Rapport Final

Concours de Tetris

CentraleSupélec

Élèves du groupe 78
Mathieu Ha-Sum
David Ratiney
Thomas Wannegue

Chargée de projet
Lina Ye

Sommaire

Introduction

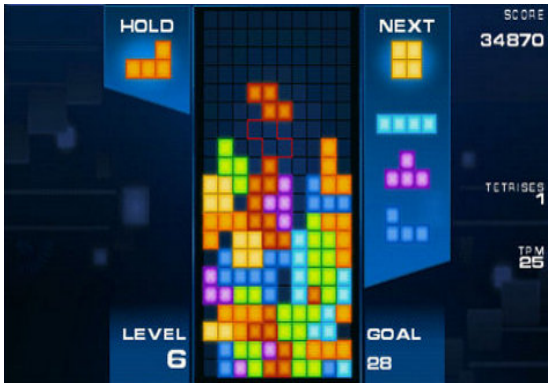
1 Brève histoire du jeu

Tetris est un jeu vidéo de puzzle de tuiles, initialement conçu et programmé par le designer de jeux russe Alexey Pajitnov en 1984. [2] Il tire son nom du préfixe numérique grec tetra- (toutes les pièces du jeu contiennent quatre segments) et du sport favori de Pajitnov, le tennis.

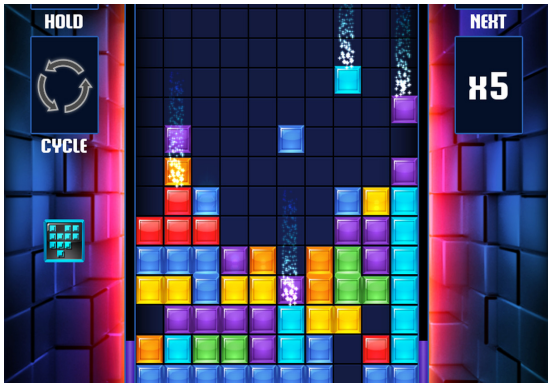
Bâti sur des règles simples et exigeant intelligence et adresse, il est considéré comme un des grands classiques de l'histoire du jeu vidéo aux côtés de Pong, Space Invaders ou encore Pac-Man. Le jeu est adapté sur la plupart des supports de jeu, aussi bien sur ordinateurs que sur consoles de jeux et téléphones mobiles. Certaines versions apportent des variantes comme un affichage 3D, un système de réserve ou encore du jeu multijoueur.

(a) Première version de Tétris (1984)

(b) Tétris sur IBM PC (1986)



(c) Tétris sur PSP (2008)



(d) Tétris Blitz pour smartphones (2013)

Figure 1 : Tétris est un grand classique de l'histoire des jeux vidéo

2 Enjeux et cahier des charges

Dans le cadre de ce projet logiciel, l'objectif est de concevoir (diagrammes de classes...) et d'implémenter le jeu en Java (architecture MVC), ainsi que développer une intelligence artificielle (IA) et de l'intégrer au programme. Lors de la présentation finale, une compétition sera organisée entre les joueurs artificiels des différents groupes.

3 Ressources en ligne

Le succès et la popularité du jeu Tétris ainsi que l'enthousiasme des joueurs font qu'une très large quantité de ressources, codes, tutoriels sont disponibles sur internet. Nous n'avons pas hésité à nous inspirer de ces nombreuses documentations lors de la conception de notre projet. Si l'ensemble de la bibliographie est disponible à la fin de ce rapport, on citera en particulier :

- Le tutoriel du site Zetcode [1] qui a fortement guidé notre conception et implémentation du jeu.

I Conception du jeu Tétris

1 Généralités

Tetris est principalement composé d'un champ de jeu où des pièces de formes différentes, appelées « **Tétriminos** », descendent du haut de l'écran [2]. Durant cette descente le joueur peut uniquement **déplacer les pièces latéralement et leur faire effectuer une rotation** sur elles-mêmes jusqu'à ce qu'elles touchent le bas du champ de jeu ou une pièce déjà placée. Le joueur ne peut ni ralentir la chute des pièces, ni l'empêcher, mais il peut dans certaines versions l'**accélérer**.

Le but pour le joueur est de réaliser le plus de lignes possibles, afin de garder de l'espace pour placer les futures pièces. Une fois une ligne complétée, elle **disparaît**, et les blocs placés au-dessus chutent d'un rang.

A chaque moment, le joueur peut connaître le Tétrimino suivant le coup actuel grâce à une indication de l'interface, et ainsi anticiper le placement de celui-ci.

Si le joueur ne parvient pas à faire disparaître les lignes suffisamment vite, l'écran peut alors se remplir jusqu'en haut. Lorsqu'un Tétrimino dépasse du champ de jeu, et empêche l'arrivée de Tétriminos supplémentaire, la partie se **termine**. Le joueur obtient un score, qui dépend essentiellement du nombre de lignes réalisées lors de la partie. Compléter plusieurs lignes d'un coup rapporte davantage de points.

Le jeu ne se termine donc jamais par la victoire du joueur. Avant de perdre, le joueur doit tenter de compléter un maximum de lignes.

Il y a quelques règles supplémentaires très populaires que l'on retrouve dans de très nombreuses réalisations du jeu :

- **Stocker un Tétrimino** : Permet de garder un Tétrimino, et de le restaurer en échangeant le Tétrimino en cours par le Tétrimino stocké.
- **Vitesse Croissante** : La vitesse de chute augmente au fil de la partie pour la faire croître la difficulté au cours de la partie. On peut également ajuster la difficulté à travers la vitesse de chute initiale.

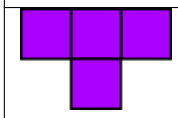
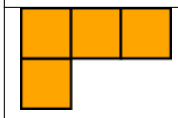
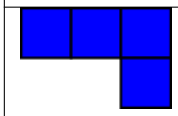
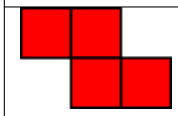
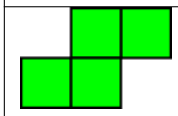
Dans un premier temps, notre implémentation ne prendra pas en compte ces additions, afin de se concentrer sur le coeur du jeu.

2 Champ de jeu

Il s'agit de l'espace dans lequel tombent les pièces. Il dispose d'une grille en arrière-plan, dont les cases sont de la même grandeur que les carrés des Tétriminos, que celles-ci suivant dans leur chute. Les Tétriminos chutent à partie du haut dans ce champ, avec une vitesse qu'on aura déterminée.

La taille du champ de jeu en norme est déterminée en cases (pas en pixels), et sera de **dix cases de largeur et de vingt-deux ou vingt cases de hauteur**.

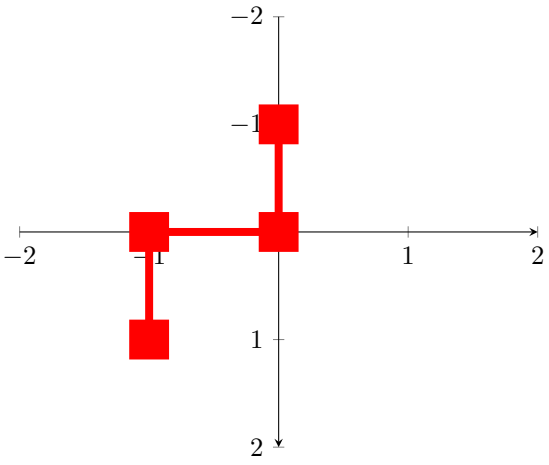
3 Représentation des pièces

Forme	Appellation	Construction
	**Tétrimino I**	Quatre carrés alignés.
	**Tétrimino O**	Méta-carré de 2x2.
	**Tétrimino T**	Trois carrés en ligne et un carré sous le centre
	**Tétrimino L**	Trois carrés en ligne et un carré sous le côté gauche.
	**Tétrimino J**	Trois carrés en ligne et un carré sous le côté droit.
	**Tétrimino Z**	Méta-carré de 2x2, dont la rangée supérieure est glissée d'un pas vers la gauche.
	**Tétrimino S**	Méta-carré de 2x2, dont la rangée supérieure est glissée d'un pas vers la droite.

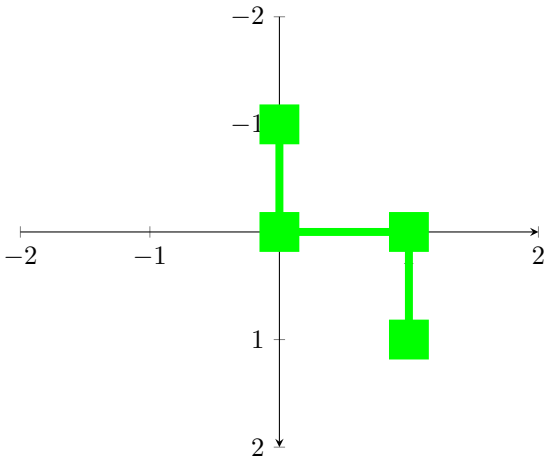
Pour représenter les pièces dans notre programme java, on utilise les coordonnées des quatre cases qui les composent.

