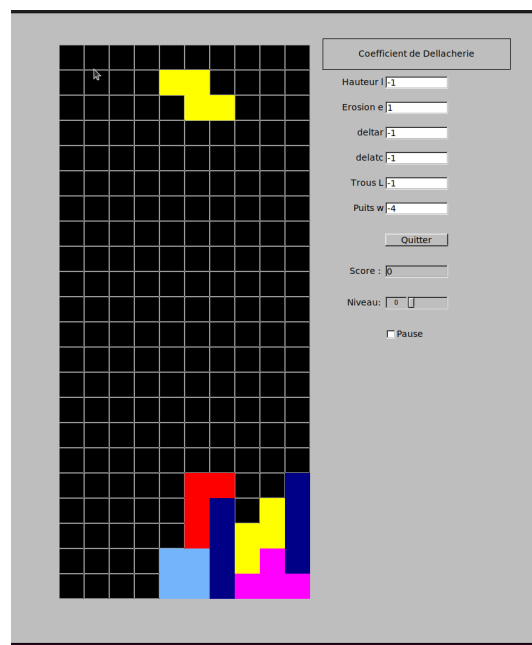




# Projet Tétris

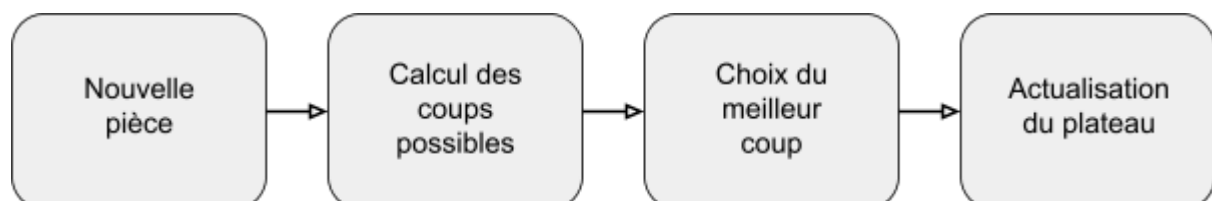
Notre projet d'informatique consiste à créer un jeu Tétris. Nous devons coder le jeu de Tétris conçu par l'ingénieur soviétique Alekseï Pajitnov à partir de juin 1984 sur Elektronika 60. Il fête ainsi son 40<sup>e</sup> anniversaire cette année. Le jeu consiste à positionner le plus de pièces prédéfinies possible sur un plateau de jeu rectangulaire. Les lignes entièrement remplies se suppriment alors afin de libérer de la place et empiler d'autres pièces.

L'objectif premier de ce projet est donc de coder un programme faisant jouer l'ordinateur de la façon la plus optimale possible.



