **) alors**
        incy ← 1
    sinon
        incy ← -1
    finsi

    x ← x_{P_n}
    tant que(x≠ $x_{P_{n+1}}$+incx **) faire**
        y ← y_{P_n}
        tant que(y≠ $y_{P_{n+1}}$+incy **) faire**
            // *Traitement...*
            y ← y+incy
        fait
        x ← x+incx
    fait
fin

On peut ainsi très facilement stocker la valeur du point proche précédent et tester à chaque boucle si le point est de type route ou pas. Si le point proche en cours (donné par (x, y)) n'est pas de type route, la valeur des coordonnées du joueur sera donnée par le point proche précédent.

2 Protocole de communication

Pour que les différentes entités du jeu puissent communiquer entre elles, il faut définir un protocole. En effet, celui-ci doit prévoir la taille de chaque paquet envoyé, mais aussi son contenu de façon précise, pour qu'il n'y ait pas d'ambiguïté lors de la réception d'un paquet.

Il doit tout d'abord y avoir un certain synchronisme entre le programme maître et le programme joueur, notamment lors de la connexion d'un nouveau joueur (*Cf. §1.1 de la page 2*).

Voici la définition de ce protocole :

1. Chaque envoi/réception dans un tube doit être précis :
   - la réception d'un paquet ne doit se faire que si la taille de ce paquet est connue.
   - l'envoi d'un paquet doit respecter le protocole (taille et contenu du paquet).
2. Pour suivre les règles d'atomicité énoncé au §3.5 de la page 6, tous les paquets envoyés seront de taille fixe. Pour ne pas surcharger le tube de caractères inutils, on prendra la taille maximum qu'un paquet peut avoir. Ainsi, 15 caractères suffiront.
3. Les différents formats sont définis comme suit :

– `P_CONNEXION_JOUEUR`	`C|%d|%d`	Envoi des 2 *pids* au maître
– `P_NUM_JOUEUR`	`N|%d`	Envoi du numéro du joueur
– `P_TAILLE_CIRCUIT`	`M|%d|%d`	Envoi de la taille du circuit
– `P_FIN_CIRCUIT`	`M|end`	Fin de la transmission du circuit
– `P_POS_JOUEUR`	`P|%d|%d|%d`	Envoi de la position de départ du joueur
– `P_POS_AUTRES_JOUEURS`	`O|%d|%d|%d`	Envoi des positions de départ des autres joueurs
– `P_FIN_POS_AUTRES_JOUEURS`	`O|end`	Fin de transmission des positions de départ
– `P_COMPTE_REBOURS`	`T|%d`	Compte à rebours
– `P_FIN_COMPTE_REBOURS`	`T|end`	Fin du compte à rebours
– `P_COUP_JOUEUR`	`J|%d|%d`	Envoi d'un déplacement de joueur
– `P_NOUV_POS_JOUEUR`	`D|%d|%d|%d`	Envoi d'une nouvelle position de joueur
– `P_FIN_POS_JOUEUR`	`D|end`	Fin du jeu
– `P_TOUR`	`L|%d|%d|%d`	Information sur les tours
– `P_GAGNE`	`L|win`	Transmis au joueur qui a gagné
– `P_ARR`	`A|%d|%d`	Transmet la position d'arrivée au joueur

3 Analyse technique du sujet

3.1 Pourquoi utilise-t-on des tubes nommés ?

Dans le cadre de notre projet, l'utilisation de tubes anonymes pour communiquer entre le maître et les joueurs n'est pas possible car ceux-ci requièrent que les processus soient liés par filiation (un père et un fils créés par `fork()`). Les tubes nommés sont visibles dans l'arborescence comme de simples fichiers. Il faut, pour pouvoir lancer des joueurs par des utilisateurs différents du système, les créer dans un dossier partagé (`/tmp` par exemple) et leur donner les droits adéquats.

3.2 Pourquoi créer un tube par joueur alors que le maître n'en utilise qu'un seul ?

Il est possible que plus d'un joueur communique avec le maître en même temps. Tant que l'atomicité des écritures des joueurs est garantie (*Cf. §3.5 de la page 6*), ils peuvent tous utiliser le même tube nommé pour envoyer des données au maître.

Par contre, le maître ne peut pas utiliser un seul tube pour communiquer avec les joueurs. Si plus d'un joueur lit le même tube, il n'est pas possible de garantir que le bon joueur reçoive les données envoyées par le maître.

La solution consiste donc à créer pour chaque joueur son propre tube en lecture seule avant d'envoyer des données au maître (*Cf.* Fig. 2). Utiliser le *pid* du processus du joueur est une bonne façon de créer des tubes ayant un nom unique et facilement identifiable. Le client transmet donc par le tube maître son *pid* qui permettra au maître de savoir dans quel tube lui retourner des informations.

