

# Le vouitris

Cynthia MAILLARD      Alexandre DILLON  
Félix ROYER

?? mars 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 . . . . .	4
<b>2</b>	<b>Modélisation du jeu</b>	<b>5</b>
2.1	Architecture réseau . . . . .	5
2.2	Échanges entre les clients et le serveur . . . . .	5
2.2.1	Les messages du serveur vers le client . . . . .	6
2.2.2	Les messages du client vers le serveur . . . . .	6
<b>3</b>	<b>Mise en oeuvre</b>	<b>7</b>
3.1	Le serveur . . . . .	7
3.2	La sérialisation . . . . .	7
3.2.1	Sérialisation des types simples . . . . .	7
3.2.2	La sérialisation des messages . . . . .	8
3.2.3	La désérialisation . . . . .	8
3.2.4	Une seule asymétrie entre Serializer et Deserializer . . .	9
3.3	Le client graphique . . . . .	9
3.3.1	Tetromino . . . . .	9
3.3.2	Grid . . . . .	9
3.3.3	Les malus . . . . .	10

## Remerciements

## 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 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 que le court temps de développement ne nous aurait pas permis d'aborder sur un jeu plus complexe. Notamment, 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 « tétriminos », descendent du haut de l'écran. Durant la chute des tetriminos, le joueur peut déplacer les pièces latéralement, leur faire effectuer une rotation sur elles-mêmes et accélérer la vitesse de la descente dans certaines versions 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 tétriminos 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.

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 possadaient 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 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

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.

L'objectif hypothétique étant que pour jouer ensemble, deux joueurs puissent télécharger le client du jeu sur leurs ordinateur puis se connectent à un serveur en ligne qui gèrerait le déroulement de la partie. Ainsi, on pourrait imaginer que le joueur rédeveloppe son propre client avec son propre interface graphique, et il pourrait jouer sans problème, à condition de reproduire les échanges entre le client et le serveur prévus dans le protocole.

Cette proposition présente le risque qu'un joueur recode sa version du

client et soit donc capable de tricher, car n'étant plus contraint aux mêmes règles que son adversaire.

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.

Les joueurs joueront une partie de Tétris classique, chacun de leur côté. 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 adversaire se voit pénalisé plus ou moins sévèrement en fonction du nombre de lignes détruites.

## 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.

Un problème que nous avons rencontré est 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 attend 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 1.

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.

### 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 2 représente une version simplifiée des échanges entre les clients et le serveur. Pour des soucis de lisibilité, nous avons assumé que seul le premier joueur place des tétrominos.