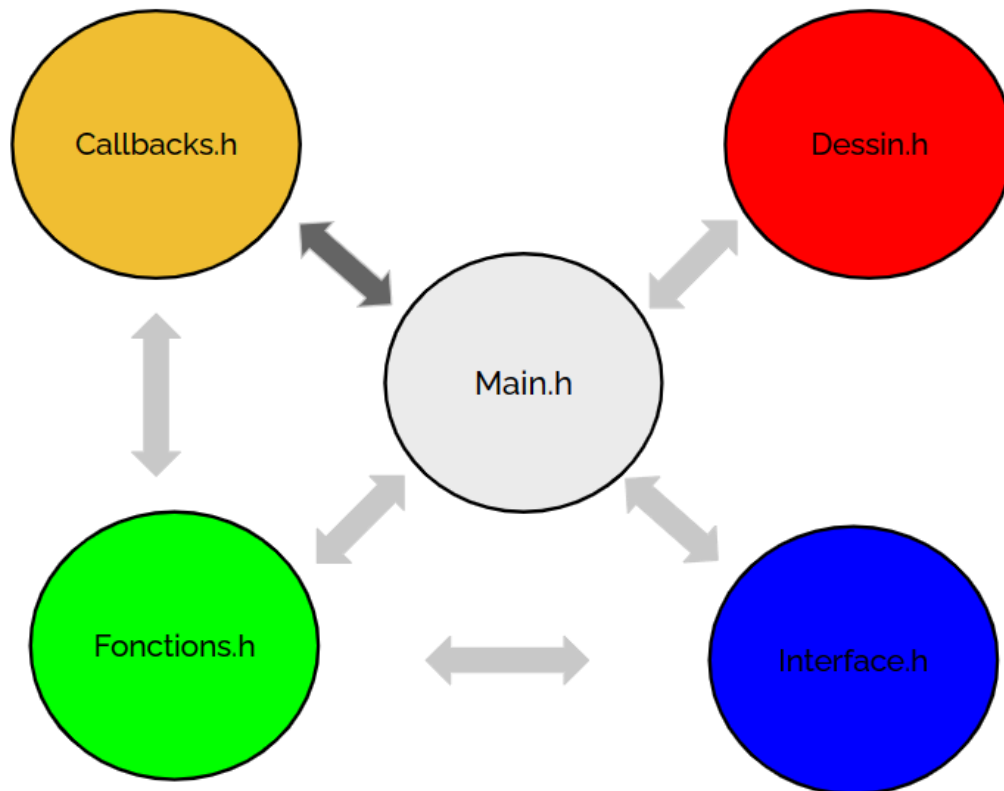
## 1. Fonctionnement général

Le programme que nous allons mettre en place est cyclique, il s'agit de répéter le schéma suivant à l'infini :



C'est la capacité d'un ordinateur à répéter ces instructions très rapidement et avec plus de précision qui devrait le distinguer des résultats humains.

Pour satisfaire cela, notre programme se découpe en plusieurs modules indépendants. Ainsi l'ensemble du code est découpé en plusieurs documents *.cpp* qui ont chacun différents objectifs mais sont reliés grâce aux documents *.h* qui répertorient les différentes fonctions et qui permettent de les utiliser ainsi que nos deux variables globales entre les différents fichiers :



- Les fichiers *main.h* et *main.cpp* sont le cœur du projet, le *.cpp* contenant la fonction *main*. Cependant, cette fonction est déjà fournie : sa fonction principale est d'appeler la fonction *callback*. Ainsi, la fonction *main* n'est pas le cœur logique du projet.
- Les fichiers *callback.cpp* et *callback.h* permettent de faire fonctionner le cycle, en appelant les fonctions des autres fichiers.
- Les fichiers *fonctions.cpp* et *fonction.h* comportent les fonctions d'apparition d'une pièce, d'évaluation du jeu et du choix du meilleur coup.
- Les fichiers *dessin.cpp* et *dessin.h* dessinent à chaque cycle le nouveau plateau avec le meilleur coup choisi.
- Les fichiers *interface.cpp* et *interface.h* concernent l'interface.

Ainsi, cycle après cycle, le plateau de jeu se construit.

## 2. Structure des données fonctionnelles

Afin de réaliser ce jeu, nous avons défini une structure *Donnees* reliée à une variable globale nommée *gDonnees* qui va permettre de stocker l'ensemble des informations

nécessaires à l'évaluation de chaque coup (lister les coups possibles, trouver le coup optimal et actualiser le plateau de jeu). *gDonnees* contient donc l'ensemble du plateau de jeu au début de cycle sur lequel il va falloir se baser pour faire l'évaluation. Le plateau est modélisé par un tableau (*Plateau[NBLIG][NBCOL]*), l'ensemble des pièces est modélisé par une liste de liste contenant une matrice 4x4 de chaque pièce pour chaque rotation possible (avec des 0 si vide et le numéro de la pièce si plein).

L'ensemble des coups possibles est un tableau de pointeurs de taille NBROT\*NBCOL appelé *Coups* dont le  $j + k \cdot \text{NBCOL}$  sera un pointeur vers un type *CoupPossible*. Ce dernier contient la position d'arrivée si la pièce est placée à la  $j$  ième colonne pour sa  $(k+1)$  ième rotation. *gDonnees* contient aussi les pondérations des coefficients de Dellacherie de la position en question. Nous aborderons en profondeur la notion de coefficient de Dellacherie dans la suite (coefficients permettant de comparer l'optimisation des coups).