Forme	Coordonnées (initiales)
Tétrimino I	`{ { 0, -1 }, { 0, 0 }, { 0, 1 }, { 0, 2 } }`
Tétrimino O	`{ { 0, 0 }, { 1, 0 }, { 0, 1 }, { 1, 1 } }`
Tétrimino T	`{ { -1, 0 }, { 0, 0 }, { 1, 0 }, { 0, 1 } }`
Tétrimino L	`{ { -1, -1 }, { 0, -1 }, { 0, 0 }, { 0, 1 } }`
Tétrimino J	`{ { 1, -1 }, { 0, -1 }, { 0, 0 }, { 0, 1 } }`
Tétrimino Z	`{ { 0, -1 }, { 0, 0 }, { -1, 0 }, { -1, 1 } }`
Tétrimino S	`{ { 0, -1 }, { 0, 0 }, { 1, 0 }, { 1, 1 } }`

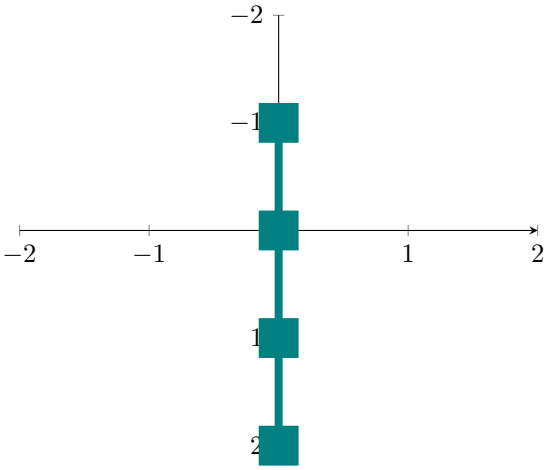
Ainsi, la chute et les actions de rotations des Tétriminos seront représentées par des translations ou symétrie sur les coordonnées.



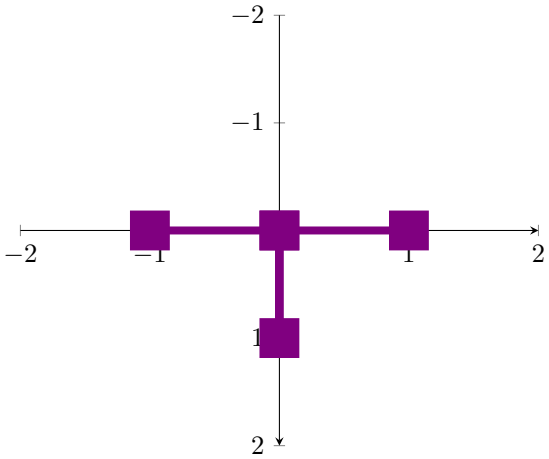
(a) Tétrimino Z



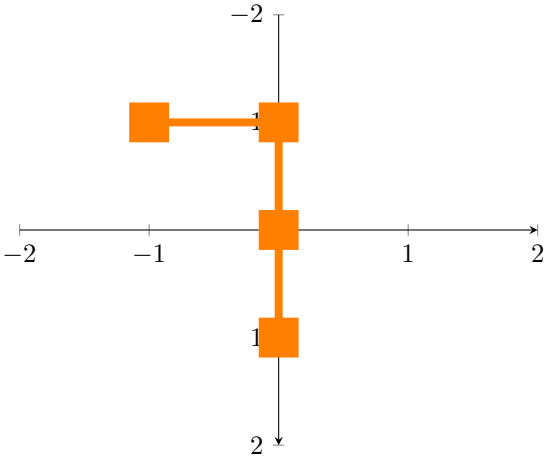
(b) Tétrimino S



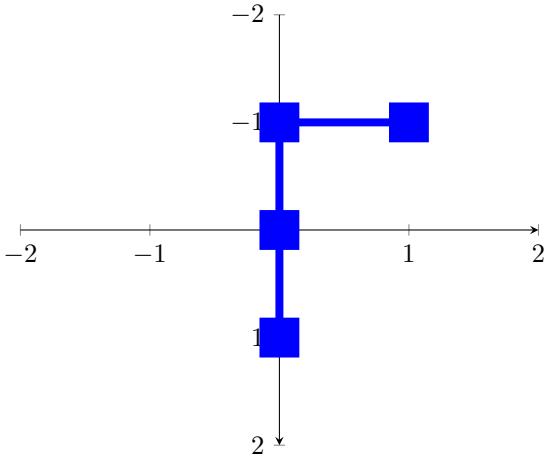
(c) Tétrimino I



(d) Tétrimino T

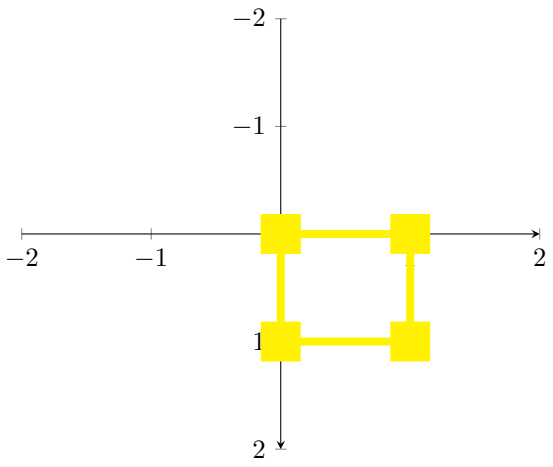


(e) Tétrimino L



(f) Tétrimino J

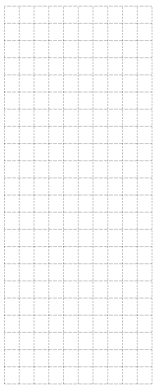
Figure 2 : Représentation des Tétriminos par 4 coordonnées (positions initiales)



Tétrimino O

4 Représentation de l'état

Pour représenter le jeu, on stocke l'état du plateau dans un tableau `etatPlateau` qui contient l'état de toutes les cases. On pourra choisir `etatPlateau` à 1 dimension avec la transposition $(x, y) \Rightarrow$ `etatPlateau`$[y * nbCasesLargeur + x]$



Par commodité pour l'utilisation de la bibliothèque Swing, l'origine (0,0) est située en haut à gauche.

On représente la pièce actuelle (en train de chuter) par les coordonnées de sa représentation `coords` et sa position dans le plateau `positionX` et `positionY`. Le jeu se déroule à une fréquence déterminée. A chaque "tic", le Tétrimino chute d'une case. Lorsqu'il atteint le fond du puits, on l'ajoute au tableau `etatPlateau` et on génère une nouvelle pièce. On vérifie également si il y a des ligne complétées à supprimer, ou si la partie est terminée (on atteint le sommet du puits).

5 Actions du joueur

Le joueur peut utiliser le clavier pour effectuer les actions suivantes :

- **Déplacer** le tétrimino actuel à gauche ou à droite
- Effectuer une **rotation** du tétrimino actuel
- Eventuellement **accélérer** la chute de la pièce, ou bien la faire **tomber instantanément**

6 Architecture MVC

On choisit l'architecture MVC pour implémenter le jeu.

- Le **modèle** représente les tétriminos, leurs propriétés et méthodes.
- Le **contrôleur** vérifie la validité des opérations du joueur et gère le déroulement de la partie.
- La **vue** permet l'interaction avec le joueur en représentant graphiquement le plateau et en écoutant les inputs du joueur.

7 Diagrammes UML

Le diagramme de classes représente l'ensemble des classes à implémenter pour le jeu Tétris.

- **Main** Classe principale qui démarre le jeu.
- **Fenetre** Implémentation de "JFrame" qui permet d'afficher une fenêtre. Fait parti de la vue.
- **VuePlateau** Implémentation de "JPanel" qui permet d'afficher le plateau de jeu et de capter les actions du joueur. Fait parti de la vue.
- **ControleurPlateau** La classe contrôleur, qui gère l'état du jeu.
- **Tetrimino** Classe qui représente les pièces du jeu Tétris et ses propriétés ainsi que ses méthodes (rotation...). Fait parti du modèle.

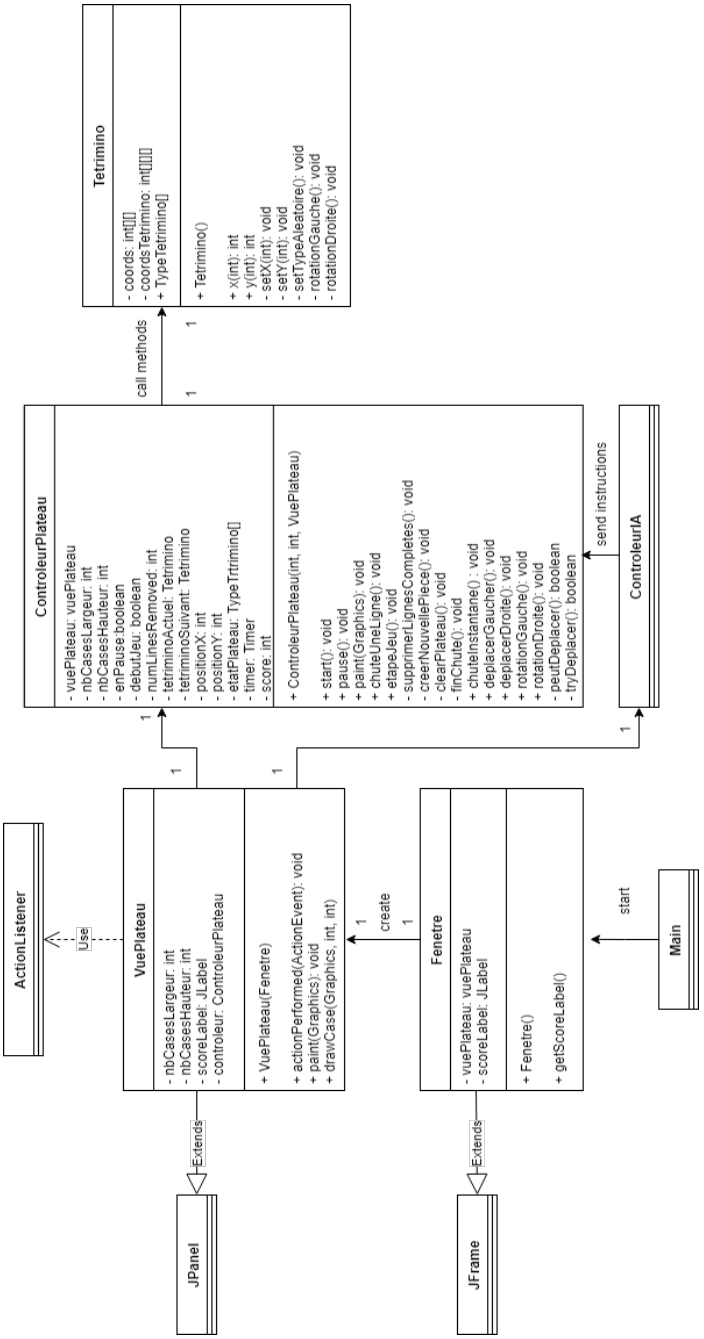


Figure 3 : Digrammes de Classes

Les diagrammes de séquences représentent l'ordre dans lequel interviennent les classes dans différents cas.