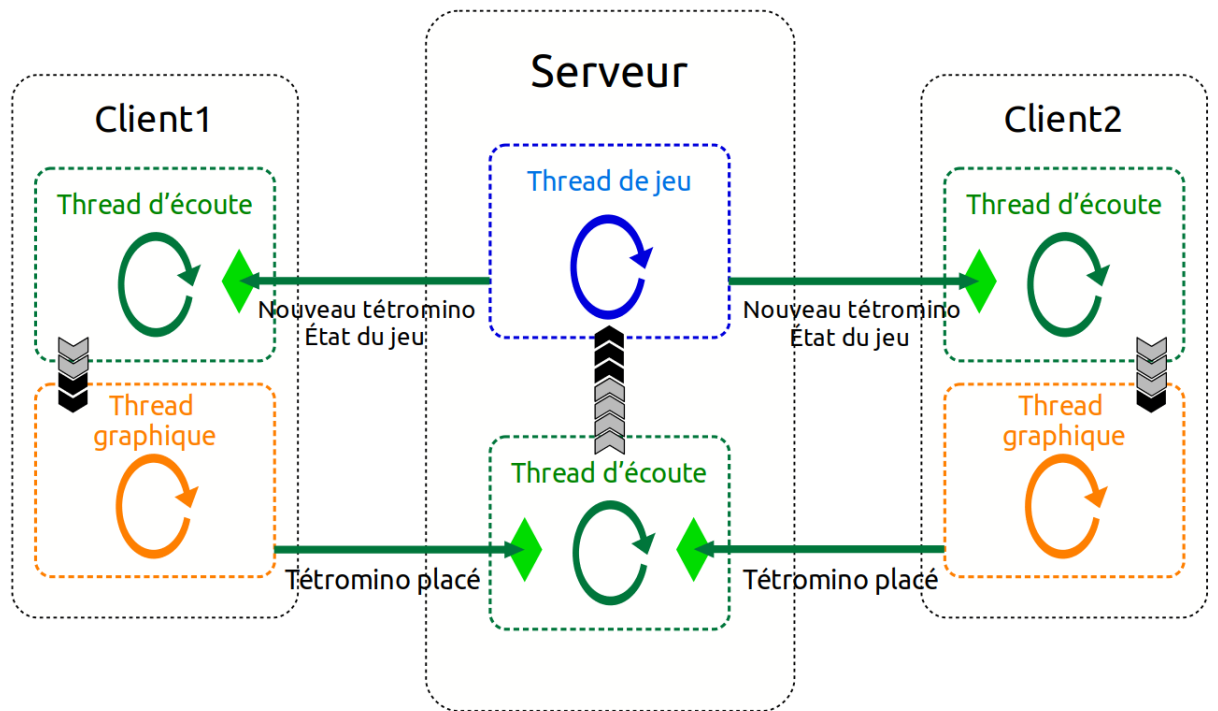


FIGURE 1 – Représentation simplifiée de l'architecture réseau entre les clients et le serveur

### 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 viens d'être placé par le joueur

- Connection lost : message signalant une déconnexion du joueur

## 3 Mise en oeuvre

### 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

**Les threads** Le serveur doit écouter les messages de la part des deux clients. Cependant, l'exécution du serveur ne peut être interromput en attendant la reception d'un message. Nous avons donc créé deux threads d'écoute, chacun attendant les message d'un des client. Lors qu'un message est réceptionné, il est stocké dans une file. La file est partagée entre le thread d'écoute et le thread principal. Á chaque tour de boucle, le thread principal regarde s'il y a un message stocké dans le file. Si c'est le cas, il interprète ce dernier, met à jour les données du client et renvoie des message aux deux clients si besoin est.

#### L'interprétation des messages

##### 3.1.2 Les malus

Présentation des différents malus :

Lorsque le joueur complète des lignes, celle ci sont supprimés. On calcul le nombre de lignes supprimé afin d'augmenté le score du joueur mais aussi d'envoyé des malus au joueur adverse. Pour les malus nous avons choisis :

- suppression de 2 lignes : le joueur adverse ne peut plus faire de rotation sur ses pièces pendant N secondes.
- suppression de 3 lignes : la vitesse des chutes des pièce de l'adversaire augmente.
- suppression de 4 lignes : certaines case sont enlevé du mur adverse pouvant l'empêcher de compléter certaine lignes.

Transmission des messages pour les malus ?

### 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

envoyés que des tableaux d'octet.

Pour ce projet, nous avons réalisé nos propres classes de sérialisation. Nous nous sommes inspirés des bibliothèques de sérialisation de Gamedev-Framework ainsi que de SFML/Packet. Notre sérialisation est organisée en deux classe symétrique : Serializer et Deserializer. 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 serialiseur et le deserialiseur.

### 3.2.1 Sérialisation des types simples

Pour la sérailisation des type simple nous utilisons une méthode templée privée qui peut sérialiser n'importe quel type simple. Cet 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 cet méthode de sérialisation, c'est-à-dire l'ordre sequentiel 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 le plus commun au infrastructure réseau. 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.

Listing 1 – Méthode de sérailisation de type simple data est notre tableau dynamique d la variable de type T à sérailiser et writePos la position d'écriture du sérialiseur

```
template <typename T>
void Serializer::serializeAnyType(T d){
    size_t size = sizeof(T);
    for (size_t i = 0; i < size; ++i) {
        data.push_back(static_cast<uint8_t>(d >> 8*(size-i-1)));
    }
    writePos += sizeof(T);
}
```

### 3.2.2 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.

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



simples étant réalisée, celles des objets ce fait par l'appel de la sérialisation de types simple pour chaque attribut de l'objet, sauf si l'attribut est lui-même un objet pour lequel on appellera la méthode de sérialisation adéquate, ceci jusqu'à la sérialisation complète de tout l'objet.

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.