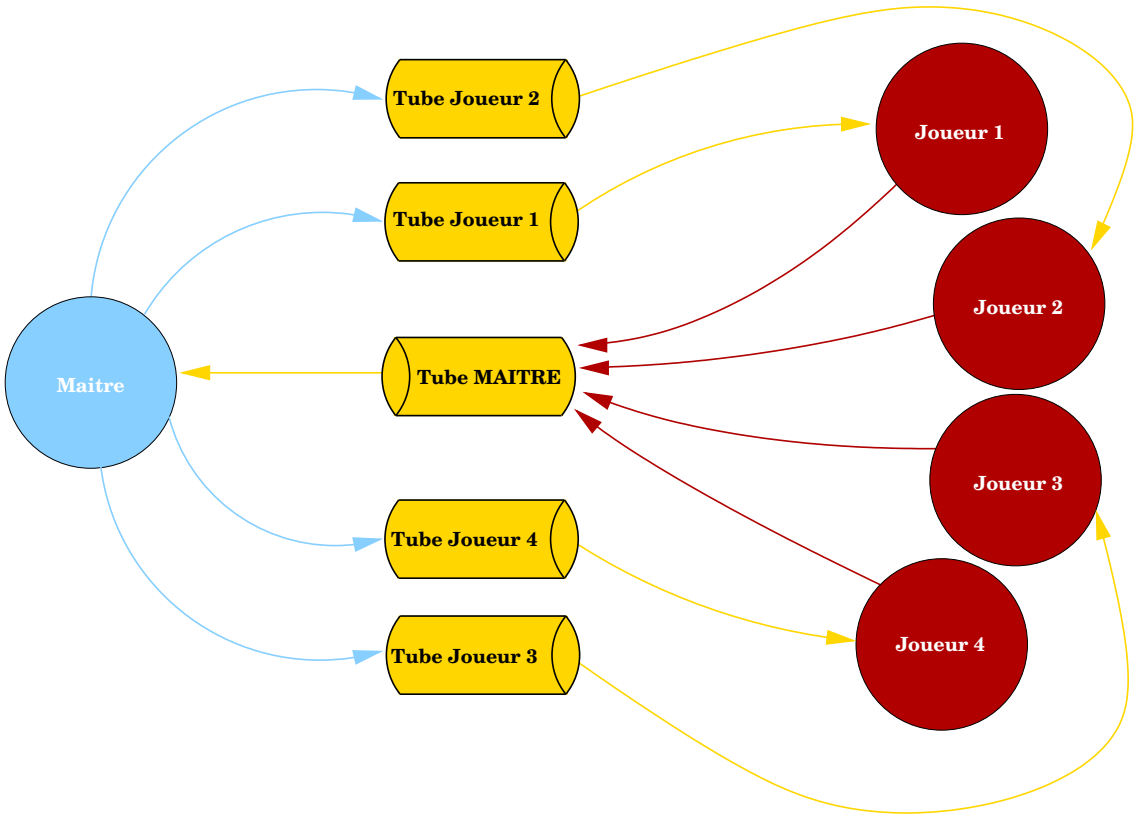

Fig. 2 – Schéma global des échanges entre maître et joueurs.

3.3 Synchronisation du processus joueur avec son processus d'affichage

Il est nécessaire de synchroniser le processus joueur avec son processus d'affichage. En effet, le processus joueur doit attendre que le processus d'affichage crée son tube et l'ouvre avant d'envoyer les *pids* des deux processus au maître. De même, le processus d'affichage doit attendre que le processus joueur lui envoie le signal de départ avant d'attendre les déplacements des joueurs.

Pour cela, nous avons créé deux tubes anonymes entre les deux processus, afin de permettre une communication bidirectionnelle. Ainsi, on pourra faire attendre un processus (bloqué en lecture sur le tube) pendant que l'autre effectuera les actions nécessitant une synchronisation. Dès que ces actions critiques seront effectuées, le processus pourra débloquer le lecteur du tube en lui envoyant une chaîne correspondant à la réussite ou à l'échec de telles ou telles actions.

3.4 Synchronisation du maître et des joueurs

La communication entre le maître et les joueurs se faisant à l'aide de tubes nommés, on en déduit le principe de synchronisation suivant :

- les tubes sont par défaut ouverts en mode bloquant.
- la lecture dans un tube est bloquante, c'est-à-dire que s'il y a au moins un producteur, le consommateur attendra les données et le `read` ne retournera une valeur qu'à ce moment.
- l'écriture est bloquante, c'est-à-dire que s'il y a au moins un consommateur, le `write` ne retournera une valeur que lorsque que toutes les données auront été écrites dans le tube.