La structure est alors la suivante :

```
// Structure globale pour les variables fonctionnelles
struct Donnees
{
    // Variables du projet a definir ici
    uint8_t Pieces[NBPIECES][NBROT][SIZE][SIZE]={-};
    enum Couleurs Plateau[NBLIG][NBCOL]; // Plateau de jeu en cours
    struct CoupPossible * Coups[NBCOL * NBROT]; // Tableau de pointeurs sur les coups possibles pour la piece en jeu

    // Variables : numero de piece en jeu, X et Y apparition de la piece ?
    uint8_t num_piece;
    uint8_t X_apparition;
    uint8_t Y_apparition;

    // Variables poids des criteres Dellacherie : poids hauteur arrivee, poids erosion, poids transitions ligne, etc...
    int coef_l;
    int coef_e;
    int coef_r;
    int coef_c;
    int coef_L;
    int coef_W;

    // Autres variables ?
    bool pause;

    int score;
    int Niveau;
    // Variable pour piloter le jeu
    bool Running; // Statut du jeu : Running = true, le jeu est en cours ; Running = false, le jeu n'est pas demarre ou en pause ou termine
};
```

Nous pouvons noter la présence d'autres paramètres non essentiels à la mécanique du joueur artificiel :

- Le score qui sera actualisé à chaque cycle.
- Le niveau qui est relié à une entrée sur l'interface modifiable de 1 à 3 (que nous n'avons pas développé).
- Le booléen Running qui définit si le programme est en fonctionnant.
- Le booléen pause qui prend la valeur True si le jeu est en pause.

Les variables *coef\_l*, *coef\_e*, ... définissent la pondération des coefficients de Dellacherie, modifiable via une entrée sur l'interface. Ils permettent d'ajuster le poids en fonction des différents critères (hauteur d'arrivée, nombre de trous...).

En détail, la variable globale *Coups* contient pour une pièce donnée : la rotation, la position d'arrivée et une copie du plateau de jeu dans laquelle la pièce est posée. Ce sont les paramètres qui définissent comment serait le jeu si nous choisissons cet emplacement. Ceci permet d'en déduire les valeurs des coefficients de Dellacherie et donc le poids résultant associé au coup possible (la variable finale qui nous intéresse). Ils sont mis à jour par les fonctions de fonction.cpp.

La structure est alors la suivante :

```
// Structure pour la liste des coups possibles (tableau de taille NBCOL * NBR0T, tableau de pointeurs de struct CoupPossible, NULL quand le coup n'est pas possible)
struct CoupPossible
{
    // Variables : numero de rotation de piece, X et Y arrivee de la piece
    uint8_t num_rot_piece;
    uint8_t X_arrivee;
    uint8_t Y_arrivee;

    // Variable : Plateau resultat possible. cette pce rentre
    enum Couleurs Plateau[NBLIG][NBCOL];

    // Variables criteres dellacherie : hauteur arrivee, erosion, tansitions de lignes, transitions de colonnes, trous, puits ?
    uint32_t l ;
    uint32_t e ;
    uint32_t r ;
    uint32_t c ;
    uint32_t L ;
    uint32_t W ;

    // Variable pour l'evaluation de ce coup avec les criteres Dellacherie
    uint32_t Poids;
};
```

Le principe de fonctionnement du programme revient à jouer entre ces deux variables globales. Si nous reprenons l'explication du 1) avec les termes structurels, cela revient à effectuer, à chaque itération de la fonction *void TraiterCycleCB()*, la sélection d'une matrice (pièce) aléatoirement parmi celles enregistrées dans *gDonnees.Pieces*. Pour cette pièce, nous allons ensuite enregistrer tous les coups possibles dans la variable globale *Coups*. Une fois tous les coups enregistrés, nous pouvons les évaluer. Les coefficients de Dellacherie sont calculés et mis à jour dans *Coups*. Puis, nous utilisons une recherche de maximum pour trouver le coup optimal et ainsi modifier le plateau de jeu *gDonnees.Plateau* qui va être stocké dans *gDonnees*. Cycle après cycle, nous construisons notre plateau de jeu.

Par ailleurs, la sélection d'une pièce se fait simplement par un tirage aléatoire d'un entier entre 0 et 6 correspondant chacun à une pièce.

### 3. Fonctions/Sous programmes – esquisses algorithmes

Jeu de l'ordinateur :

Dans notre jeu, l'ordinateur que nous avons mis en place prend en compte une pièce que nous lui donnons et va, selon des valeurs de pondération définies par l'utilisateur interface, placer la pièce sur le plateau de jeu.