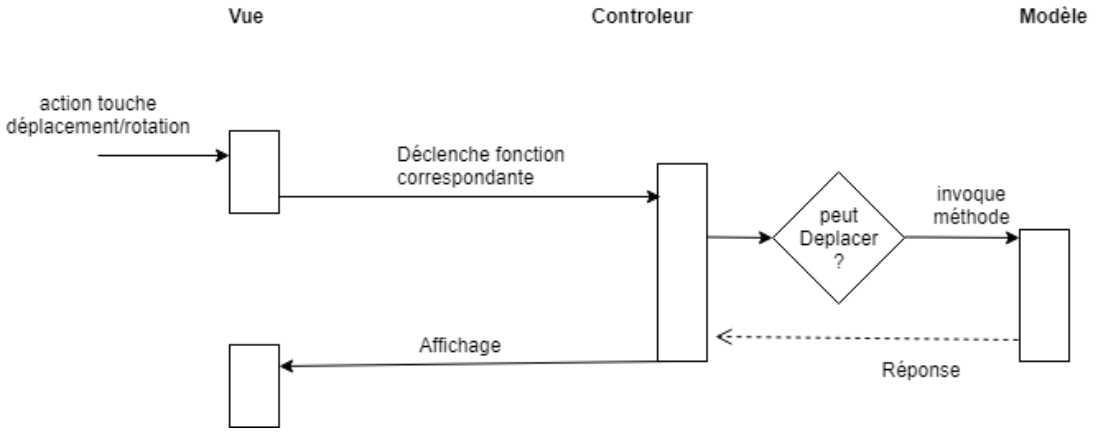


Figure 4 : Digrammes de Séquence : Action du joueur

Lorsqu'un joueur presse une touche du clavier, la classe `vuePlateau` détecte l'action et appelle la fonction correspondante de `controleurPlateau`, qui fait appel aux méthodes du modèle `Tetrimino`.

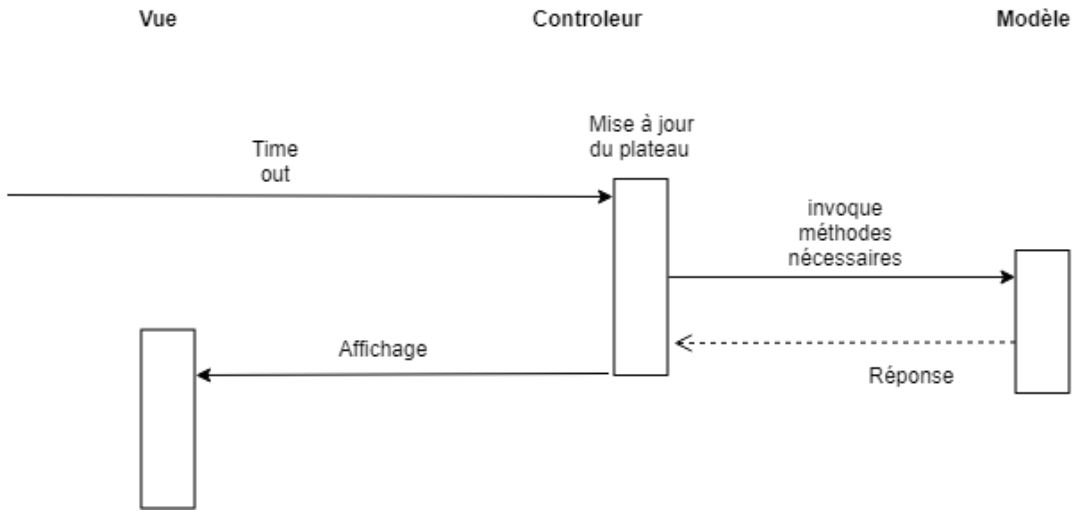


Figure 5 : Digrammes de Séquence : Déroulement du jeu

Lors d'une partie de Tetris, le plateau se met à jour à chaque action du joueur (rotation,

déplacement) mais aussi périodiquement, à chaque fois que le Tétrimino chute d'une ligne. La variable `timer` permet de déclencher la mise à jour du plateau toutes les réalisations d'une période. On peut ajuster la difficulté du jeu en augmentant cette fréquence (les Tétriminos tombent plus rapidement).

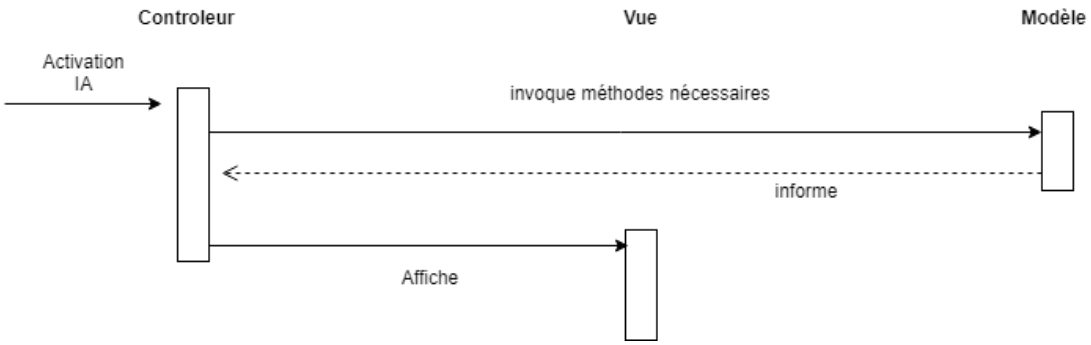


Figure 6 : Digrammes de Séquence : IA

Lorsqu'on laisse la place de joueur à l'intelligence artificielle, celle-ci communique directement avec le contrôleur (et n'a pas besoin de la vue). On met à jour la vue pour que les utilisateurs réels (nous) puissions voir les actions de l'IA.

8 Fiche: Ensemble des choix de conception

Représentation des Tétriminos Un Tétrimino est représenté par sa classe modèle `Tetrimino`. Il est caractérisé par 4 points en 2 dimensions et son type (Vide, S, Z, I, T, O, L, J).

Représentation de l'état Une variable `etatPlateau`, tableau de dimension 10x22 qui contient des "types", permet de garder l'état actuel du plateau (Tétriminos déjà tombés). Chaque type est associé à une couleur, ce qui permet de dessiner le plateau de jeu.

Tétrimino en cours Le contrôleur garde dans une variable le type de Tétrimino en train de tomber et sa position actuelle `X` et `Y`.

Tétrimino suivant Le contrôleur garde dans une variable le type de Tétrimino suivant.

Déroulement du jeu Un timer permet de déclencher la chute du Tétrimino toutes les périodes. Si le Tétrimino atteint le fond du plateau, il est ajouté à `etatPlateau`. On supprime les lignes complètes. On vérifie que la partie n'est pas terminée. Enfin, on crée un nouveau Tétrimino.

Actions du joueur A chaque action du joueur, on réalise le déplacement ou la rotation si possible et on dessine le plateau.

Calcul du score Le score est calculé de la manière suivante : +40 pour une ligne, +100 pour deux lignes, +300 pour trois lignes, +1200 pour quatre lignes.

Architecture MVC Modèle-Vue-Contrôleur

II Implémentation en Java

1 Modèle

```
public class Tetrimino
```

On liste l'ensemble des types et des coordonnées possibles.

```
// Tous les types de Tetriminos possibles
public enum TypeTetrimino {
    Vide, Z, S, I, T, O, L, J
}

// Toutes les coordonnees possibles
private int[][][] coordsTetrimino = new int[][][]{
        {{0, 0}, {0, 0}, {0, 0}, {0, 0}}, // Vide
        {{0, -1}, {0, 0}, {-1, 0}, {-1, 1}}, // Type Z
        {{0, -1}, {0, 0}, {1, 0}, {1, 1}}, // Type S
        {{0, -1}, {0, 0}, {0, 1}, {0, 2}}, // Type I
        {{-1, 0}, {0, 0}, {1, 0}, {0, 1}}, // Type T
        {{0, 0}, {1, 0}, {0, 1}, {1, 1}}, // Type O
        {{-1, -1}, {0, -1}, {0, 0}, {0, 1}}, // Type L
        {{1, -1}, {0, -1}, {0, 0}, {0, 1}} // Type J
};
```

Ainsi, on peut définir un type de Tétrimino et lui associer les coordonnées correspondantes.

```
// Definit le Tetrimino actuel, et charge ses coordonnees.
public void setType(TypeTetrimino type) {
    for (int i = 0; i < 4; i++) {
        System.arraycopy(coordsTetrimino[type.ordinal()][i], 0, coords[i], 0, 2);
    }
    this.type = type;
}
```

Ce qui se révèle utile pour créer un nouveau Tétrimino aléatoirement.

```java
// Definit aleatoirement un Tetrimino
public void setTypeAleatoire() {
    Random r = new Random();
    int x = Math.abs(r.nextInt()) % 7 + 1;
    TypeTetrimino[] values = TypeTetrimino.values();
    setType(values[x]);
}
```

On doit également pouvoir facilement accéder aux coordonées du Tétrimino, on créer donc les méthodes correspondantes.

```java
private void setX(int index, int x) {
    coords[index][0] = x;
}

private void setY(int index, int y) {
    coords[index][1] = y;
}

public int x(int index) {
    return coords[index][0];
}

public int y(int index) {
    return coords[index][1];
}
```