### 3.2.3 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 methode appeler pour deserialiser le message.

Listing 2 – Méthode de désérialisation de type simple data est notre tableau dynamique d la varaible de type T à desérialiser et readPos la position de lecture du desérialiseur

```
template <typename T>
void Deserializer::deserializeAnyType(T & d){
    T res = 0;
    for (size_t i = 0; i < sizeof(T); ++i) {
        res = (res << 8) + data[readPos+i];
    }
    readPos += sizeof(T);
    d = res;
}
```

### 3.2.4 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 serialiseur garde toujours huit octets en debut 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'assigné à un deserailiseur.

## 3.3 Les structures de données communes

### 3.3.1 Tetromino

créé a quel moment ?

L'objet de la classe Tétromino contient les informations sur le tetromino actuellement en jeu :

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

Explication des getter et setter ??

La fonction Tetromino : :getCases permet de récupérer la [liste] a modifier, des vecteur représentant les coordonnées de toute les cases du tetromino en fonction de sa rotation, de sa forme et de la position de l'ancre.

### 3.3.2 Grid

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

Fonction dans Grid :

les constructeur besoin d'en parler ? IsValid ?? printGrid -> debug , besoin d'en parler ?

**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 Tétromino en jeu (tetromino qui chute et qui est contrôlé par le joueur) et un vecteur (x, y) pour savoir dans quel direction le mouvement est fait.

Cette fonction est appelé 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'autorisé où non la rotation de la pièce en jeu. La rotation peut être par exemple refusé si le tetromino est bloqué entre 2 autres tétrominos ou si il est sur le bord de la zone de jeu. Dans ces 2 cas, la rotation du tétrromino le ferait soit passé a travers d'autre pièce ce qui est impossible ou encore sortir de la zone de jeu. On vérifie donc si les case du tétrromino après rotation sont bien dans la zone de jeu et si elle sont vide 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'envoyé un nouveau tetromino en jeu. On appel la fonction `Grid : :printGrid` qui remplis le tableau en faisant appel au fonction `Tetromino : :getCases` et `Tetromino : :getType`.

**Suppression de lignes** La fonction `Grid : :deleteLines` parcourt le tableau et détecte si des lignes sont pleine. Si c'est le cas, elle appel la fonction `Grid : :fallLines` sur la lignes correspondante. Elle renvoie le nombre total de ligne pleine qui ont été supprimé pour effectué 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 vide. Si une pièce est présente dans cette zone, on estime que la zone de jeu est complètement remplis et que l'arrive d'un prochain tétramino est impossible, dans ce cas elle renvoie true.

**clear (titre a changer)** Si l'appel à la fonction `Grid : :gameOver` nous retourne true alors on vide entièrement la zone de jeu et le score du joueur subit un malus (score divisé par 2). Pour vidé la zone de jeu on utilise la fonction `Grid : :clear` qui parcourt tout le tableau est passe tout les case à 0.

## 3.4 Le client graphique

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

### 3.4.1 Les structures de données locales

**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.

Toutes les cases 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 toutes les cases du tetromino sont récupérées. Pour la gestion de la chute, il nous était plus simple de n'avoir qu'une seule 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.

Ces 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 utilisée par le client. Elle contient tous 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 affichées).

On récupère les données stockées dans le tableau de la zone de jeu (`Grid`) à partir de la 5ème ligne et on modifie les sprites contenues dans le tableau de `gameArea`.

La fonction `GameArea : :setScale` utilise la classe `gf : :Transformable` afin de modifier la taille de l'affichage en fonction de vecteur et ainsi nous permet de réutiliser les mêmes sprites 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 paramètres.

Les scores des 2 joueurs sont à 0. Toutes les cases des grilles de zone de jeu des 2 joueurs sont initialisées à 0. Le booléen « `enPartie` » de notre boucle de jeu est passé à `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 joueurs quitte la partie. Le timer tombe à zéro. Pendant la boucle de jeu le client :

- Reçoit les messages du serveur
- Récupère les actions du joueur
- Vérifie que les actions faites par le joueur sont réalisables et met à jour la position et la rotation du tetromino en jeu

— Si le tetromino est posé alors il envoie un message au serveur pour lancer le prochain tetromino

Une fois toutes les données mises à jour, l’affichage est réalisé et la boucle suivante commence

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

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

### 3.4.2 La fenêtre graphiques

Les différents éléments composant notre écran de jeu :

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

En jaune : Zone du joueur de l’adversaire. On ne voit pas la pièce en jeu mais on voit les pièces déjà posées. 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 créées grâce à la 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 petite 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

### 3.4.3 Les actions du joueur

Lors de la chute du tetromino en jeu, le joueur peut interagir sur celui-ci avec les touches du clavier :

Les flèches gauche et droite et les touches 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 tourner 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 contrôles.