Dans le cas de l'écriture, si le nombre de lecteurs est nul, le système envoie alors le signal `SIGPIPE` au processus tentant d'écrire dans le tube, le comportement par défaut de ce signal étant de sortir du programme. Il convient donc d'ignorer ce signal (`SIG_IGN`) et de se contenter de la valeur de retour du `write`, ou d'appliquer un traitement à ce signal, comme par exemple supprimer un joueur déconnecté (*Cf. §3.6 de la page 7*).

Dans le cas de la lecture, si le nombre d'écrivains est nul, l'appel à `read` retourne simplement la valeur 0.

3.5 Atomicité des écritures dans les tubes

3.5.1 Définition d'atomicité

Durant une procédure atomique, il est impossible d'interrompre son déroulement afin d'exécuter d'autres instructions que celles contenues dans ladite procédure.

Donc, dans le cas présent, si un client écrit dans le tube maître, aucun nouveau client (y compris le 1er) ne doit pouvoir écrire pendant la 1re opération.

Exemple :

deux clients écrivent dans le même tube :
le 1er écrit Bonjour, le 2e écrit Salut.

Durant une écriture atomique, si le 1er client écrit dans le tube, le 2e ne doit pas pouvoir écrire tant que le 1er n'a pas fini son écriture.

Il faut ainsi éviter la situation suivante : **Bon**Sa**jour**\0lut**\0**

Mais avoir Bonjour\0Salut\0

3.5.2 Comment la garantir ?

Pour garantir l'atomicité des écritures, il faut que la taille des données à écrire soit inférieure à PIPE_BUF octets (généralement défini dans `limits.h`), sachant que :

`PIPE_BUF` $\leq$ `_POSIX_PIPE_BUF` (= 512 octets)

Il faut donc concevoir le protocole pour que la taille des données à écrire en une seule fois fasse moins de 512 octets. Cela sera donc garanti pour tous les systèmes respectant la norme POSIX.

3.5.3 Est-ce suffisant ?

La règle citée précédemment est suffisante pour garantir l'atomicité de l'écriture dans un tube. Mais qu'en est-il de la lecture ?

L'appel à `read` retourne toujours le maximum de données possible, même si les appels à `write` correspondants sont issus de processus différents.

Comment, dans ce cas, garantir que le consommateur lira bien d'abord le 1er paquet et ensuite le 2e et non pas par exemple la moitié du 1er d'abord puis le reste ?

Pour que le consommateur lise bien le bon bloc de données ni plus ni moins, il faut qu'il en connaisse la taille. Pour cela, nous avons choisi de fixer la taille des paquets envoyés dans le tube. Le consommateur connaît donc à l'avance la taille des paquets à lire.

3.6 Détection et suppression des joueurs déconnectés

Pour cela il faut exploiter les valeurs de retour des fonctions qui permettent de communiquer avec les joueurs. Plusieurs cas de figure :

- le joueur se déconnecte avant de s'être enregistré auprès du maître (envoi des *pids*). Dans ce cas, le maître n'a rien à faire car il n'a même pas encore connaissance de ce joueur.
- le joueur se déconnecte durant sa procédure d'enregistrement (ouverture des 2 tubes, envoi du numéro de joueur et du circuit). On peut le détecter pendant l'ouverture des tubes (`open` retourne alors -1), ou pendant les opérations d'écriture dans les tubes du joueur (`write` retourne -1 mais il ne faut pas oublier d'ignorer le signal `SIGPIPE`). On supprime alors le joueur dont on ne tiendra plus compte pour la suite.
- le joueur se déconnecte en cours de jeu. Lorsqu'on détecte que le joueur s'est déconnecté (`write` retourne -1), il faut le supprimer, ne plus en tenir compte pour la suite et en informer les autres joueurs.

3.7 Détection d'un tour de circuit

Pour savoir si un joueur a effectué un tour de circuit, on détermine deux points de passage[1] (*Cf.* Fig. 3). L'un tout de suite après la ligne d'arrivée, l'autre avant (il est à noter que la ligne de départ/arrivée ne peut être qu'horizontale ou verticale). La méthode consiste a stocker dans un *tableau de deux cases*, l'état du point de passage (*Cf.* Tab. 3) :

- -1 signifie que le point de passage n'a pas été franchi.
- 1 signifie que le point de passage a été franchi en 1er.
- 2 signifie que le point de passage a été franchi en 2e.

Si un joueur passe plusieurs fois sur le même point de passage, seul le 1er passage est comptabilisé. De plus, si un joueur passe la ligne d'arrivée, les deux points de passage sont réinitialisés à -1. Cela signifie que si le joueur fait demi-tour après avoir franchi le 1er point de passage, passe le 2e et la ligne d'arrivée, le tableau ayant été réinitialisé par la ligne d'arrivée, le joueur n'aura pas son tour validé. Il reste cependant à déterminer un sens de parcours au circuit. On le fixera soit vers le *haut* ou le *bas* pour une ligne de départ horizontale, soit vers la *droite* ou la *gauche* pour une ligne de départ verticale. Ainsi, seul les passages dans le bon sens seront comptabilisés.