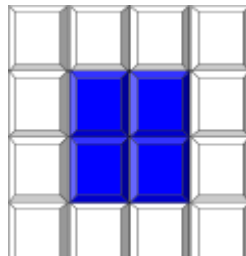
Afin de pouvoir commencer le jeu, nous initialisons d'abord un plateau de jeu vide avec un score de 0. Une fois ceci fait, nous entrons dans notre boucle *void TraiterCycle*, Nous choisissons ainsi une pièce aléatoirement que nous venons placer en haut de l'écran de jeu.

À partir de là, nous devons déterminer le meilleur coup. Nous disposons alors de deux fonctions : *CoupPossible()* et *JouerMeilleurCoup()* qui ne renvoie rien mais qui modifient *gDonnees.Coups* en y ajoutant les différents coups possibles et en remplaçant le tableau de jeu actuel par celui considéré comme le meilleur coup.

### **CoupPossible() :**

À terme, cette fonction remplit la liste continue de pointeurs *gDonnees.Coups[]*. Si le coup *m* est impossible, *gDonnees.Coups[m]* se voit associer le pointeur NULL. Si le coup est possible, nous remplissons la position d'arrivée et le tableau de jeu associé à ce coup. Les 11 premiers pointeurs correspondent aux 11 colonnes pour la rotation 1, les 11 suivantes pour la rotation 2 et ainsi de suite.

Le principal enjeu est de savoir ce que signifie "possible" pour un coup. Nous entendons par là que la pièce ne se superpose pas avec le plateau existant, ni ne dépasse, ni ne flotte dans le vide. Pour simplifier le parcours de la pièce sur le plateau, nous cherchons si des lignes et des colonnes de la pièce sont vides. En effet, le cas du carré illustre bien cette nécessité :



Dans ce cas, la position de la pièce est située sur une ligne et une colonne vides (en haut à gauche), il faut donc démarrer les indices un cran plus loin. De même, la pièce ne fait véritablement que deux colonnes et deux lignes, donc pas besoin d'en parcourir trop (surtout que sortir du plateau est possible si on regarde la 12<sup>e</sup> colonne par exemple). Nous calculons donc les colonnes et lignes vides sur la pièce.

Puis, nous raisonnons colonne par colonne pour chaque rotation *k*. Pour une colonne *j* donnée, nous cherchons la première ligne *i* telle que la pièce rentre. Si nous la trouvons, le coup *gDonnees.Coups[j + k\*NBCOL]* est possible et nous remplissons ses données. S'il ne l'est pas, nous laissons la valeur par défaut NULL.

### **JouerMeilleurCoup() :**

Une fois la liste *CoupPossible* mise à jour, le programme calcule les coefficients de Dellacherie. Ce sont des coefficients qui sont associés à différents critères :

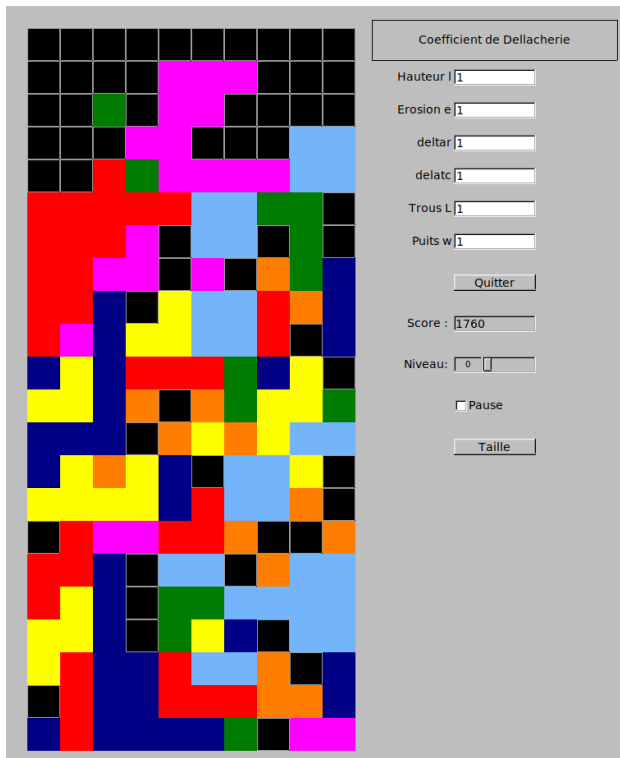
- Hauteur d'arrivée
- Capacité à faire des lignes
- Éviter les trous
- Transitions lignes / colonnes pour rendre le mur homogène
- Le nombre de Puits

Ces différents facteurs prennent en compte l'environnement autour de la pièce pour une position donnée et notre fonction *Evaluer* va venir donner un poids à la pièce avec la pondération défini par l'utilisateur interface.