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

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

Pour les actions liées 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 stopper toutes les actions en cours. Nous l'utilisons dans la boucle de jeu.

+ d'explication sur le fonctionnement des classes Action et Event ?

## Conclusion

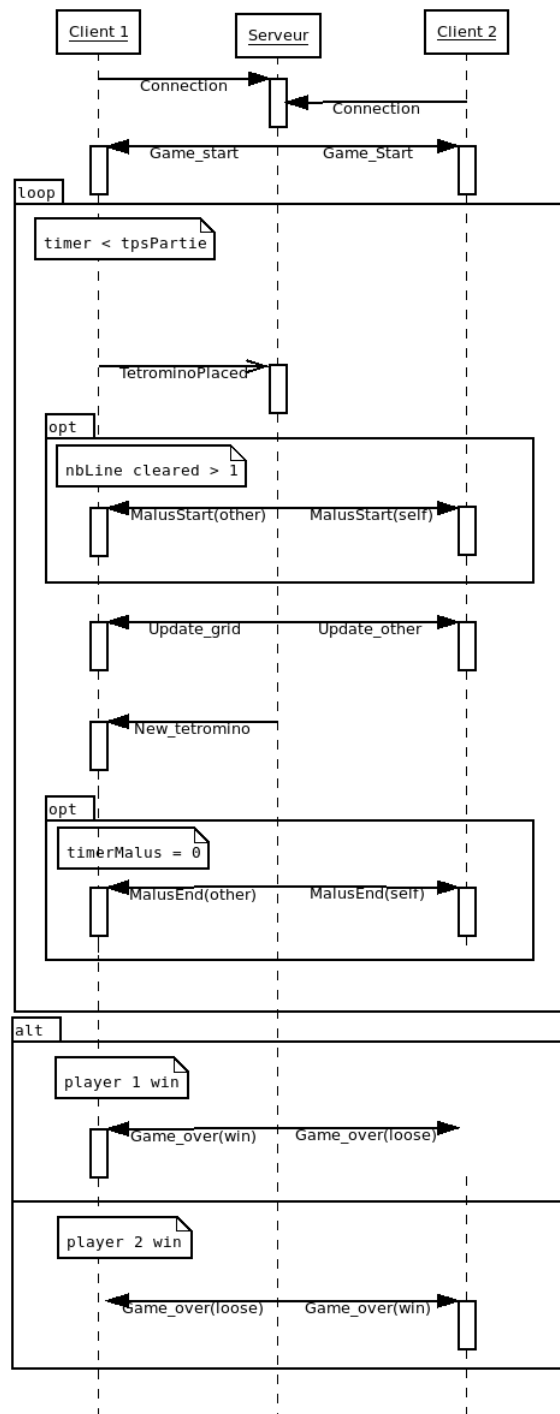


FIGURE 2 – Représentation simplifiée des échanges de messages entre les clients et le serveur

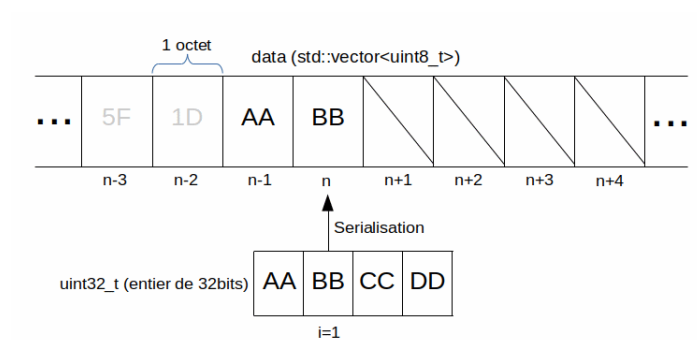


FIGURE 3 – Représentation simplifiée de la sérialisation d'un entier de 32bit, est ici représenté la sérialisation du second octet avec la position d'écriture égale à n.