Valeurs du tableau	Description
$\{-1,-1\}$	Valeur de tableau au début du jeu ou après le passage de la ligne de départ/arrivée
$\{1,-1\}$	Passage dans le bon sens du premier point de passage
$\{1,2\}$	Passage dans le bon sens des deux points de passage

Tab. 3 – Valeurs que peut prendre le tableau des points de passage.

3.8 Affichage du circuit dans un terminal

Les fonctions standard de sortie du C (`printf`, ...) ne permettent pas de réaliser un affichage satisfaisant. En effet, plusieurs problèmes se posent :

[1] Un point de passage correspond à un ensemble de points, horizontal ou vertical

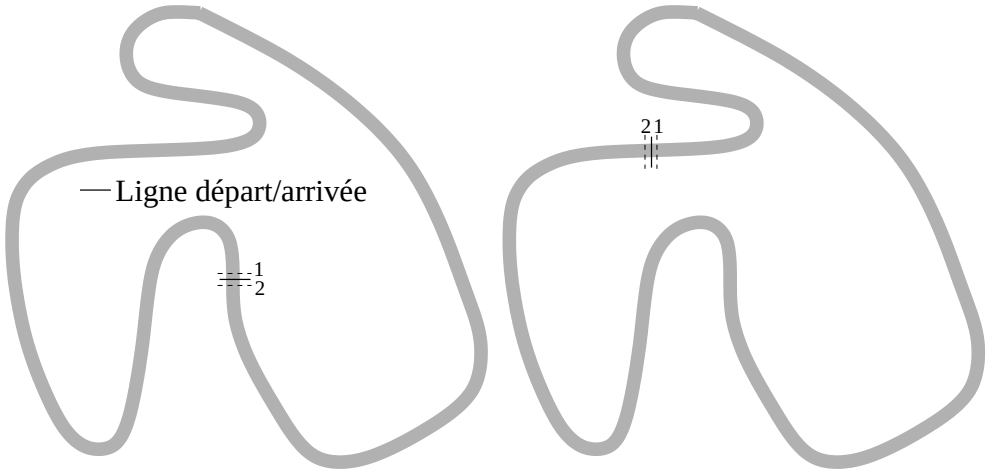

FIG. 3 – Exemple de points de passage d'un circuit.

- à chaque déplacement d'un joueur, il faut rafraîchir tout l'affichage, c'est-à-dire effacer l'affichage courant, calculer le nouvel affichage avec les nouvelles positions et l'afficher.
- il n'existe pas de fonction standard pour effacer l'affichage d'un terminal.
- le fait d'effacer tout l'écran à chaque déplacement provoque un scintillement peu agréable.

Il serait plus pratique de ne pas avoir à rafraîchir tout l'écran mais seulement les caractères qui ont changé sur le circuit. Pour cela, on peut envoyer des séquences d'échappement au terminal pour positionner le curseur à telle ou telle position. Mais ces séquences ne sont pas standard et diffèrent selon le type de terminal utilisé.

Pour résoudre les problèmes évoqués précédemment, nous avons utilisé la librairie `ncurses` qui permet de gérer de manière transparente les problèmes d'affichage sur les terminaux.

3.9 Positions précédentes des joueurs

L'affichage ne contiendra pas toutes les positions précédentes des joueurs mais seulement les 4 ou 5 dernières. Ainsi, on pourra aisément voir la trajectoire d'un joueur coup après coup.

Pour se faire, on mettra à jour à chaque déplacement, une file de positions pour chaque joueur. La taille de la file dépendra du nombre de *traces* que l'on désire. Ce nombre sera défini dans un fichier d'entête `.h`. Il sera alors très facile de le modifier pour en afficher plus ou moins.

Conclusion

Ce projet nous a permis d'approcher la notion de tubes et de processus UNIX de près, ce qui a constitué une bonne illustration du cours. Cela nous a également fait découvrir la librairie `ncurses`, qui peut s'avérer utile dans bon nombre de projets.

Enfin, l'écriture de ce rapport en LaTeXnous a donné l'opportunité de développer nos connaissances actuelles et d'en acquérir de nouvelles.

Pour améliorer le projet, on pourrait développer une interface graphique, ce qui serait plus convenable que le terminal pour l'affichage. Pour un jeu plus vivant, il serait intéressant de détecter les collisions entre les joueurs et pourquoi pas, d'ajouter des bonus, un peu comme dans *Mario Kart*, pour piéger ses adversaires.