Ensuite, la fonction *void JouerMeilleurCoup()* utilise une recherche de maximum pour trouver le coup avec un poids optimal.

Vient ensuite le test pour savoir si la pièce posée casse actuellement une ligne ou pas. Dans le cas positif, il faut ainsi abaisser toutes les pièces qui étaient en haut de la ligne supprimée.

Pour constater l'influence des pondérations des coefficients de Dellacherie, nous faisons plusieurs parties avec des valeurs différentes :



Coefficient de Dellacherie

Hauteur l

Erosion e

deltar

delatc

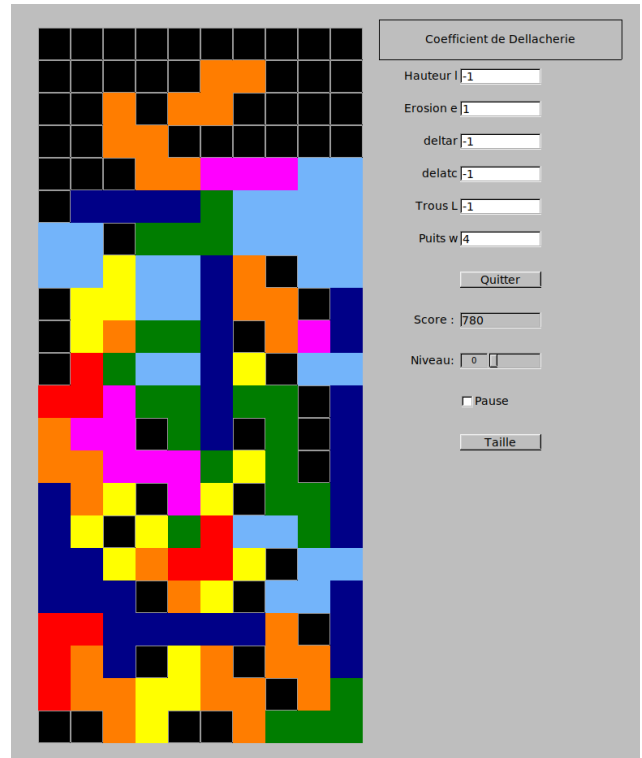
Trous L

Puits w

Score :

Niveau:

☐ Pause



Coefficient de Dellacherie

Hauteur l

Erosion e

deltar

delatc

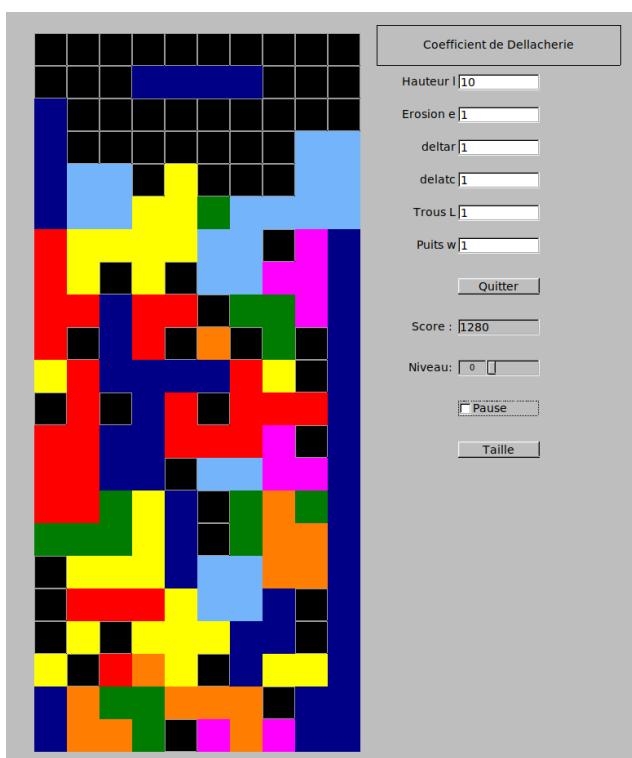
Trous L

Puits w

Score :