Par ailleurs, on a besoin des méthodes de rotations, qui peuvent être implémentées ainsi.

```
public int minX() {
        int m = coords[0][0];
        for (int i = 0; i < 4; i++) {
            m = Math.min(m, coords[i][0]);
        }
        return m;
    }


    public int minY() {
        int m = coords[0][1];
        for (int i = 0; i < 4; i++) {
            m = Math.min(m, coords[i][1]);
        }
        return m;
    }

    // Rotation Gauche
    public Tetrimino rotationGauche() {
        if (type == TypeTetrimino.O)
            return this;

        Tetrimino result = new Tetrimino();
        result.type = type;

        for (int i = 0; i < 4; ++i) {
            result.setX(i, y(i));
            result.setY(i, -x(i));
        }
        return result;
    }

    // Rotation Droite
    public Tetrimino rotationDroite() {
        if (type == TypeTetrimino.O)
            return this;

        Tetrimino result = new Tetrimino();
        result.type = type;

        for (int i = 0; i < 4; ++i) {
            result.setX(i, -y(i));
            result.setY(i, x(i));
        }
        return result;
    }
```

2 Vue

```
public class Fenetre extends JFrame
```

La fenêtre affiche un score et le plateau.

```
public Fenetre() {
    // Reglage des dimensions
    SCALE = 1;

    largeurFenetreJeu = (int) (500 * SCALE);
    hauteurFenetreJeu = (int) (1100 * SCALE);
    largeurMenu = largeurFenetreJeu;

    scoreLabel = new JLabel("");
    vuePlateau = new VuePlateau(this);
}
```

On ajuste la taille des fenêtre en fonction de la résolution de l'écran. On ajoute aussi quelques couleurs pour l'affichage du score. On a besoin d'une méthode pour passer le score à `vuePlateau`, responsable de mettre à jour le score.

```
public void init() {
        setLayout(new BorderLayout());

        add(scoreLabel, BorderLayout.SOUTH);
        scoreLabel.setBackground(Color.GRAY);
        scoreLabel.setForeground(Color.WHITE);
        scoreLabel.setOpaque(true);
        scoreLabel.setFont(new java.awt.Font("Open", Font.BOLD, (int) (60*SCALE)));

        add(vuePlateau, BorderLayout.CENTER);
        vuePlateau.start();

        setPreferredSize(new Dimension(largeurFenetreJeu + largeurMenu,
            hauteurFenetreJeu));
        setTitle("Tetris");
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        pack();
        setVisible(true);
        setResizable(false);
    }

    JLabel getScoreLabel() {
        return scoreLabel;
    }

    int getLargeurMenu() {
        return largeurMenu;
    }
}
```

Le reste de la vue est gérée par la classe `vuePlateau`.

```
public class VuePlateau extends JPanel implements ActionListener
```

On initialise le plateau, en créant notamment le contrôleur associé.

```
    private final int nbCasesLargeur = 10;
    private final int nbCasesHauteur = 22; // (On peut aussi choisir 20)
    private JLabel scoreLabel;
    private int largeurMenu;
    private ControleurPlateau controleur;
    private ControleurIA IA;

    VuePlateau(Fenetre parent) {
        setFocusable(true);
        controleur = new ControleurPlateau(nbCasesLargeur, nbCasesHauteur, this); //
            Creation du controleur
        IA = new ControleurIA(controleur, nbCasesLargeur, nbCasesHauteur); // Creation du
            controleur de l'IA

        scoreLabel = parent.getScoreLabel();
        largeurMenu = parent.getLargeurMenu();

        addKeyListener(new clavierAdapter());
    }
```

Entre autre, `vuePlateau` est responsable de dessiner le plateau. On dessine chaque case individuellement avec la couleur associé au type.

```
// Dessine chaque case en fonction du type de Tetrimino
    public void drawCase(Graphics g, int x, int y, Tetrimino.TypeTetrimino type) {
        Color colors[] = {
                new Color(0, 0, 0),     // Vide
                new Color(204, 102, 102), // Z
                new Color(102, 204, 102), // S
                new Color(27, 138, 122), // I
                new Color(102, 102, 204), // T
                new Color(204, 204, 102), // O
                new Color(218, 170, 0), // L
                new Color(87, 164, 204),// J
        };

        // Choix de la couleur appropriee
        Color color = colors[type.ordinal()];

        g.setColor(color);
        g.fillRect(x + 1, y + 1, largeurCase() - 2, hauteurCase() - 2);

        // Effet Ombre (Clair : gauche, haut)
        g.setColor(color.brighter());
        g.drawLine(x, y + hauteurCase() - 1, x, y);
        g.drawLine(x, y, x + largeurCase() - 1, y);

        // Effet Ombre (Sombre : droite, bas)
        g.setColor(color.darker());
        g.drawLine(x + 1, y + hauteurCase() - 1,
                x + largeurCase() - 1, y + hauteurCase() - 1);
        g.drawLine(x + largeurCase() - 1, y + hauteurCase() - 1,
                x + largeurCase() - 1, y + 1);
    }
```

D'autre part, on dessine aussi le menu (qui contient entre autre le Tétrimino suivant) et la prévisualisation du Tétrimino actuel

```
// Dessine la previsualisation de la chute de la piece
    public void drawPreview(Graphics g, int x, int y) {
        g.setColor(new Color(200, 200, 200));
        g.drawLine(x, y + hauteurCase() - 1, x, y);
        g.drawLine(x, y, x + largeurCase() - 1, y);
        g.drawLine(x + 1, y + hauteurCase() - 1,
                x + largeurCase() - 1, y + hauteurCase() - 1);
        g.drawLine(x + largeurCase() - 1, y + hauteurCase() - 1,
                x + largeurCase() - 1, y + 1);
    }

    // Dessine le menu
    public void drawMenu(Graphics g) {
        g.setColor(Color.GRAY);
        g.fillRect(nbCasesLargeur * largeurCase(), 0, largeurMenu, hauteurCase() *
            nbCasesHauteur);
        g.setColor(Color.WHITE);

        g.fillRect(nbCasesLargeur * largeurCase() + 2*largeurCase(), 2*largeurCase(),
            5*largeurCase(), 5*hauteurCase());
    }
    public void setScoreLabel(String text) {
        scoreLabel.setText(text);
    }
```

La `vuePlateau` doit également capter les actions du joueur et appeler la fonction correspondante.

```
private class clavierAdapter extends KeyAdapter {
       public void keyPressed(KeyEvent e) {

           if (!controleur.getDebutJeu() || controleur.tetriminoEstVide()) {
               return;
           }

           int keycode = e.getKeyCode();

           if (keycode == 'p' || keycode == 'P') {
               controleur.pause();
               return;
           }

           if (controleur.getEnPause())
               return;

           switch (keycode) {
               case KeyEvent.VK_LEFT:
                   controleur.deplaceGauche();
                   break;
               case KeyEvent.VK_RIGHT:
                   controleur.deplaceDroite();
                   break;
               case KeyEvent.VK_DOWN:
                   controleur.rotationDroite();
                   break;
               case KeyEvent.VK_UP:
                   controleur.rotationGauche();
                   break;
               case KeyEvent.VK_SPACE:
                   controleur.chuteInstantane();
                   break;
               case 'd':
                   controleur.chuteUneLigne();
                   break;
               case 'D':
                   controleur.chuteUneLigne();
                   break;
               // Toutes les touches pour declencher les actions IA
               case '1':
                   IA.joue1ActionAleatoire();
                   break;
           // ...
               case '8':
                   IA.joue1CoupAvecAnticipationIndefiniment();
                   break;
           }
       }
   }
```

3 Contrôleur

```
public class ControleurPlateau
```

Le contrôleur permet de gérer l'état de la partie.

```
    // Parametres du Jeu
    public VuePlateau vuePlateau;
    // Variables de la partie en cours
    public int iteration = 0;
    public int nblignesup;
    public boolean finChute = false;
    public Tetrimino tetriminoActuel;
    public Tetrimino.TypeTetrimino[] etatPlateau;
    public Tetrimino.TypeTetrimino[] saveetatPlateau;
    public boolean iamode = false;
    public Tetrimino.TypeTetrimino savetype;
    private int nbCasesLargeur;
    private int nbCasesHauteur;
    private boolean debutJeu = false;
    private boolean enPause = false;
    private int positionX = 0;
    private int positionY = 0;
    private Timer timer;
    private int score = 0;
    private int max = 0;
    private int nbrot1 = 0;
    private int indicedep1 = 0;
    private int nbrot2 = 0;
    private int indicedep2 = 0;


    // Constructeur
    public ControleurPlateau(int nbCasesLargeur, int nbCasesHauteur, VuePlateau
         vuePlateau) {
        this.nbCasesLargeur = nbCasesLargeur; // 10
        this.nbCasesHauteur = nbCasesHauteur; // 22
        this.vuePlateau = vuePlateau; // Vue Plateau associee
        tetriminoActuel = new Tetrimino(); // Tetrimino en cours
        timer = new Timer(400, vuePlateau); // Frequence du jeu
        etatPlateau = new Tetrimino.TypeTetrimino[nbCasesLargeur * nbCasesHauteur]; //
            Etat du plateau, (x,y) => y*largeur + x
        saveetatPlateau = new Tetrimino.TypeTetrimino[nbCasesLargeur * nbCasesHauteur]; //
            Etat du plateau, (x,y) => y*largeur + x
        timer.start();
        clearPlateau();

    }
```

A chaque réalisation du timer, le jeu se met à jour en faisant chuter le Tétrimino ou bien en en créant un nouveau si la chute est terminée.

```
    // Le jeu se met a jour a chaque tic du timer
    public void etapeJeu() {
        if (finChute) {
            finChute = false;
            creerNouveauTetrimino();
        } else {
            chuteUneLigne();
        }
    }

    private void creerNouveauTetrimino() {
        tetriminoActuel.setTypeAleatoire();
        positionX = nbCasesLargeur / 2 + 1;
        positionY = -tetriminoActuel.minY() ;

        if (!tryDeplacer(tetriminoActuel, positionX, positionY)) {
            tetriminoActuel.setType(Tetrimino.TypeTetrimino.Vide);
            timer.stop();
            debutJeu = false;
            vuePlateau.setScoreLabel("Game Over");
        }
    }
```

