

UNIVERSITÉ DE FRANCHE-COMTÉ

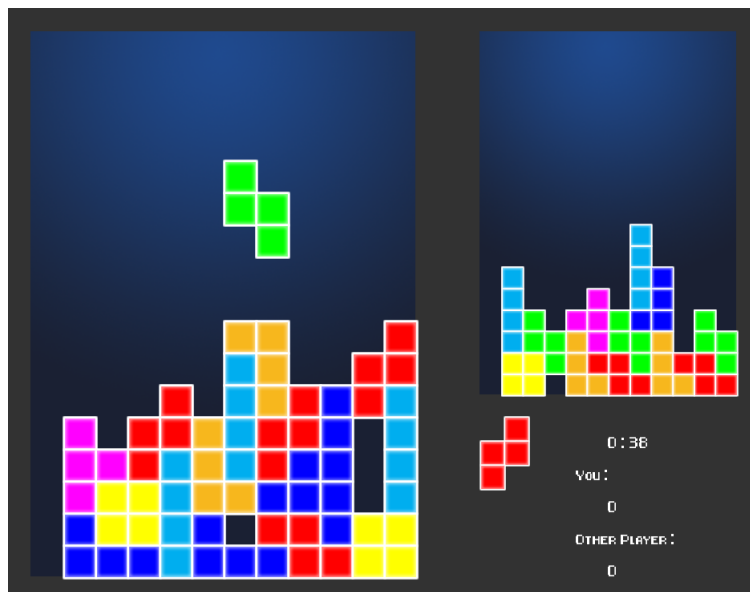
L3 CMI Informatique

PROJET INTÉGRATEUR

---

# Tétris en réseau - Le Vouitris

---



Cynthia MAILLARD, Alexandre DILLON, Félix ROYER

Tuteur de projet : Julien BERNARD

Année universitaire 2018-2019

# Table des matières

<b>1</b>	<b>Adaptation du jeu d'origine</b>	<b>4</b>
1.1	Tétris originel . . . . .	4
1.2	Notre adaptation . . . . .	5
1.2.1	Les objectifs du développement . . . . .	5
1.2.2	Les contraintes de développement . . . . .	5
1.2.3	Les malus . . . . .	5
<b>2</b>	<b>Modélisation du jeu</b>	<b>6</b>
2.1	Architecture réseau . . . . .	6
2.2	Échanges entre les clients et le serveur . . . . .	6
2.2.1	Les messages du serveur vers le client . . . . .	7
2.2.2	Les messages du client vers le serveur . . . . .	8
<b>3</b>	<b>La communication</b>	<b>9</b>
3.1	Le serveur . . . . .	9
3.1.1	La structure du serveur . . . . .	9
3.1.2	Le traitement des messages . . . . .	9
3.1.3	Le système anti-triche . . . . .	10
3.2	La sérialisation . . . . .	11
3.2.1	Sérialisation des types simples . . . . .	11
3.2.2	La sérialisation des types complexes . . . . .	12
3.2.3	La sérialisation des messages . . . . .	12
3.2.4	La désérialisation . . . . .	12
3.2.5	Une seule asymétrie entre Serializer et Deserializer . . .	12
<b>4</b>	<b>La jouabilité</b>	<b>13</b>
4.1	Les structures de données communes . . . . .	13
4.1.1	Tetromino . . . . .	13
4.1.2	Grid . . . . .	13
4.2	Le client graphique . . . . .	15
4.2.1	Les structures de données . . . . .	15
4.2.2	La fenêtre graphiques . . . . .	17
4.2.3	Les actions du joueur . . . . .	17

## Remerciements

Nous tenons à Monsieur Julien Bernard, notre encadrant pour ce projet, pour le soutien qu'il nous apporter tout au long du développement ainsi que l'aide pour la préparation de la soutenance et de ce rapport.

# Introduction

La réalisation de ce projet s'inscrit dans le cadre du projet intégrateur de notre troisième année de licence informatique à l'Université de Besançon. Pour ce projet, nous avons réalisé le sujet proposé par Julien Bernard, enseignant-chercheur en informatique à l'Université de Franche-Comté : un téttris multijoueur en réseau.

Le but est de développer notre version du célèbre jeu de puzzle, en faisant s'affronter deux joueurs l'un contre l'autre. Nous devons donc concevoir des interfaces graphiques pour les joueurs et créer un réseau pour permettre aux joueurs de jouer ensemble.

Le développement devait être fait en C++, avec la bibliothèque Boost.asio pour la communication réseau et les parties graphiques devait être réalisées en utilisant la bibliothèque Gamedev Framework, bibliothèque conçue pour le développement de jeux vidéo, développée par Julien Bernard. Le développement s'étendait d'octobre 2018 à mars 2019.

Nous étions intéressés par ce projet car nous voulions améliorer nos connaissances dans le langage C++. De plus la simplicité de la conception d'un Tétris nous permettait d'approfondir certains domaines spécifiques liés au développement, tel que approfondir la programmation distribuée et la sérialisation nous motivait pour ce projet.

Ce rapport a pour but de rendre compte du travail que nous avons réalisé au cours du projet.

# 1 Adaptation du jeu d'origine

## 1.1 Tétris originel

Tetris est un jeu vidéo de puzzle conçu par Alekseï Pajitnov en 1984. Tetris est principalement composé d'une zone de jeu où des pièces de formes différentes, appelées « tétrominos » — dont les différentes formes sont représentée sur la figure 2 —, descendent du haut de l'écran.

Durant la chute des tetrominos, le joueur peut déplacer les pièces latéralement, leur faire effectuer une rotation sur elles-mêmes et dans certaines versions, accélérer la vitesse de chute jusqu'à ce qu'elles se pose sur le bas de la zone de jeu ou sur une autre pièce.