Niveau:

☐ Pause



Coefficient de Dellacherie

Hauteur l

Erosion e

deltar

delatc

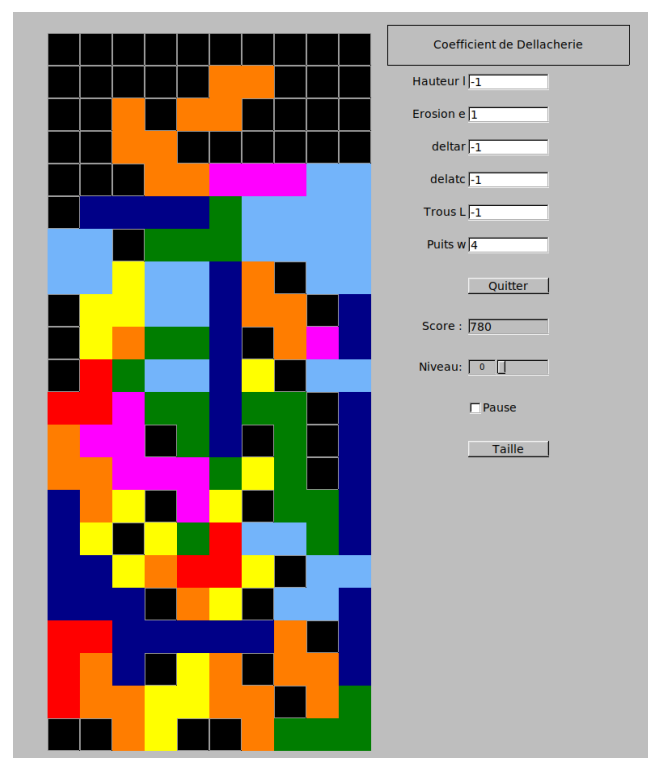
Trous L

Puits w

Score :

Niveau:

☐ Pause



Coefficient de Dellacherie

Hauteur l

Erosion e

deltar

delatc

Trous L

Puits w

Score :

Niveau:

☐ Pause

Les différences sont nettes. Les coefficients de Dellacherie ont donc bien un impact. Il semble que le jeu soit plus efficace lorsque les pondérations sont uniformes. Dellacherie a trouvé des coefficients capables d'effectuer 600 000 lignes.

## 4. Structure des données de l'interface graphique

### - esquisse de l'interface :

L'interface graphique du jeu est codée dans la partie *u1-Interface.cpp* et *u2-Dessin.cpp*.

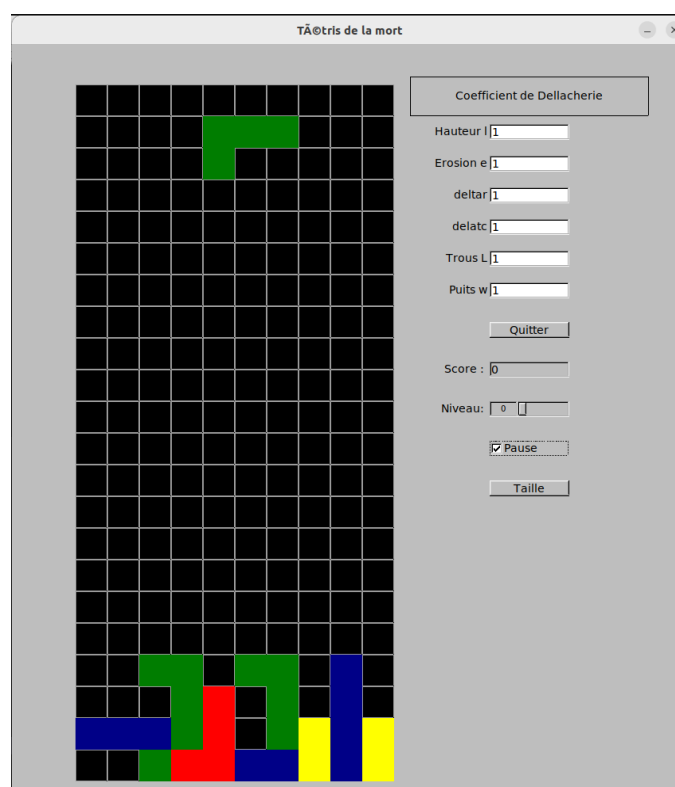
Dans le fichier *u1-Interface.cpp*, la fonction *CreerInterface* permet d'introduire tous les boutons dont le jeu a besoin pour fonctionner : Les emplacements pour mettre les critères de Dellacherie, le bouton Quitter ainsi que le score qui est actuellement affiché.