Avant de déplacer un Tétrimino, il faut vérifier qu'il y a encore de la place.

```
// Controle des deplacements et rotations aux limites
    private boolean peutDeplacer(Tetrimino tetrimino, int newX, int newY) {
        for (int i = 0; i < 4; ++i) {
            int x = newX + tetrimino.x(i);
            int y = newY + tetrimino.y(i);
            if (x < 0 || x >= nbCasesLargeur || y < 0 || y >= nbCasesHauteur)
                return false;
            if (getTypeAt(x, y) != Tetrimino.TypeTetrimino.Vide)
                return false;
        }
        return true;
    }
    private boolean tryDeplacer(Tetrimino newPiece, int newX, int newY) {
        if (!peutDeplacer(newPiece, newX, newY)){return false;}
        tetriminoActuel = newPiece;
        positionX = newX;
        positionY = newY;
        vuePlateau.repaint();
        return true;
    }
```

Lorsqu'un Tétrimino tombe, il faut mettre à jour `etatPlateau` et supprimer les lignes complètes.

```
private void finChute() {
       for (int i = 0; i < 4; ++i) {
           int x = positionX + tetriminoActuel.x(i);
           int y = positionY + tetriminoActuel.y(i);
           etatPlateau[(y * nbCasesLargeur) + x] = tetriminoActuel.getType();
       }

       supprimerLigneCompletes();

       if (!finChute)
           creerNouveauTetrimino();
   }

   // Supprime les lignes completees
   private void supprimerLigneCompletes()
   {
       int nbLignesCompletes = 0;

       for (int i = 0 ; i < nbCasesHauteur; ++i) {
           boolean estComplet = true;

           for (int j = 0; j < nbCasesLargeur; ++j) {
               if (getTypeAt(j, i) == Tetrimino.TypeTetrimino.Vide) {
                   estComplet = false;
                   break;
               }
           }

           if (estComplet) {
               ++nbLignesCompletes;
               for (int k = i; k > 0; --k) {
                   for (int j = 0; j < nbCasesLargeur; ++j)
                       etatPlateau[(k * nbCasesLargeur) + j] = getTypeAt(j, k - 1); //
                           Translation des cases restantes
               }
           }
       }

       if (nbLignesCompletes > 0) {
           if (nbLignesCompletes == 1) {
               score = score + 40;
           }
           if (nbLignesCompletes == 2) {
               score = score + 100;
           }
           if (nbLignesCompletes == 3) {
               score = score + 300;
           }
           if (nbLignesCompletes == 4) {
               score = score + 1200;
           }
           vuePlateau.setScoreLabel("Score : " + String.valueOf(score));
           finChute = true;
           tetriminoActuel.setType(Tetrimino.TypeTetrimino.Vide);
           vuePlateau.repaint();
       }
   }
   }
```

D'autre part, à chaque actualisation il faut dessiner le plateau, le Tétrimino en cours (facultatif mais pratique : on peut aussi dessiner la prévisualisation de la fin de chute).

```
// Retourne le type de Tetrimino en (x,y)
    private Tetrimino.TypeTetrimino getTypeAt(int x, int y) {
        return etatPlateau[(y * nbCasesLargeur) + x];
    }

    public void paint(Graphics g, double largeur, double hauteur) {
        int largeurCase = (int) largeur / nbCasesLargeur;
        int hauteurCase = (int) hauteur / nbCasesHauteur;

        // Dessine toutes les pieces deja tombees
        for (int i = 0; i < nbCasesHauteur; ++i) {
            for (int j = 0; j < nbCasesLargeur; ++j) {
                Tetrimino.TypeTetrimino shape = getTypeAt(j, i);
                if (shape != Tetrimino.TypeTetrimino.Vide)
                    vuePlateau.drawCase(g, j * largeurCase,
                           i * hauteurCase, shape);
            }
        }

        // Dessine la piece en cours
        if (tetriminoActuel.getType() != Tetrimino.TypeTetrimino.Vide) {
            for (int i = 0; i < 4; ++i) {
                int x = positionX + tetriminoActuel.x(i);
                int y = positionY + tetriminoActuel.y(i);
                vuePlateau.drawCase(g, x * largeurCase,
                       y * hauteurCase,
                       tetriminoActuel.getType());
            }
        }

         // Dessine la "Previualisation" de la chute du Tetrimino
        if (tetriminoActuel.getType() != Tetrimino.TypeTetrimino.Vide) {
            int Y = positionY;
            Tetrimino previewTetrimino;
            previewTetrimino = tetriminoActuel;
            while (Y < nbCasesHauteur && peutDeplacer(previewTetrimino, positionX, Y + 1)) {
                ++Y;
            }
            for (int i = 0; i < 4; ++i) {
                int x = positionX + previewTetrimino.x(i);
                int y = Y + previewTetrimino.y(i);
                vuePlateau.drawPreview(g, x * largeurCase,
                       y*hauteurCase);
            }
        }

    }
```

Le contrôle vérifie si les déplacement du joueur sont valides et les execute le cas échéant.

```
// Actions du joueurs
public void deplaceGauche() {
    tryDeplacer(tetriminoActuel, positionX - 1, positionY);
}
public void deplaceDroite() {
    tryDeplacer(tetriminoActuel, positionX + 1, positionY);
}
public void rotationGauche() {
    tryDeplacer(tetriminoActuel.rotationGauche(), positionX, positionY);
}
public void rotationDroite() {
    tryDeplacer(tetriminoActuel.rotationDroite(), positionX, positionY);
}
```

III Intelligence Artificielle pour Tétris

1 Implémentation Neutre (aléatoire)

Afin de mettre en place des routines de jeu permettant de créer l'intelligence artificielle, nous avons construit une intelligence artificielle aléatoire. Celle-ci nous permet de voir comment structurer notre future intelligence artificielle. L'implémentation de cette IA se trouve dans la classe `ControleurIA`.

L'IA basique joue chaque pièce de manière totalement aléatoire. On choisit au hasard une position horizontale et au hasard un nombre de rotations grâce à des nombres entiers aléatoires. Enfin on affecte au Tétrimino une position horizontale ainsi qu'une rotation définis par ces nombres aléatoires.

```
public void joue1CoupAleatoire() {
       int choixRotation = (int) (Math.random() * 4);
       int choixPositionX = (int) (Math.random() * nbCasesLargeur / 2);

       for (int i = 0; i < choixRotation; i++) {
           controleurPlateau.rotationDroite();
       }

       if (Math.random() < 0.5) {
           for (int i = 0; i < choixPositionX; i++) controleurPlateau.deplaceDroite();
       } else for (int i = 0; i < choixPositionX; i++) controleurPlateau.deplaceGauche();

       controleurPlateau.chuteInstantane();
```

Notons que pour savoir si la pièce se déplace à gauche ou à droite, on utilise un autre nombre aléatoire. Si le nombre aléatoire est inférieur à 0.5 on déplace la pièce à droite, sinon elle sera déplacée à gauche.

On peut jouer aléatoire jusqu'à la fin de la partie.

```
public void joue1CoupAleatoireIndefiniment() {
       while (controleurPlateau.getDebutJeu()) {
           joue1CoupAleatoire();
           controleurPlateau.etapeJeu();
       }
   }
```

Nous avons également crée une IA qui joue aléatoirement mais légèrement différemment. Cette deuxième façon permet de créer un nombre aléatoire qui va déclencher une action. Les actions

peuvent être : déplacement à droite, déplacement à gauche, rotation droite, rotation gauche, chute instantanée. Ce n'est plus un coup aléatoire, mais chaque action qui est aléatoire.

```
public void joue1ActionAleatoire() {
       int p = (int) (Math.random() * 5);
       if (p == 0) {
           controleurPlateau.deplaceGauche();
       }
       if (p == 1) {
           controleurPlateau.deplaceDroite();
       }
       if (p == 2) {
           controleurPlateau.rotationGauche();
       }
       if (p == 3) {
           controleurPlateau.rotationDroite();
       }
       if (p == 4) {
           controleurPlateau.chuteInstantane();
       }
   }
```

De la même manière que précédemment on utilise une méthode permettant d'utiliser le principe précédant de manière répétitif.

```
public void joue1ActionAleatoireIndefinimient() {
       while (controleurPlateau.getDebutJeu()) {
           joue1ActionAleatoire();
           controleurPlateau.etapeJeu();
       }
   }
```

Sans surprise, ces IA aléatoires ont beaucoup de mal à compléter des lignes, et terminent la partie le plus souvent avec un score nul.

2 Simulations et choix du meilleur coup

2.1 Principe

Afin que l'IA puisse jouer le mieux possible à Tetris, il faut tester les différents coup possibles et choisir le meilleur. Pour cela nous disposons à chaque coup, du Tétrimino actuel ainsi que du Tétrimino suivant qui sont tous les deux connus à chaque étape.

Pour obtenir le meilleur coup possible jouable par l'IA nous avons pensé à jouer l'ensemble de tous les coups possibles en prenant en compte un ou bien deux Tétriminos (le Tétrimino actuel et le Tétrimino suivant). En quelque sorte, il s'agit d'explorer l'arbre de toutes les possibilités. Ces possibilités prennent en compte la capacité pour chaque Tétrimino de se déplacer horizontalement sur toute la longueur du plateau de jeu et de rotationner sur lui même jusqu'à 4 fois. A chaque possibilité jouée, on évalue le plateau résultant grâce à une fonction d'évaluation que l'on détaillera plus tard. Cette fonction d'évaluation sera appelée à chaque fois que l'on relance une nouvelle possibilité de coup. On choisit le meilleur coup qui correspond au plus grand score donné par la fonction d'évaluation.