Le but pour le joueur est de réaliser le plus de lignes possibles. Une fois une ligne complétée, elle disparaît, et les blocs placés au-dessus chutent d'un rang. Lorsque le joueur accumule les pièce et remplit la zone de jeu jusqu'en haut, ce qui empêche l'arrivée de tétriminis supplémentaires, la partie se termine. Le joueur obtient un score, qui dépend essentiellement du nombre de lignes réalisées lors de la partie. On ne peut donc jamais gagner à Tetris, le but étant d'améliorer son précédent score.

Après la version originale du jeu sortie sur l'*Elektronika 60*, le tétris a connu un succès mondial dans les années 1990 grâce à sa version Gameboy, dont on voit une capture d'écran sur la figure 1.

Le jeu a été adapté sur pratiquement toutes les consoles de toutes les générations, soit dans une version strictement identique soit dans une adaptation plus libre ne conservant que certains point du gameplay original. Au début des années 2010, on comptait plus de 65 plate-formes qui possédait un portage du jeu.

Tétris s'est imposé comme l'un des plus grands succès de l'histoire du jeu vidéo et l'une de ces icônes les plus mondialement connues. Il faut noter



FIGURE 1 – Interface de tétris sur Gameboy

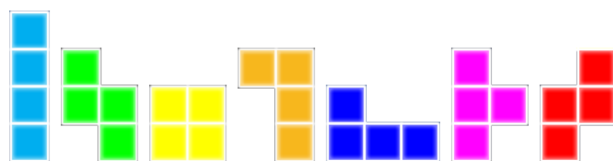


FIGURE 2 – Les différentes formes de tétramino

également que le jeu a connue des adaptations multijoueurs - notamment le projet *Tétrinet* à la fin des années 1990 - et ce encore aujourd'hui, avec par exemple *Tétris 99*, qui est sorti le 13 février 2019 sur Nintendo Switch.

## 1.2 Notre adaptation

### 1.2.1 Les objectifs du développement

Notre adaptation est une version multijoueur du Tétris, où chaque joueur a un jeu qui tourne sur son ordinateur, sur lequel il joue seul avec son clavier.

Le but est que chaque joueur puisse se connecter sur un serveur qui gèrerait le déroulement du jeu. Les joueurs n'aurait accès qu'aux minimum requises pour afficher le jeu et communiquer leurs action aux serveur.

Nous devons donc développer un serveur capable de gérer le jeu et de communiquer à des clients et des clients capables d'afficher l'état du jeu tel qu'il est sur le serveur et de lui renvoyer les interactions du joueur.

### 1.2.2 Les contraintes de développement

Cela nous a donné une contrainte à respecter : le serveur fait loi. Les clients ne servent qu'à l'affichage du jeu et à la réception des actions de l'utilisateur, mais ne contrôlent pas le déroulement de la partie.

De plus nous avons créer un système d'anti-triche afin de contrôler que les action du clients ne soit cohérente avec le déroulement de la partie.

### 1.2.3 Les malus

Les joueurs joueront une partie de Tétris classique, chacun de leur coté. Cependant, pour ne pas limiter l'affrontement à un concours de score, nous avons choisi d'ajouter un système de malus.

Lorsqu'un joueur parvient à détruire des lignes, son adversaires se voit pénalisé plus ou moins sévèrement en fonction du nombre de lignes détruites :

- suppression de 2 lignes : le joueur adverse ne peut plus faire de rotation sur ses pièces pendant 10 secondes.

- suppression de 3 lignes : la vitesse des chutes des pièce de l'adversaire augmente pendant 10 secondes.
- suppression de 4 lignes : certaines case sont enlevé du mur adverse pouvant l'empêcher de compléter certaine lignes.

Il n'y a pas de malus pour la suppression d'une seule ligne, car les parties était trop déséquilibrées avec.

## 2 Modélisation du jeu

### 2.1 Architecture réseau

Nous avons déjà défini que nous avons besoin d'une architecture client-serveur dans la partie précédente.

La difficulté étant que les clients et le serveur doivent être capable d'attendre l'arrivée d'un message sur leur socket, tout en exécutant une boucle simultanément - la fenêtre graphique pour les clients et la boucle de jeu pour le serveur.

Pour résoudre ce problème nous avons séparé la réception des messages dans un thread à part dans les clients et le serveur.

Ce thread attends la réception d'un message et le place dans une file. Le thread principal récupère ensuite les messages contenu dans cette file et les exploite, comme on peut le voir sur la figure 3.

Ainsi, les clients et le serveur vont pouvoir s'envoyer des messages sans problèmes, étant donné qu'il y a en permanence une socket qui écoute la réception de message. L'interprétation des messages est faite durant la boucle principale des deux applications, ce qui ne bloque pas leur exécution.

Le réseau a donc une architecture de réseau temps réel, mais les échanges de messages sont assez peu nombreux, donc notre réseau peut être considéré comme un réseau temps réel lent.

### 2.2 Échanges entre les clients et le serveur

Notre protocole d'échange tel que l'avons défini implique que le serveur gère seul la partie, et les clients uniquement l'affichage et les interactions de l'utilisateur.

Par conséquent, les échanges vont consister en la mise à jour de l'état du jeu de la part du serveur et les actions du joueur de la part du client. La figure 4 représente une version simplifiée des échanges entre les clients et le serveur. Pour des soucis de lisibilité, nous avons considéré que seul le premier joueur place des tétrominos.