Tous ces boutons sont associés à des callbacks qui renvoient une information dans notre variable globale *gInterface*. Ces callbacks sont codés dans *u3-Callbacks.cpp*.

Exemple : Bouton Quitter

Nous introduisons le bouton dans l'interface grâce à *Fl\_Button*, avec les dimensions voulues, importé de la bibliothèque *kit-c phelma*. Nous lui associons le callback *BoutonQuitterCB* sans oublier de l'ajouter dans la structure de l'interface. Puis, dans le fichier callback, nous codons la fonction *BoutonQuitterCB* qui ferme le programme.

Dans le fichier *u2-Dessin.cpp* la fonction *ZoneDeDessinCB* permet de dessiner le plateau de jeu et est actualisée en continu par la fonction *TraiterCycleCB*. Lorsque nous exécutons le jeu en utilisant la commande *./projet-type*, une fenêtre apparaît dans laquelle nous créons le plateau dont les dimensions sont *H\_ZONE* x *L\_ZONE* (en pixels).



Nous dessinons d'abord un rectangle noir qui définit notre plateau de jeu. Nous venons ensuite dessiner le plateau de jeu actuel, c'est-à-dire les pièces déjà posées avec les bonnes couleurs. Ensuite, nous venons dessiner un quadrillage pour plus de lisibilité. À chaque traitement de cycle, nous répétons ce processus de façon à afficher une case noire si une ligne venait à être complétée et détruite, ainsi qu'à laisser le quadrillage au-devant de l'interface encore une fois, pour des raisons de lisibilité.

## 5. Détails Techniques spécifiques

Dans l'ensemble, le projet a été mené à bien, avec un cycle fonctionnel, un système de score correspondant à celui du jeu Tétris et une interface satisfaisante. Nous sommes cependant déçus que le Tétris ne soit pas capable de dépasser les 2000 points, qui sont largement atteignables par l'humain. Cela est certainement lié à un problème dont nous n'avons pas réussi à nous défaire : souvent, dans le tableau *gDonnees.Coup*, des tableaux non définis apparaissent sans que nous comprenions pourquoi, et ils sont parfois choisis ce qui fausse le jeu.

Le problème identifié est lié à la recherche de maximum, qui ne retourne pas la bonne valeur. Nous sommes encore en attente du retour de l'enseignant.

## 6. Conclusion

Nous avons presque atteint les objectifs de ce projet, notre programme fonctionne malgré quelques erreurs rémanentes dues au placement des pièces et en termes d'efficacité. De nombreuses pistes d'améliorations sont possibles, le jeu étant complexe, il y a plusieurs facteurs assez compliqués à expliciter tel que l'ancrage des pièces au sein d'une colonne ou tout autre mouvement qui pourrait se faire si la pièce tombait bloc par bloc. Il est possible de rajouter un mode de jeu où l'on viendrait nous-mêmes jouer à la place de l'ordinateur.



# Annexe

## Constantes :

```
#define DUREE_CYCLE 1
// 0.500 sec, dépend du matériel utilise
#define NBLIG 22
// Nombre de lignes du plateau de jeu, version traditionnelle : 22 x 10, 2 lignes du haut pour
// faire apparaitre la piece
#define NBCOL 10
// Nombre de colonnes du plateau de jeu, version traditionnelle : 22 x 10
#define NBPIECES 7
// Nombre de pieces Tetris
#define NBROT 4
// Nombre de rotations de chaque pièce
#define SIZE 4
// Taille d'une pièce

// Une enumeration pour les couleurs de pieces ?
enum Couleurs {VIDE=0, LIGHTBLUE=1, DARKBLUE=2, ORANGE=3, YELLOW=4,
GREEN=5, RED=6, MAGENTA=7};
```

```
-----
// Definition des constantes
#define X_ZONE 80 // X de la zone
#define Y_ZONE 50 // Y de la zone
#define L_ZONE 400 // Largeur de la zone → elle vaut Taille_pixel*NBLIG
#define H_ZONE 880 // Hauteur de la zone → elle vaut Taille_pixel*NBCOL
#define Taille_pixel 40
```

```
// Declaration des objets de l'interface
struct Interface
{
    Fl_Double_Window* Fenetre ;
    DrawingArea* ZoneDessin ;
    Fl_Box* Box;

    Fl_Button* BoutonQuitter;
    Fl_Value_Output* score ;
    Fl_Value_Input* Bouton_l;
    Fl_Value_Input* Bouton_e;
    Fl_Value_Input* Bouton_r;
    Fl_Value_Input* Bouton_c;
    Fl_Value_Input* Bouton_L;
```

```

    FI_Value_Input*    Bouton_W;

    // Si jamais on veut faire un mode joueur:
    FI_Check_Button*   Jouer ;
    FI_Value_Slider*   Niveau ;

};