Ainsi, nous avons créer deux IA : une qui permet d'obtenir le meilleur coup possible en prenant en compte seulement le Tétrimino actuel, et une autre IA prenant en compte le Tétrimino actuel et le Tétrimino suivant. Il est évident que cette dernière IA doit jouer de manière sensiblement meilleure, mais augmente considérablement la complexité calculatoire.

Remarque : Pour simplifier, nous n'avons pas pris en compte la possibilité qu'il soit possible de déplacer un Tétrimino horizontalement pendant sa chute. Ainsi chaque Tétrimino a dès le début de sa chute sa position et sa rotation initiale qu'il ne peut modifier pendant la durée de sa chute. Ceci peut être dommageable pour notre IA car si il y a possibilité de placer un tetrimino dans un renforcement de Tétriminos déjà existant sur le plateau, notre intelligence artificielle ne pourra pas prendre en compte cette possibilité.

Pour implémenter notre IA nous avons crée une nouvelle classe à part. Elle se nomme `ControleurIA`.

2.2 Quelques fonctions utiles

La procédure `miseAGauche()` :

```
public void miseAGauche() {
       for (int z = 0; z < nbCasesLargeur; ++z) {
           deplaceGauche();
       }
   }
```

Cette procédure permet à un Tétrimino de se placer sur le bord gauche du plateau de jeu. Cette procédure va être appelée à chaque fois que un Tétrimino va apparaître. On place le Tétrimino sur le bord gauche du plateau et on va pouvoir tester toutes les possibilités de déplacement horizontal du Tétrimino en déplaçant progressivement le Tétrimino vers la droite grâce à divers boucle for.

Cette procédure n'est pas forcément utile, on aurait pu déplacer la pièce à gauche ou à droite dès son apparition pour tester tout les coups possibles. Cependant il paraît plus élégant de compter le nombre de déplacements que fais un Tétrimino en partant d'un point de repère situé sur le bord gauche que sur un point de repère situé au centre du plateau.

La classe `copieControleurPlateau` :

```
class copieControleurPlateau {

        // Parametres du Jeu
        private int nbCasesLargeur;
        private int nbCasesHauteur;

        // Variables de la partie en cours
        private boolean finChute = false;
        private int positionX;
        private int positionY;
        private Tetrimino tetriminoActuel;
        private Tetrimino tetriminoSuivant;
        private Tetrimino.TypeTetrimino[] etatPlateau;


        // Constructeur
        public copieControleurPlateau(int nbCasesLargeur, int nbCasesHauteur,
            Tetrimino.TypeTetrimino[] originalEtatPlateau,
                                Tetrimino originalTetriminoActuel, Tetrimino
                                    originalTetriminoSuivant) {
            this.nbCasesLargeur = nbCasesLargeur; // 10
            this.nbCasesHauteur = nbCasesHauteur; // 22
            tetriminoActuel = new Tetrimino();
            tetriminoActuel.setType(originalTetriminoActuel.getType());
            tetriminoSuivant = new Tetrimino(); // Tetrimino suivant
            tetriminoSuivant.setType(originalTetriminoSuivant.getType());
            etatPlateau = new Tetrimino.TypeTetrimino[nbCasesHauteur * nbCasesLargeur];
            System.arraycopy(originalEtatPlateau, 0, etatPlateau, 0, nbCasesHauteur *
                nbCasesLargeur);
            positionX = nbCasesLargeur / 2;
            positionY = -tetriminoActuel.minY() +1;
        }
```

Cette classe permet de faire une copie "légère" du contrôleur afin de tester l'ensemble des coups. En effet le contrôleur représente l'état actuel de la partie. On crée donc un nouvel objet `copieControleurPlateau` à chaque nouveau coup à tester.

3 IA Sans Anticipation

On appelle l'IA sans anticipation l'IA qui ne prend en compte que le Tétrimino actuel pour évaluer le meilleur coup.

Pour implémenter cette fonction, on crée la méthode `joue1Coup()`. On initialise des variables qui vont permettre de déterminer le meilleur score et le nombre de déplacements à droite et le nombre de rotations liés à ce meilleur score. On crée une copie du contrôleur actuel. Cela permettra de tester et évaluer les possibilités de positionnement de Tétrimino.

On balaye pour le Tétrimino toutes les possibilités de rotation, puis toutes les possibilités de déplacement à droite (car nous avons placé la pièce initialement grâce à la procédure `miseAGauche()`). On crée alors une copie du contrôleur actuel et place la pièce en bas du plateau grâce à la procédure `chuteInstantane()`. **C'est sur le plateau** `copieControleur` **que l'on teste un nouveau coup.**

On obtient ainsi un nouveau plateau, que l'on évalue grâce à la fonction d'évaluation. Si le score donné par la fonction d'évaluation est meilleur que l'ancien meilleurs score, on met en mémoire les variables correspondant au score, au nombre de rotations et déplacements associés. On effectue cela jusqu'à avoir balayer toutes les possibilités.

Une fois toutes les possibilités explorées, on connaît le nombre de rotations et le nombre de déplacements vers la droite à faire en partant du bord de gauche du plateau pour obtenir le meilleur coup potentiel. On place alors le Tétrimino actuel sur le bord gauche avec la procédure `miseaGauche()`, et on effectue le nombre de rotations et de translations vers la droite donnés par les calculs précédents. Enfin on utilise la procédure `chuteInstantane()` pour faire tomber en bas la pièce de Tétrimino.

```
public void joue1Coup() {
        int meilleurNbRotationsDroite = 0;
        int meilleurNbDeplacementsDroite = 0;
        int meilleureEvaluation = -9999;
        int evaluation;
        copieControleurPlateau copieControleurPlateau;

        Tetrimino tetriminoActuel = controleurPlateau.getTetriminoActuel();
        Tetrimino tetriminoSuivant = controleurPlateau.getTetriminoSuivant();
        Tetrimino.TypeTetrimino[] etatPlateau = controleurPlateau.getEtatPlateau();

        for (int nbRotationsDroite = 0; nbRotationsDroite <= 3; ++nbRotationsDroite) { //
            Iterere toutes les rotations possibles
           for (int nbDeplacementsDroite = 0; nbDeplacementsDroite <= nbCasesLargeur;
               ++nbDeplacementsDroite) { // Itere tous les deplacements possibles

               // Creer une copie du controleur
               copieControleurPlateau = new copieControleurPlateau(nbCasesLargeur,
                   nbCasesHauteur, etatPlateau, tetriminoActuel, tetriminoSuivant);

               // Teste le coup sur la copie du controleur
               for (int i = 0; i < nbRotationsDroite; ++i) {
                   copieControleurPlateau.rotationDroite();
               }
               copieControleurPlateau.miseAGauche();
               for (int i = 0; i < nbDeplacementsDroite; ++i) {
                   copieControleurPlateau.deplaceDroite();
               }
               copieControleurPlateau.chuteInstantane();

               // Evalue le coup
               evaluation = evaluationPlateau(copieControleurPlateau.getEtatPlateau());

               // Sauvegarde le coup si meilleur
               if (evaluation > meilleureEvaluation) {
                   meilleureEvaluation = evaluation;
                   meilleurNbDeplacementsDroite = nbDeplacementsDroite;
                   meilleurNbRotationsDroite = nbRotationsDroite;
               }
           }
       }

       // Execute le coup avec la meilleure evaluation
       for (int i = 0; i < meilleurNbRotationsDroite; i++) {
           controleurPlateau.rotationDroite();
       }
       controleurPlateau.miseAGauche();
       for (int i = 0; i < meilleurNbDeplacementsDroite; i++) {
           controleurPlateau.deplaceDroite();
       }
       controleurPlateau.chuteInstantane();
   }
```

Pareil que précédemment, on utilise une méthode permettant de jouer indéfiniment de cette façon grâce au code suivant.

```
public void joue1CoupIndefiniment() {
    while (controleurPlateau.getDebutJeu()) {
        joue1Coup();
        controleurPlateau.etapeJeu();
    }
}
```

4 IA avec Anticipation

On appelle l'IA avec anticipation, l'IA qui prend en compte le Tétrimino actuel et le Tétrimino suivant pour évaluer le meilleur coup. Cette IA comporte une profondeur supplémentaire par rapport à l'IA sans anticipation.

Pour implémenter cette fonction, on crée la méthode du nom de `joue1CoupAvecAnticipation()`. On initialise des variables qui vont permettre de déterminer le meilleur score et le nombre de déplacements à droite et le nombre de rotations liés à ce meilleur score comme tout à l'heure.