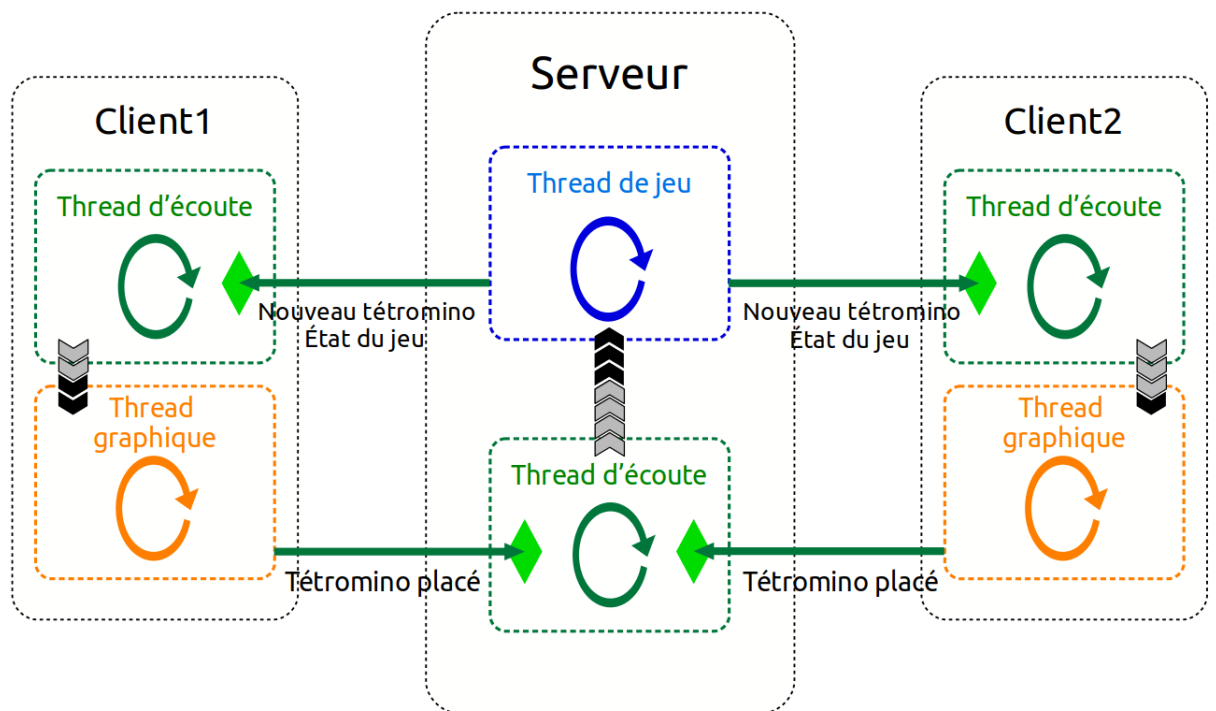


FIGURE 3 – Représentation simplifiée de l'architecture réseau entre les clients et le serveur

### 2.2.1 Les messages du serveur vers le client

- Game start : message de début de partie, signale le début de la partie aux clients lorsque les deux joueurs sont prêts à jouer
- Game over : message de fin de partie, il indique au joueur s'il a gagné, perdu ou s'il y a une égalité
- New tetromino : message contenant le prochain tétramino qui sera utilisé par le client
- Update Grid : message qui contient la grille de jeu du joueur
- Update Other Grid : message qui contient la grille de l'adversaire, pour l'afficher
- Malus start : signale le début d'une période malus pour le joueur ainsi que le type du malus qui s'applique
- Malus end : signale la fin de la période de malus actuelle

### 2.2.2 Les messages du client vers le serveur

- Tétramino placed : contient le tétramino avec sa position et son orientation tel qu'il vient d'être placé par le joueur



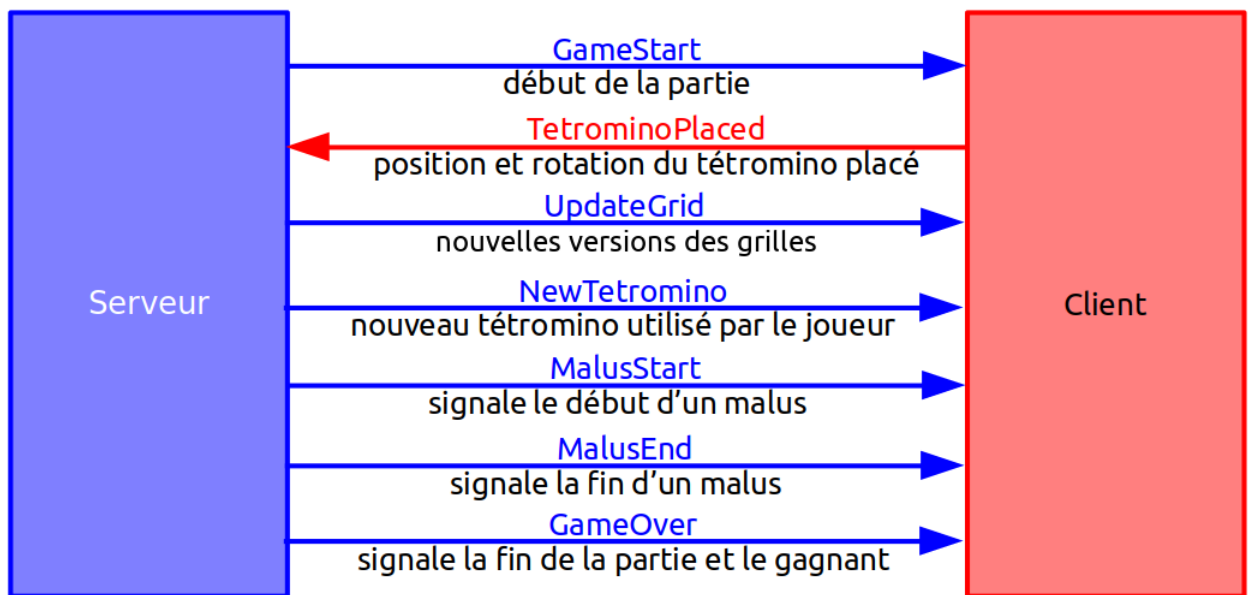


FIGURE 4 – Représentation simplifiée des échanges de messages entre les clients et le serveur