```

#### Définition des données dans u1-interface.h

Cette partie a pour objectif de définir l'interface utilisateur.  
 Nous créons une interface graphique dans laquelle le joueur pourra intervenir.

```
void CreerInterface();
```

Cette fonction permet un jeu en "temps réel"

```
void ActualiserInterface();
```

Si le jeu est fini, un affichage pour en informer le joueur est créé.

```
void affichage_findujeu();
```

#### Définition des données dans u2-dessin.h

Les fonctions ci-dessous servent à dessiner notre environnement de jeu.

Le plateau de jeu est redessiné en utilisant les deux fonctions ci-dessous :

- void ZoneDessinDessinerCB( FI\_Widget\* widget, void\* data ) ;
- void choisircouleur(enum Couleurs COULEUR);

La première fonction sert à choisir la bonne couleur à dessiner en fonction de celles définies sur le plateau de jeu.

La seconde fonction quant à elle sert à convertir des données uint8\_t en leur couleur correspondante : enum Couleurs conversion(uint8\_t);

#### Définition des données dans u3-callbacks.h

Il s'agit ici de faire le lien entre notre interface graphique, les données ainsi que les fonctions qui en découlent.

Nous traitons d'abord le cycle passé.

```
void TraiterCycleCB() ;
```

```
void ZoneDessinSourisCB( FI_Widget* widget, void* data ) ;    ??
```

```
void ZoneDessinClavierCB( FI_Widget* widget, void* data ) ;  ??
```

Nous prenons en compte les modifications de valeurs / curseurs / booléens qui vont intervenir par l'utilisateur sur l'interface et nous donnons la valeur correspondante dans nos variables définies précédemment (*gDonnees*).

```
void BoutonQuitterCB( FI_Widget* w, void* data ) ;
```

```
void Bouton_ICB( FI_Widget* widget, void* data ) ;
```

```
void Bouton_eCB( FI_Widget* widget, void* data ) ;
```

```
void Bouton_rCB( FI_Widget* widget, void* data ) ;
```

```
void Bouton_cCB( FI_Widget* widget, void* data );  
void Bouton_WCB( FI_Widget* widget, void* data );  
void Bouton_LCB( FI_Widget* widget, void* data );  
void NiveauCB(FI_Widget* w, void* data);
```

#### Définition des données dans u4-fonctions.h

Nous initialisons les données et nous créons la liste complète des coups possibles pour que le plateau passe en paramètre.

```
void InitialiserDonnees();  
void CreerListeCoupsPossibles();
```

Nous calculons les paramètres de Dellacherie et évaluons la qualité du mouvement.

```
void CalculHauteurArrivee(uint8_t m);  
void CalculErosion(uint8_t m) ;  
void CalculTrous(uint8_t m);  
void CalculPuits(uint8_t m);  
void CalculTransCol(uint8_t m);  
void CalculTransLig(uint8_t m);  
void Evaluer();
```

Nous évaluons, pour chaque coup, celui qui est de meilleure qualité et nous jouons ce dernier. Si ce dernier vient compléter une ligne, il faut casser la ligne et faire redescendre toutes les pièces.

```
void JouerMeilleurCoup();  
void cassermur();
```

Nous regardons si le jeu est fini ou non :

```
bool FinDuJeu(enum Couleurs **Plateau);
```

Nous créons le son et gérons le temps entre chaque moment de jeu

```
void JouerSon(char *);  
void Attente(double Seconds);
```