De manière similaire, on réalise toutes les possibilités en balayant toutes les rotations possibles et les déplacements possibles vers la droite (car on place à chaque étape le Tétrimino sur le bord du plateau grâce à la procédure de `miseAGauche()`) du Tétrimino actuel mais aussi du Tétrimino suivant. Une fois les deux Tétrimino placés, on obtient ainsi un nouveau plateau, que l'on évalue grâce à la fonction d'évaluation. Si le score donné par la fonction d'évaluation est meilleur que l'ancien meilleur score, on met en mémoire les variables correspondant au score, au nombre de rotations et déplacements associés.

```
public void joue1CoupAvecAnticipation() {
       int meilleurNbRotationsDroite = 0;
       int meilleurNbDeplacementsDroite = 0;
       int meilleureEvaluation = -9999;
       int evaluation;
       copieControleurPlateau copieControleurPlateau;

       Tetrimino tetriminoActuel = controleurPlateau.getTetriminoActuel();
       Tetrimino tetriminoSuivant = controleurPlateau.getTetriminoSuivant();
       Tetrimino.TypeTetrimino[] etatPlateau = controleurPlateau.getEtatPlateau();


       for (int nbRotationsDroite1 = 0; nbRotationsDroite1 <= 3; ++nbRotationsDroite1) {
           for (int nbDeplacementsDroite1 = 0; nbDeplacementsDroite1 <= nbCasesLargeur;
               ++nbDeplacementsDroite1) {
              for (int nbRotationsDroite2 = 0; nbRotationsDroite2<= 3;
                   ++nbRotationsDroite2) {
                  for (int nbDeplacementsDroite2 = 0; nbDeplacementsDroite2 <=
                       nbCasesLargeur; ++nbDeplacementsDroite2) {

                      copieControleurPlateau = new copieControleurPlateau(nbCasesLargeur,
                          nbCasesHauteur, etatPlateau, tetriminoActuel, tetriminoSuivant);

                      for (int i = 0; i < nbRotationsDroite1; ++i) {
                          copieControleurPlateau.rotationDroite();
                      }
                      copieControleurPlateau.miseAGauche();
                      for (int i = 0; i < nbDeplacementsDroite1; ++i) {
                          copieControleurPlateau.deplaceDroite();
                      }
                      copieControleurPlateau.chuteInstantane();

                      for (int i = 0; i < nbRotationsDroite2; ++i) {
                          copieControleurPlateau.rotationDroite();
                      }
                      copieControleurPlateau.miseAGauche();
                      for (int i = 0; i < nbDeplacementsDroite2; ++i) {
                          copieControleurPlateau.deplaceDroite();
                      }
                      copieControleurPlateau.chuteInstantane();

                      evaluation =
                          evaluationPlateau(copieControleurPlateau.getEtatPlateau());

                      if (evaluation > meilleureEvaluation) {
                          meilleureEvaluation = evaluation;
                          meilleurNbDeplacementsDroite = nbDeplacementsDroite1;
                          meilleurNbRotationsDroite = nbRotationsDroite1;

                     \\...
```

```
                \\... (suite)

                    }
                }
            }
        }
    }

    for (int i = 0; i < meilleurNbRotationsDroite; i++) {
        controleurPlateau.rotationDroite();
    }
    controleurPlateau.miseAGauche();
    for (int i = 0; i < meilleurNbDeplacementsDroite; i++) {
        controleurPlateau.deplaceDroite();
    }
    controleurPlateau.chuteInstantane();
}
```

```
public void joue1CoupAvecAnticipationIndefiniment() {
    while (controleurPlateau.getDebutJeu()) {
        joue1CoupAvecAnticipation();
        controleurPlateau.etapeJeu();
    }
}
```

5 La fonction d'évaluation

Une fois les routines de placements créées, il nous faut une fonction d'évaluation performante pour optimiser au mieux nos deux IA. En effet si la fonction d'évaluation est mauvaise, le code de placement du meilleur coup va retourner un coup médiocre et l'IA va mal jouer. Cette fonction d'évaluation doit retourner un score élevé lorsque l'état du plateau est «bon» . Ce pose alors la question de: Qu'est ce qu'un «bon» plateau ? Il faut alors prendre en compte des paramètres permettant de de quantifier l'état d'un plateau.

5.1 Paramètre 1 : L'Augmentation du Score

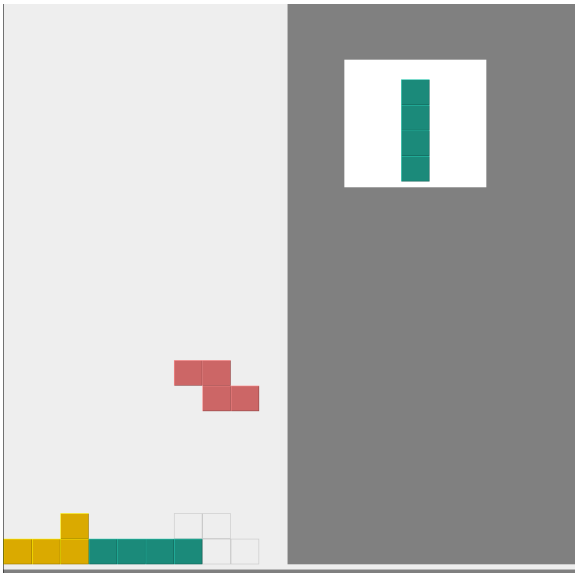
Le paramètre le plus évident qui nous permettait de dire qu'un coup est bien joué, est un paramètre permettant d'augmenter le score. En effet, lorsque le score augmente cela signifie que l'on a réussi à aligner une ligne horizontale complète de Tétrimino. Ceci permet donc de libérer une ligne mais surtout d'augmenter le score ce qui est le but même du jeu. On crée alors une variable

`evaluation_lignesCompletes`, qui compte le nombre de fois que l'on complète une ligne lorsque l'on essaie un coup. Ainsi on va chercher à maximiser le score grâce à cette variable.

```
if (estComplet) {
                ++evaluation_lignesCompletes;
            }
```

5.2 Paramètre 2 : Favoriser l'alignement horizontale

Lorsque que l'on joue à Tétris, il est préférable de placer une pièce de telle sorte qu'elle maximise les alignements horizontaux. En effet, si l'on réalise cela on a plus de chance de compléter une ligne en plaçant une pièce en créant un alignement horizontal car c'est en complétant horizontalement une ligne que l'on supprime la ligne et que l'on augmente le score. De plus il est préférable de placer une pièce de façon à augmenter l'alignement horizontal car si on aligne des Tétriminos de manière verticale, on crée facilement un plateau encombré par des pièces, le haut du plateau est vite rempli et il est plus difficile de compléter une ligne horizontalement. De ce fait on supprime plus difficilement des lignes et donc on se retrouve plus vite encombré par des pièces sur le plateau ce qui accélère grandement la défaite.



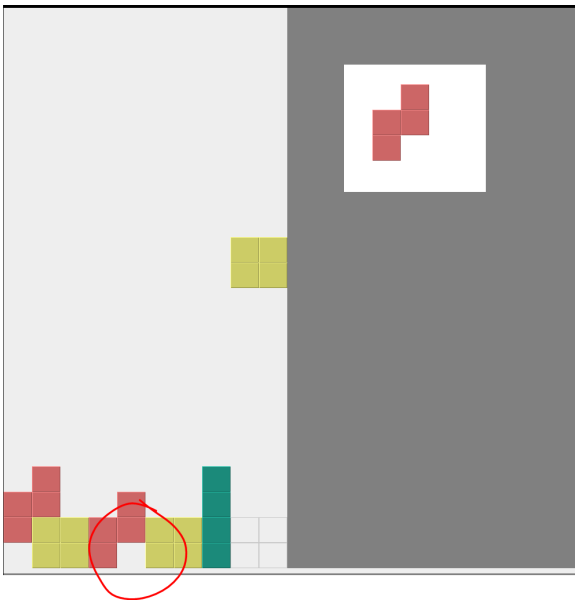
On crée la variable `evaluation_alignementHorizontal` qui augmente lorsque l'on place des Tétriminos sur la même ligne. Cette incrémentation est plus importante lorsque l'on favorise une ligne plus basse. En effet si un coup favorise autant un alignement sur une ligne située en bas du plateau et sur une ligne située plus en hauteur, on préfère placer la pièce sur la ligne en bas du plateau afin d'encombrer le moins possible la partie haute du plateau. Ainsi on augmente d'autant

plus la variable lorsque l'on peut augmenter le nombre de Tétriminos sur une même ligne située dans le niveau inférieur du plateau.

```
for (int k = 0; k < nbCasesLargeur; k++) {
    if (getTypeAt(plateau, k, y) != Tetrimino.TypeTetrimino.Vide) {
        evaluation_alignementHorizontal= evaluation_alignementHorizontal+y;
}
```

5.3 Paramètre 3 : Diminution du nombre de trous

Pour éviter de créer des "trous" dans notre plateau (ce qui empêche de remplir des lignes) on crée une variable `evaluation_NbTrous`. Cette variable va augmenter lorsque l'on crée des trous, c'est-à-dire lorsqu'il y a un espace vide entre le Tétrimino et le bas du plateau.



Cette variable, `evaluation_NbTrous` va donc compter de manière négative dans le score car on cherche à minimiser le nombre de trous créer et donc à minimiser cette variable.

```
 for (int q = y; q < nbCasesHauteur; q++) {
                    if (getTypeAt(plateau, x, q) == Tetrimino.TypeTetrimino.Vide) {
                        evaluation_NbTrous++;
                    }
                }
```

6 Bilan des trois paramètres

Ces trois paramètres font partie intégrante de la fonction d'évaluation. Il ne reste plus qu'à trouver les bons coefficients à attribuer à ces paramètres pour optimiser le score du jeu. Cette fonction d'évaluation s'appelle `evaluationPlateau` et prend en paramètre un plateau de Tétriminos.

```
public int evaluationPlateau(Tetrimino.TypeTetrimino[] plateau) {
       int evaluation;
       int evaluation_NbTrous = 0;
       int evaluation_alignementHorizontal = 0;
       int evaluation_lignesCompletes = 0;

       for (int y = 0; y < nbCasesHauteur; y++) {
           for (int x = 0; x < nbCasesLargeur; x++) {
               if (getTypeAt(plateau, x, y) != Tetrimino.TypeTetrimino.Vide) {

                   // Favorise l'alignement horizontal
                   for (int k = 0; k < nbCasesLargeur; k++) {
                       if (getTypeAt(plateau, k, y) != Tetrimino.TypeTetrimino.Vide) {
                           evaluation_alignementHorizontal++;
                       }
                   }

                   // Compte le nombre de trous
                   for (int q = y; q < nbCasesHauteur; q++) {
                       if (getTypeAt(plateau, x, q) == Tetrimino.TypeTetrimino.Vide) {
                           evaluation_NbTrous++;
                       }
                   }
               }
           }
           boolean estComplet = true;

           for (int j = 0; j < nbCasesLargeur; ++j) {
               if (getTypeAt(plateau, j, y) == Tetrimino.TypeTetrimino.Vide) {
                   estComplet = false;
                   break;
               }
           }
           if (estComplet) {
               ++evaluation_lignesCompletes;
           }

       }
       // Coefficients a optimiser
       evaluation = 3000 * evaluation_lignesCompletes + 1 *
           evaluation_alignementHorizontal - 15 * evaluation_NbTrous;
   }
```

IV Recherche des bons coefficients des paramètres de la fonction d'évaluation

Nous avons pris en compte trois paramètres pour prendre en compte le bon état d'un plateau de jeu. Cependant il est difficile pour nous de savoir quel paramètre privilégier par rapport à un autre.