- Connection lost : message signalant une déconnexion du joueur
- Game\_Start : Contient les données des 2 premier tétramino de la partie et le temps total de la partie.
- New\_Tetromino : Contient les données d'un nouveau tétramino choisi au hasard par le serveur
- Update : Contient la grille de la zone de jeu et le score du joueur
- Update\_Other : Contient la grille de la zone de jeu et le score du joueur adverse
- Malus\_Start : Contient le type du malus et à quel joueur il est appliqué
- Malus\_End : Contient un identifiant pour savoir quel joueur est concerné par la fin d'un des malus
- Game\_Over : Contient les scores finaux des 2 joueurs
- Connection\_Lost :

## 3 La communication

### 3.1 Le serveur

Le serveur est la clé de voute du jeu. Il fait le lien entre les deux clients et gère l'évolutions du jeu.

### 3.1.1 La structure du serveur

Selon notre paradigme, le serveur fait loi, par conséquent, il contient toutes les informations nécessaires pour le déroulement de la partie. Ainsi, pour chaque client, il contient :

- une socket qui permet de communiquer avec le client
- une file contenant les messages de la part du client attendant d'être traité
- la grille de jeu du client, avec toutes les cases du jeu
- son score
- un vérificateur de triche
- un gestionnaire de malus, définissant quel malus est en cours pour le client et le temps restant pour le malus

Le serveur doit écouter les messages de la part des deux clients. Cependant, l'exécution du serveur ne peut être interrompue en attendant la réception d'un message. Nous avons donc créé deux threads d'écoute, chacun écoutant une socket associée à un client.

Lorsqu'un message est réceptionné, il est stocké dans la file du client en question. La file est partagée entre le thread d'écoute et le thread principal. À chaque tour de boucle, le thread principal regarde s'il y a un message stocké dans la file. Si c'est le cas, il traite ce dernier, met à jour les données du client et renvoie des messages aux clients si besoin est.

La figure 5 représente la structure du serveur avec ses différents threads, ainsi que les files et les sockets des clients qui sont partagées par ces derniers.

### 3.1.2 Le traitement des messages

Le serveur peut recevoir deux messages de la part du client : Connection lost et Tetromino placed.

Le message Connection lost sert à indiquer que le client se déconnecte du serveur, la partie est donc terminée et permet au serveur et au client de se fermer correctement. Dans ce cas, le serveur envoie un message GameOver à l'autre client - le désignant comme le vainqueur - et arrête son exécution.

Le message le plus fréquent sera Tetromino Placed. Sa réception indique que le joueur vient de placer le tétramino avec lequel il jouait. Ce message contient le type de tétramino, sa position et sa rotation.

D'abord le serveur va commencer par vérifier qu'il n'y a pas eu de triche - ce procédé sera décrit en détail dans la partie suivante.

La grille du joueur correspondant est ensuite mise à jour avec le tétramino reçu. Si une ligne est complétée, elle est supprimée, ce qui augmente le score du joueur en question.

Le malus est ensuite déterminé en fonction du nombre de ligne détruite :  
Si une ligne est détruite, aucun malus n'est envoyé.

Si le joueur détruit deux lignes, le serveur envoie à chacun des joueurs un message Send Malus Start, avec le type de malus "rotation".

- le joueur reçoit un message indiquant que l'adversaire est le cible du malus

- l'adversaire reçoit un message indiquant qu'il est la cible du dit malus

Le cas est similaire si trois lignes détruites, avec simplement le type du message, qui sera "accélération".

Dans ces deux cas, une horloge se lancera au moment de l'envoi du message. Au bout de 10 seconde, un message Send Malus End sera envoyé de manière similaire aux deux clients.

Si le joueur a effectué un téttris - quatres lignes détruites - un tétramino de type aléatoire est généré et placé aléatoirement en bas de grille de l'autre joueur. Les cases qui correspondent à ce tétramino seront supprimé de la grille l'empêchant de compléter des lignes.

Après la gestion des malus, les deux grilles mises à jour seront envoyées aux deux clients.

### 3.1.3 Le système anti-triche

Comme nous l'avons vu, le serveur fait loi, nous avons donc décidé d'ajouter un système anti-triche au serveur.

Le but est de vérifier que les données envoyées par les clients dans le message tétramino placed sont cohérente.

Pour ce faire, le serveur va vérifier le temps entre les deux messages du client, donc le temps que le client a pris pour jouer un tétramino.

A partir de la position du tétramino qui vient d'être placé, on calcule le temps maximum que celui-ci peut prendre pour arriver à cet endroit. Donc sa hauteur multiplié par le temps de chute normal. S'il dépasse le mouvement est considéré comme suspect. La vérification inclue aussi les potentiels malus en cours - délai plus court en cas de malus d'accélération et vérification sur la rotation pour le malus rotation.

Bien entendu si le réseau connaît des problèmes lors du transfert du message, cela peut retarder l'arrivée du message et donc fausser le calcul. Nous avons pris deux précautions contre cela.

D'abord nous avons ajouté une marge d'erreur de quelques secondes qui tiens compte de cette éventualité. Nous avons également prévu un système de fautes.

Un délai trop long entre deux messages - ou une rotation en cas de malus - cause une faute, au bout de trois fautes, le joueur est considéré comme un

tricheur et est éliminé. Son adversaire est alors déclaré vainqueur.