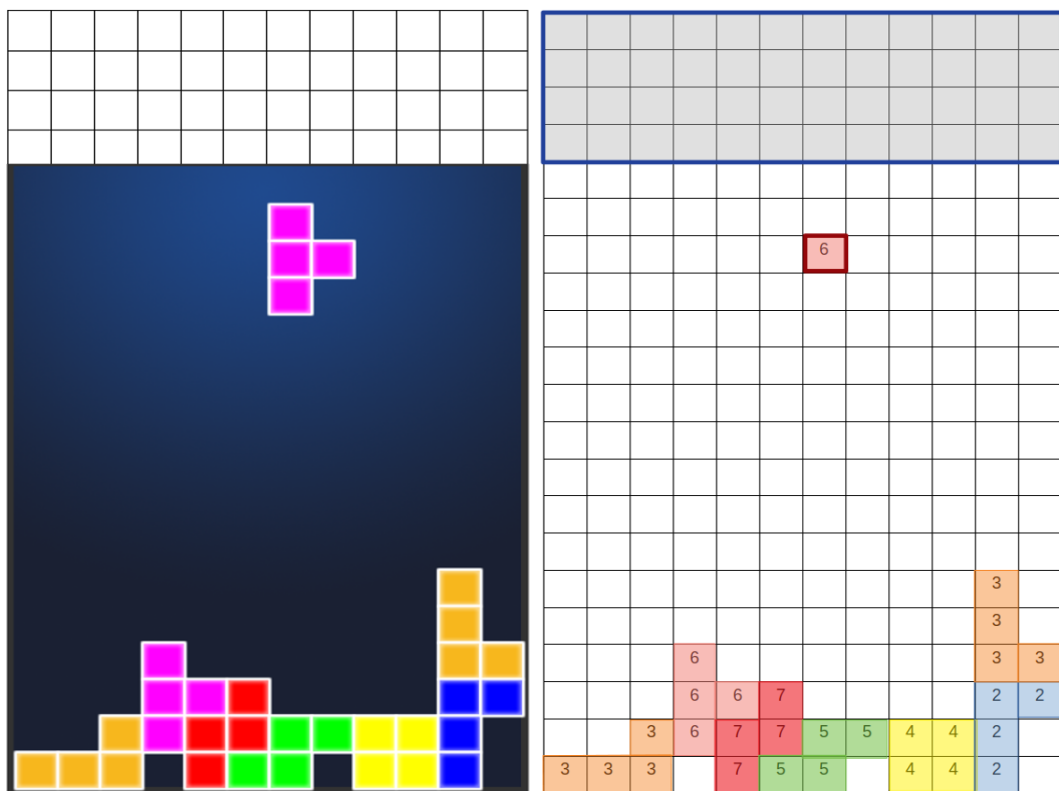


FIGURE 4 – Représentation numérique des données dans le tableau de jeu



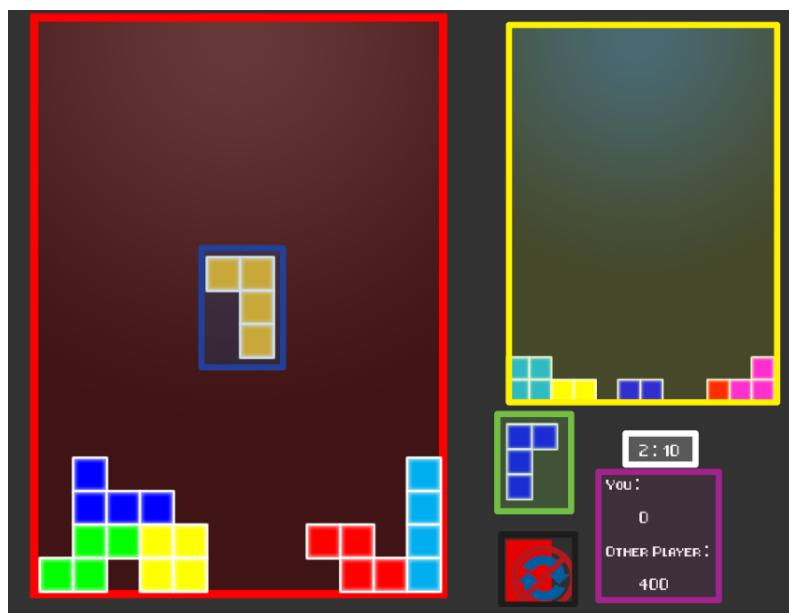


FIGURE 5 – Différentes zone d’affichage de la fenêtre du jeu