1 Mise en œuvre théorique

Pour déterminer quelle pondération nous devons donner à notre fonction d'évaluation nous avons réaliser un programme nous donnant le meilleur score moyen en faisant varier les coefficients de la fonction d'évaluation.

Rappelons que dans notre fonction d'évaluation nous avons trois coefficients qui détermine le score de fonction d'évaluation. Il y a tout d'abord le coefficient qui augmente lorsque l'on complète une ligne avec les Tétriminos, nous l'appellerons `A` ou `coeffLigne`. Il y a le coefficient qui cherche à maximiser les alignements horizontaux de Tétriminos, nous appellerons ce coefficient `B` ou `coeffHorizon`. Ainsi que le coefficient qui calcule le nombre de trous crées afin de minimiser leur formation, que l'on nomme `C` ou `coeffTrou`.

Comme il nous faut une pondération des trois coefficients les uns par rapport au autres, on peut fixer un des trois coefficients, ici nous avons fixé le coefficient associé à l'horizontalité (`B`) et fait varier les deux autres ce qui permet de balayer un ensemble de plages de valeurs pour les coefficients de la fonction d'évaluation.

Nous avons donc le coefficient `B` à 1 et nous avons voulu faire varier les deux autres coefficients. Nous avons donc fait varier `A` entre 2000 et 5000 et `C` entre 10 et 35 (après avoir manuellement chercher la plage de valeurs intéressante). Notons que nous avons tout de même chercher autour d'autres valeurs de coefficients pour voir si nous ne pouvons pas trouver d'autre valeurs inattendus de coefficient pouvant donner des scores élevés.

2 Mise en œuvre pratique

Pour faire varier ces deux coefficients, il a fallu mettre en propriété de la classe `controleurIA` ces deux coefficients et créer deux méthodes permettant de faire varier leur valeur à chaque fois que on appelle cette fonction. Ces deux fonctions s'appellent `setCoeffLigne` et `setCoeffTrou`.

```
public void setCoeffLigne( int aa ) {
     coeffLigne=aa;
   }
   public void setCoeffTrou( int cc ) {
     coeffTrou=cc;
   }
```

Ensuite nous avons cherché à créer une fonction qui joue plusieurs fois avec l'IA d'anticipation.

Cette fonction prend en paramètre le nombre de fois que cette fonction joue jusqu'à ce qu'elle perde. Nous avons appelé cette fonction `donneMoyenne(int n)`. Cette fonction joue jusqu'à ce que l'IA perde, récupère le score associé à cette défaite et recommence jusqu'à ce que l'on a fait `n` fois. Elle retourne le score moyen associé au valeur des coefficients `coeffLigne` et `coeffHorizon`.

Notons que la moyenne retournée par cette fonction est d'autant plus pertinente que le nombre n augmente (loi faible des grands nombres). Nous avons choisi une valeur de `n` assez grande pour avoir une valeur de moyenne la plus pertinente possible et une valeur de n pas trop grande pour ne pas exploser la multiplicité de la fonction. Nous nous sommes mis d'accord sur le nombre de 70 car le programme met approximativement 7 min à s'exécuter avec ce nombre d'itération ce qui n'est pas rédhibitoire du point de vue temporel.

```
    public int donneMoyenne(int n) {
      int moyenne =0;
      for (int i = 0; i < n ;i++) {
      this.joue1CoupAvecAnticipationIndefiniment();
         moyenne=moyenne+controleurPlateau.getScore();
         controleurPlateau.start();
      }
      return moyenne/n;

    }
```

Enfin il faut utiliser une fonction qui va faire varier les coefficients `coeffLigne` et `coeffTrou`. Cette fonction va utiliser la fonction précédente (`donneMoyenne(int n)`) pour donner le score moyen associé à un certain couple de `coeffLigne` et `coeffTrou`. Cette fonction s'appelle `donneTableauScore` et renvoie un tableau donnant le score moyen associé au `coeffLigne` et `coeffTrou` que l'on incrémente de pas à pas à chaque fois.

Cette fonction nous est très gourmande en calcul, elle a mis plusieurs heures à s'exécuter.

```
public ArrayList<Integer> donneTableauScore(int iteration) {
   ArrayList<Integer> tab= new ArrayList<>();

   for (int i =2000 ; i<5500;i=i+250) {
      for (int j =10 ; j<40;j=j+5) {
         this.setCoeffLigne(j);
         this.setCoeffTrou(i);
         tab.add(this.donnemoyenne(iteration));
      }
   }
   return tab;
    }
```

Cette fonction nous a renvoyé le tableau dynamique avec les scores moyens associés au différentes valeurs de coefficients.

3 Analyse des résultats

Meilleurs scores réalisés en moyenne

A	B	(-)C	Moyenne
3500	1	20	4478
4750	1	15	4434
3750	1	15	4381
1500	1	20	4369
2000	1	35	4302
500	1	10	4302
4750	1	20	4264
3000	1	25	4256
500	1	15	4254
3250	1	15	4244
4250	1	20	4152
2500	1	25	4100
750	1	20	4078
2500	1	15	4026
3250	1	25	4024

Ce tableau nous a permit de voir quelle valeur de `coeffLigne` et quelle valeur de `coeffTrou` nous donne la meilleur qualité de jeu pour notre IA. Ces valeurs sont pour `coeffLigne` de **3500** et pour `coeffTrou` de **20**.

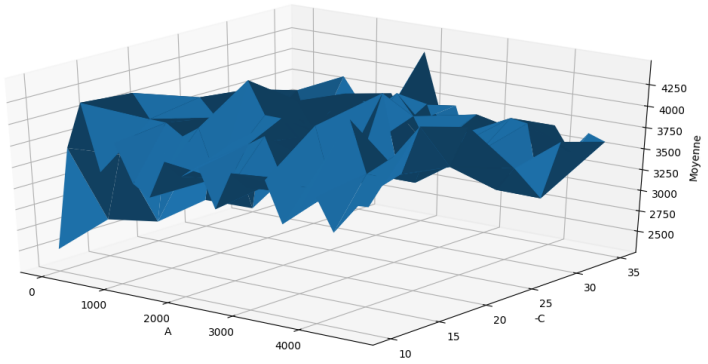
Le score moyen associé à ces deux coefficients 4478 ce qui représente plus de 110 lignes supprimées en moyenne avant de faire Game over.

Le meilleur score obtenu à ce jour par l'IA est 13410.

Remarque : Avec un nombre plus grand d'itérations (300), on remarque que la moyenne obtenue avec 70 itérations est soumis à une variance non négligeable. Le score précédent (4478) et un peu faussé par les erreurs d'incertitudes (coups de chances ?) de la moyenne que globalement le score reste bon pour ces coefficients.

La valeur de ces coefficients nous montre clairement que l'IA joue d'autant mieux que lorsque que elle cherche dès que possible à compléter une ligne. Ce qui explique la forte valeur de ce coefficient. En effet si l'IA cherche dès le début à supprimer une ligne, elle libère de la place pour placer d'autres Tétriminos sur le plateau de jeu et donc va pouvoir jouer plus longtemps et donc faire un plus grand score.

Nous avons tracé en 3D le nuage de point que nous donnait cette fonction.



Ce graphique 3D nous montre qu'il existe des zones où la cote de la surface est basse. Comme par exemple pour des valeurs de `A` faibles ou des valeurs extrêmes. Nous pensons également que l'incertitude sur les moyennes est trop grande. En effet avec 70 itérations pour calculer un score moyen, il est possible que les scores moyens retournés par la fonction ne soient pas représentatifs de la vraie moyenne.

Nous avons également regardé l'efficacité de l'IA qui ne prenait en considération qu'un seul Tétrimino pour calculer le meilleur coup jouable. Avec les mêmes paramètres pour la fonction d'évaluation, on obtient un score moyen de **1150**. Cette IA joue donc beaucoup moins bien que celle qui prend en compte deux coups Tétriminos pour calculer les meilleurs coups possibles.

V Conclusion

Notre IA sait jouer à Tétris, et arrive le plus souvent à nous battre (en terme de nombres de lignes) ! Sans surprise, l'IA avec anticipation est notre meilleure implémentation.

Les deux limites non résolues dans le temps imparti pour le projet sont que certains coups (complexes) ne sont pas envisagés par l'algorithme, et que l'IA ne cherche à remplir plusieurs lignes d'un seul coup (ce qui permettrait d'avoir un meilleur score). Ces deux considérations constituent des pistes immédiates d'amélioration.

Ce projet sur le jeu Tétris a été très enrichissant pour nous. Il nous a permis de nous familiariser encore plus avec le langage Java au travers d'une application concrète, de réfléchir sur des problématiques d'optimisation et d'implémentation dans le cadre ludique d'une IA pour jeu vidéo, mais aussi d'améliorer l'aspect graphique que nous avons donné à notre jeu. Ce projet nous a montré l'importance du travail en équipe mais aussi les joies de celui-ci lorsque l'on parvient à surmonter une difficulté à laquelle on essaye de faire face depuis un moment.

A Bibliographie

[1] Java tetris. http://zetcode.com/tutorials/javagamestutorial/tetris/, 2018. [Online; accessed 12-April-2018].

[2] Wikipedia. Tetris — Wikipedia, the free encyclopedia. https://fr.wikipedia.org/wiki/Tetris, 2018. [Online; accessed 12-April-2018].