Ce système anti-triche peut être amélioré, notamment en vérifiant que la position du tétramino est bien atteignable depuis le haut du tableau.

Le serveur est donc la plaque tournante des communications du jeu, gérant et contrôlant les messages qu'il échange avec les clients.

## 3.2 La sérialisation

Les échanges des messages vus précédemment nécessite une sérialisation afin qu'ils puissent être envoyés par les sockets, ces dernières ne pouvant envoyer que des tableaux d'octet.

Pour ce projet, nous avons réalisé notre propre bibliothèque de sérialisation. Nous nous sommes inspirés des bibliothèques de sérialisation de GameDevFramework ainsi que de SFML/Packet. Notre sérialisation est organisée en deux classe symétrique : `Serializer` et `Deserializer`, qui sont communes au serveur et aux clients. Elles contiennent tous les deux un tableau dynamique d'octet qui représente les informations sérialisés ainsi qu'une position d'écriture ou de lecture, respectivement pour le sérialiseur et le désérialiseur.

### 3.2.1 Sérialisation des types simples

Pour la sérialisation des type simple nous utilisons une méthode templée privée qui peut sérialiser n'importe quel type simple. Cette sérialisation est ensuite appelé par d'autre méthode auxquelles sont assignés des types spécifiques afin d'éviter la sérialisation de type non-désiré.

L'endianess de cette méthode de sérialisation, c'est-à-dire l'ordre séquentiel dans lequel sont ranger nos données sérialisées, définit l'endianess de toute notre sérialisation. Celle-ci est au format Big-Endian pusique celui-ci est le format par défaut des infrastructures réseaux. Cela signifie que l'octet le plus significatif (octet de poid fort) est stocker en premier dans notre sérialisation et est donc envoyé en premier lors des échanges. Par exemple dans la figure 6, l'octet de valeur 0xAA est placé avant l'octet de valeur 0xBB dans le tableau d'octet lors de la sérialisation.

### 3.2.2 La sérialisation des types complexes

L'objectif étant de pouvoir sérialiser des messages, il nous faut pouvoir sérialiser des types complexes : objets ou structures. Pour cela nous sérialisons chaque attributs du type complexe que l'on souhaite échanger. Si l'attribut est un type simple alors on utilise la sérialisation des types simples vu précédemment sinon on fait appel la sérialisation du type complexe correspondant

à l'attribut, cela jusqu'à la sérialisation complète de notre type complexe.

### **3.2.3 La sérialisation des messages**

Nos messages sont répartis en deux structures, une pour les échanges client vers serveur et une pour les échanges serveur vers client. Chacune de ces structures contient une union contenant les structures des messages ainsi que le type du message représenté par une énumération : c'est une union taguée.

Les structures de messages contiennent ensuite ce que chaque message doit envoyer, des types simples ou des types complexes.

La sérialisation des messages est donc la suivante : on sérialise tout d'abord le type de message puis la structure de message présente dans l'union correspondante au type du message.

### **3.2.4 La désérialisation**

Comme précédemment évoquer, le Deserializer est le symétrique du Serializer, de ce fait la désérialisation va effectuer les opérations inverse de la sérialisation. Pour le cas des messages, le Deserializer va d'abord désérialiser la type du message pour savoir quelle méthode appeler pour désérialiser le reste du message.

### **3.2.5 Une seule asymétrie entre Serializer et Deserializer**

Une seule asymétrie entre Serializer et Deserializer existe, il s'agit de la gestion de la taille du message. Car pour envoyé notre message sur une socket, nous devons connaitre sa taille et la spécifier. Pour cela le sérialiseur garde toujours huit octets en début de son tableau dynamique pour que lorsqu'on récupère le tableau d'octet, la taille du tableau soit insérée au debut de celui-ci. Ainsi cela permet lors de la réception du message, de lire les huit premiers octets pour connaitre la taille du message et alloué un tableau dynamique de la bonne taille pour enfin l'assigner à un désérialiseur.

## **4 La jouabilité**

### **4.1 Les structures de données communes**

#### **4.1.1 Tetromino**

Le tirage au sort de tétramino se fait du côté serveur. Le serveur envoie ensuite les données concernant les tétrominos aux clients.

Au début de la partie, le client récupère les données du premier tétramino en jeu, `currentTetro` et les données du tétramino suivant, appelé `nextTetro` grâce au message `GameStart`. Pour la suite de la partie, lorsque le tétramino en jeu se pose, le client le signale au serveur, avec le message `TetrominoPlaced`. Le serveur renvoie le message `NewTetromino` au client contenant les données d'un nouveau tétramino. Ce nouveau tétramino prend la place du tétramino de prédiction (`nextTetro`) alors que l'ancien `nextTetro` devient le `currentTetro` et entre en jeu.

La classe `Tetromino` contient les informations sur un tetromino :

- son type
- son sens de rotation actuel
- sa forme représentée par une matrice de 4x2
- la position de son ancre représentée par un vecteur (x, y)

Cette fonction permet de récupérer la liste des coordonnées des cases du tétramino.

1. On récupère les coordonnées de l'ancre ainsi que sa place dans la matrice 2x4 représentant la forme du tétramino
2. On récupère l'entier représentant la rotation du tétramino
3. Suivant la rotation du tétramino, le sens du parcours de la matrice 2x4 est différent
4. Si on est sur une case contenant une partie du tétramino soit une case avec un entier différent de 0, on ajoute les coordonnées de cette case, sous forme de vecteur, dans la liste à retourner
5. Une fois le parcours de la matrice terminé on renvoie la liste de coordonnées

#### 4.1.2 Grid

Pour représenter la zone de jeu nous utilisons un tableau d'entier à une dimension. Une case vide est représentée par un 0. On crée le tableau avec l'objet `Grid`.

**Vérification des déplacements** La fonction `Grid : :movePossible` permet de vérifier si l'action de déplacement faite par le joueur est valide. Elle prend en paramètre le `Tetromino` en jeu (`tetromino` qui chute et qui est contrôlé par le joueur) et un vecteur (x, y) pour savoir dans quelle direction le mouvement est fait.

Cette fonction est appelée dans :

- `Grid : :downPossible`, avec le vecteur 0, 1
- `Grid : :rightPossible`, avec le vecteur 1, 0

— Grid : :leftPossible, avec le vecteur -1, 0

La fonction Grid : :rotatePossible permet d'autoriser ou non la rotation de la pièce en jeu. La rotation peut être par exemple refusée si le tétramino est bloqué entre 2 autres tétraminoes ou si il est sur le bord de la zone de jeu. Dans ces 2 cas, la rotation du tétramino le ferait soit passer à travers d'autre pièce ce qui est impossible ou encore sortir de la zone de jeu. On vérifie donc si les cases du tétramino après rotation sont bien dans la zone de jeu et qu'il n'y ai pas déjà un tétramino sur cette case grâce à la fonction Tetromino : :getCases.

**Placement d'un tetromino** Si la fonction Grid : :downPossible renvoie false, le tetromino actuellement en jeu est placé. Hors, seul son ancre est visible dans le tableau lors de sa chute, il faut donc ajouter toute les cases qui compose le tetromino dans la zone de jeu afin d'envoyer un nouveau tetromino en jeu. On appelle les fonctions Tetromino : :getCases et Tetromino : :getType pour placer toutes les cases du tétramino dans le tableau.

**Suppression de lignes** La fonction Grid : :deleteLines parcourt le tableau et détecte si des lignes sont pleines, elle est appelée lorsqu'un tétramino est posé. Si c'est le cas, elle appelle la fonction Grid : :fallLines sur la ligne correspondante. Elle renvoie le nombre total de ligne pleine qui ont été supprimé pour effectuer le calcul du score et l'envoi de malus à l'adversaire.

La fonction Grid : :fallLines récupère en paramètre la ligne pleine, et fait descendre toutes les lignes au dessus de celle-ci d'une case vers le bas en commençant par le bas du tableau.

**Vérification de l'état de la grille** La fonction Grid : :gameOver est appelé à chaque tétramino posé. Elle vérifie si les lignes du haut du tableau sont vides. Si une pièce est présente dans cette zone, on estime que la zone de jeu est complètement remplie et que l'arrivée d'un prochain tétramino est impossible, dans ce cas elle renvoie true. Et le tableau est vidé entièrement.

## 4.2 Le client graphique

Le client est la partie visible de l'application pour les joueurs. Son rôle est de recevoir les messages envoyés par le serveur, de capter les interactions du joueur sur la partie et de les transmettre au serveur et d'afficher les différents éléments composant la fenêtre de jeu.

### 4.2.1 Les structures de données

**Le tableau de la zone de jeu** Comme expliqué précédemment, notre zone de jeu est représenté par un tableau crée avec l'objet Grid de largeur 12 et de hauteur 21. Dans la figure ?? nous avons représenté de façon schématique les données stockés dans ce tableau à un instant T d'une partie.

Toute les case sans chiffres sont des cases contenant un 0. Nous voyons que le numéro stocké correspond au type du tetromino.

Encadré en rouge nous avons l'ancre du tetromino actuellement en chute. Ce n'est qu 'au moment de l'affichage que toute les case du tetromino sont récupéré. Pour la gestion de la chute, il nous etait plus simple de n'avoir qu'une seul case à déplacer.

Tout en haut du tableau de la zone de jeu, encadré en bleu, nous retrouvons 4 lignes qui n'apparaissent pas dans l'affichage pour le joueur.

C'est 4 lignes nous permettent de faire apparaître le nouveau tetromino depuis le dessus de la zone de jeu et ce sont sur ces lignes que s'effectue le test Grid : :gameOver pour savoir si la zone de jeu est entièrement remplie.

**GameArea** La classe GameArea est uniquement utilisé par le client. Elle contient tout les chargements des textures pour les sprites des différentes pièces.

Nous avons un tableau de Sprite de largeur 12 et de hauteur 17 (on ne retrouve pas ici les 4 lignes du haut du tableau de la zone de jeu puisqu'elles ne sont pas affichée.

On récupère les données stocké dans le tableau de la zone de jeu (Grid) à partir de la 5ème lignes et on modifie les sprite contenue dans le tableau de gameArea.

La fonction GameArea : :setScale utilise la classe gf : :Transformable afin de modifié la taille de l'affichage en fonction de vecteur et ainsi nous permettre d'utilisé les même sprite pour la zone de jeu du joueur et celle du joueur adverse.

**Initialisation de la partie** Au tout début de la partie nous initialisons certains paramètre.

Les score des 2 joueurs sont à 0 Toute les cases des grilles de zone de jeu des 2 joueurs sont initialisés à 0 Le booléen « enPartie » de notre boucle de jeu est passé a true Le client reçoit le premier tetromino en jeu et le prochain tetromino Le timer est initialisé avec la durée de la partie

**Boucle de jeu** Notre boucle de jeu se repose sur le booléen « enPartie »

Il passe à false si : Un des joueur quitte la partie Le timer tombe à zero



Pendant la boucle de jeu le client :

- Reçois les messages du server
- Récupère les actions du joueur
- Vérifie que les action faites par le joueur sont réalisable et mets a jour la position et la rotation du tetromino en jeu
- Si le tetromino est posé alors le tétramino de prédiction devient de le tétramino en jeu et il est placé en au de la grille de jeu, et le client envoie un message au serveur pour demander le prochain tetromino

Une fois toute les donnée mises a jour, l’affichage est réalisé et la boucle suivant commence

Si le message concernant les malus est reçu alors certaine des données sont modifié jusqu’à ce que le temps de ce malus soit terminée.

Si le booléen « enPartie » passe a false alors on sort de la boucle de jeu et on entre dans une autre boucle qui permet l’affichage final pour indiqué si le joueur à gagné ou perdu.

A la réception des message du serveur :

Game\_Start : Récupère et stocke les données du tétramino en jeu et du prochain tétramino.

New\_Tetromino : Récupère et stocke les données du prochain tétramino

Update : Mets à joueur l’affichage de la zone de jeu et du score du joueur

Update\_Other : Mets à joueur l’affichage de la zone de jeu et du score du joueur adverse

Malus\_Start : Vérifie sur quel joueur le malus s’applique. Si c’est le joueur qui reçoit un malus, on récupère le type du malus. On affiche le logo correspondant et la grille du joueur devient rouge. Si c’est le joueur adverse qui reçoit un malus, on affiche sa grille en rouge.

Malus\_End : Vérifie sur quel joueur le malus correspondant s’appliquait. Si c’est le joueur qui est concerné alors on enlève le logo du malus de l’affichage et sa zone de jeu revient à la couleur initiale. Si c’est le joueur adverse qui est concerné, on affiche sa zone de jeu de la couleur initiale.

Game\_Over : On récupère les résultats de la partie et on passe le booléen de la boucle de jeu à false.

Connection\_Lost : On passe le booléen de la boucle de jeu à false

#### 4.2.2 La fenêtre graphiques

Les différents element composant notre ecran de jeu :

En rouge : Zone du jeu du joueur. On y trouve les tetromino déjà posé ainsi que le tetromino en jeu en train de chuté. C’est avec cette fenetre que le joueur va interagir.

En jaune : Zone du joueur de l'adversaire. On ne voit pas la piece en jeu mais on voit les piece déjà posé. Le joueur peut alors voir si son adversaire est en difficulté ou si au contraire une ligne est sur le point d'être formée.

Ces 2 zones sont crée grace au classe Grid, et Game Area ainsi qu'en utilisant la classe Transformable de la bibliothèque gf nous permettant facilement de rendre la vision de la zone de jeu adverse plus petit que celle du joueur.

En vert : Prochaine pièce

En blanc : le timer

En violet : Les scores des 2 joueurs

En noir : Affichage du malus en cours. Si aucun malus n'est actif cette partie de l'écran est vide. Une jauge de couleur rouge permet de savoir approximativement le temps qu'il reste pour que le malus se termine

#### **4.2.3 Les actions du joueur**

Lors de la chute du tetromino en jeu, le joueur peu interagir sur celui ci avec les touche du clavier :

Les flèches gauche et droite et les touche Q et D permettent de déplacer latéralement le tétramino. La flèche du bas et la touche S permettent d'accélérer la chute si on reste appuyé dessus. La barre d'espace permet de faire tourné le tetromino sur lui même.

De plus, la fermeture de la fenêtre étant aussi une action du joueur sur le jeu, elle fait partie de la liste des contrôle.

Pour capter les actions réalisés par le joueur nous avons crée une classe Controls.

Dans cette classe nous définissons des objets de classe Action de GF.

Pour les actions lié à une touche de clavier nous utilisons la fonction `gf : :Action : :addScancodeKeyControl` avec le `gf : :Scancode` correspondant à la touche voulue.

Nous avons aussi une fonction `reset` qui permet de stoppé toute les actions en cours. Nous l'utilisons dans la boucle de jeu.

## **Conclusion**

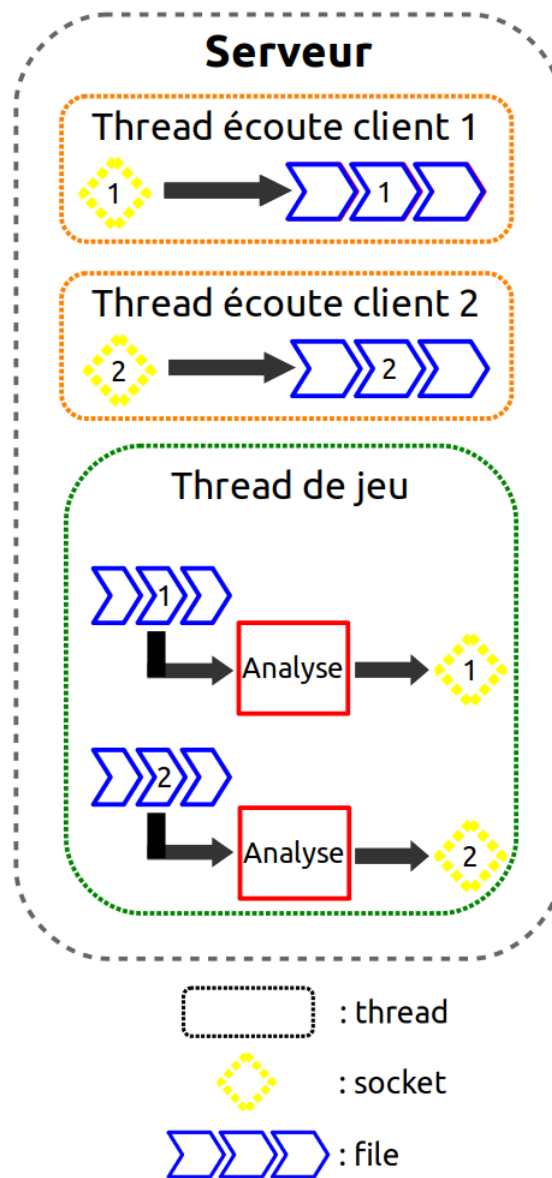


FIGURE 5 – Représentation simplifiée de la structure du serveur

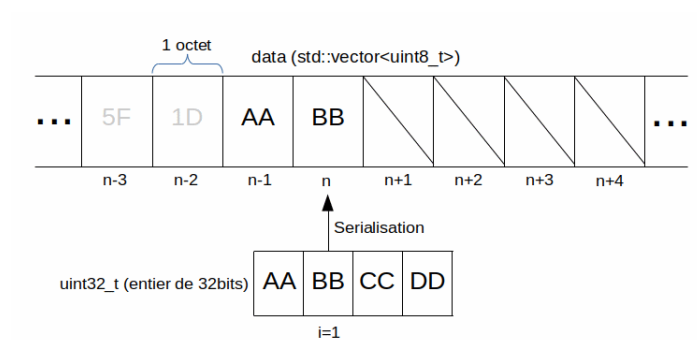


FIGURE 6 – Représentation simplifiée de la sérialisation d’un entier de 32bit, est ici représenté la sérialisation du second octet avec la position d’écriture égale à  $n$ .

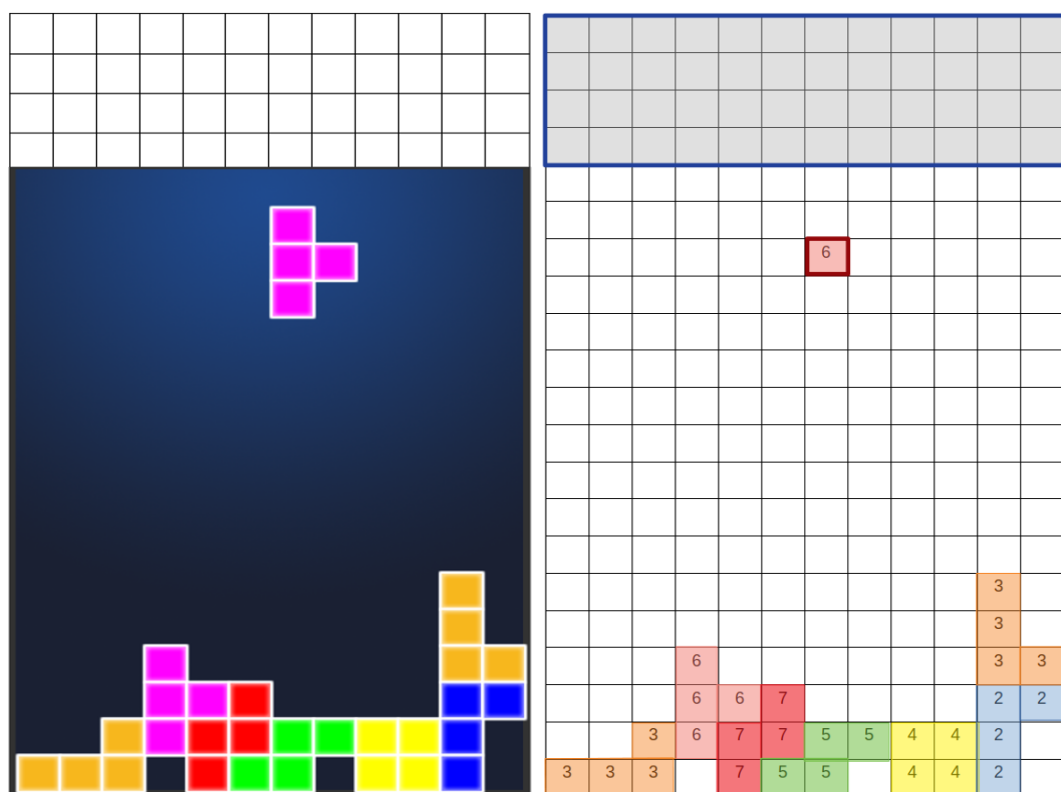


FIGURE 7 – Représentation numérique des données dans le tableau de jeu

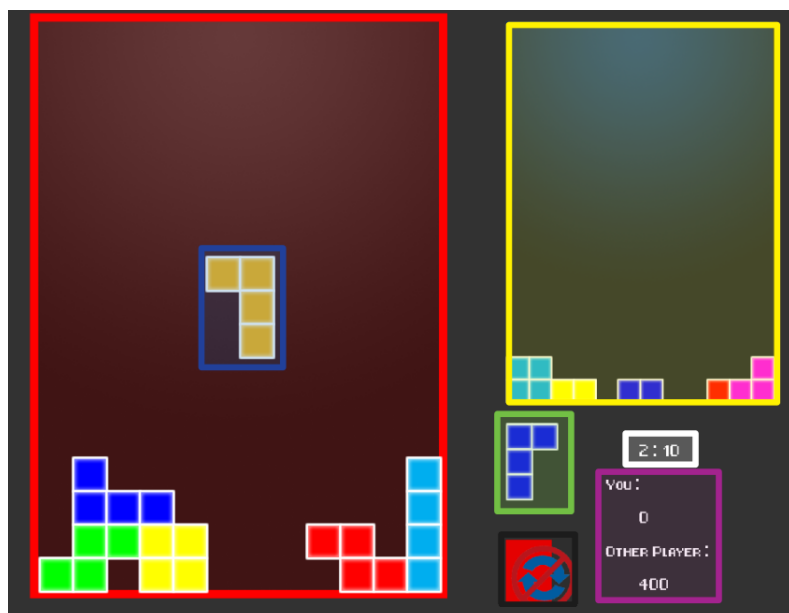


FIGURE 8 – Différentes zone d’affichage de la fenêtre du jeu