CHIBOUB Isaac  
 MICLO Thibault

# TO52: Tetris C++ / OpenGL



**GI04** : Semestre de Printemps

**Enseignant suiveur** : Fabrice Lauri

# Tables des matières

# Introduction 3

# Règles du Jeu 4

# Choix des types et structures 5

# Diagramme de classe 7

# Fonctionnement 8

# Etapes de Développement 12

# Mouvement et autonomie 14

# Introduction

Contexte  
OpenGL est une puissante librairie graphique permettant le développement d’application 2D et 3D, et, entres autres, de jeux-vidéos.  
Cette librairie est totalement portable et extrêmement évoluée, elle offre de nombreuses ressources pour la synthèse d’image une fois quelques principes assimilés, et à ce propos, la documentation proposée est claire et très fournie.

Le C++ est le langage le plus largement utilisé dans le développement des jeux-vidéos puisqu’étant le langage de prédilection de la plupart des développeurs dans le domaine de l’objet.   
C’est pourquoi nous avons porté notre choix sur ce dernier.

L’application a été développée sous Qt, afin de poursuivre le travail réalisé en IN55 ainsi que pour son framework performant et la praticabilité de ses options.

Travail à effectuer  
 L’objectif de cette TO52 était de créer un clone du célèbre jeu d’arcade Tetris en C++ pour le back-end et la librairie OpenGL pour le visuel.  
  
Le jeu devait être paramétrable (taille du plateau, vitesse de chute des pièces) puisqu’il doit ensuite être intégré à la plateforme IPSEITY dédiée à l’étude des systèmes multi-agents.  
Le cahier des charges imposait également que le jeu soit le plus léger possible en terme de taille c’est pourquoi le choix des types et la structure du diagramme de classe a dû être longuement réfléchie.  
  
Pour les règles du jeu, ce clone est assez proche du Tetris classique, mais les scores, les points, la taille du plateau et la vitesse de chute par niveau sont paramétrables.

# Règles du jeu

Le jeu TETRIS Classique est composé d’un tableau est de taille 40x10 cases, chaque case étant coloriée ou non par une seule et même couleur. C’est la règle que nous avons gardée lors de l’exécution par défaut du programme.

Les tetrominos sont des pièces de géométries et couleurs simples qui tombent pour remplir le tableau.  
Ces pièces sont formées d’agglomérat de case et ont une couleur particulière pour les différencier entre elles.  
Il existe sept types de pièces comportant sept formes différentes :

  
Figure 1 : différents Tetrominos

Ces pièces sont générées aléatoirement et après seulement que la précédente soit tombée au fond du tableau ou bien qu’elle soit rentrée en collision avec une autre.  
Le joueur voit la pièce qui arrivera après qu’il aura déposé celle qui est en train de tomber dans le tableau, il peut ainsi organiser sa stratégie.

Chaque fois qu’une ligne est remplie, c’est-à-dire que toutes ses cases sont remplies par un morceau d’une pièce. La ligne est alors supprimée et le joueur marque des points.  
Le but du jeu est donc de remplier le plus de lignes sans jamais qu’une pièce touche le haut du tableau  ce qui marque la fin de la partie.

Le Score du joueur est mis à jour à chaque ligne effacée : le nombre de lignes effacées simultanément augmente le nombre de points marqués.   
Les niveaux sont passés par un seuil de points et la vitesse de chute des pièces augmente à chaque niveau.

Pendant la chute, la pièce peut être déplacée d’une case sur la droite, sur la gauche, ou effectuer une rotation de 90° selon les limites de la physique, c’est-à-dire que la pièce ne peut être déplacée en dehors des limites du tableau ou à l’intérieur des pièces déjà en place.

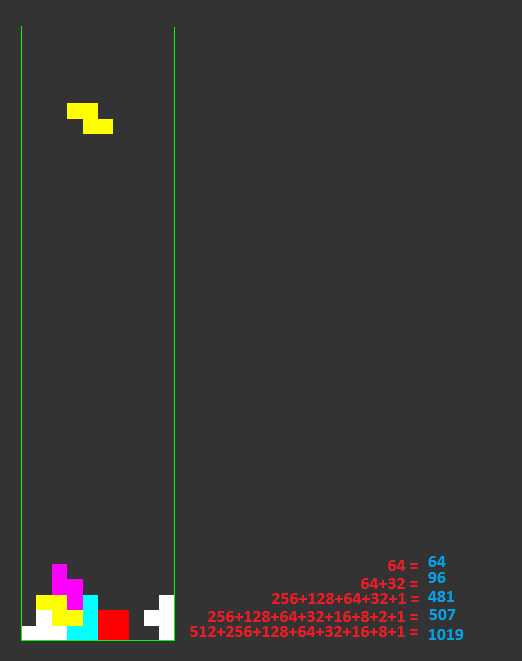
Une fois la pièce déposée elle ne peut plus être déplacée ni subir une rotation, la suivante apparaît au sommet du tableau après un court instant si la première ligne du tableau le permet.  
   
La boucle qui fournit ce comportement compose le jeu en majeure partie.

# Choix des types et structures

Ce jeu ayant été créé dans l’optique d’être intégré à une plateforme de systèmes multi agents, notre cahier des charges requérait des types et structures relativement légers en mémoire et permettant une efficacité de traitement.

La première idée qui vient à l’esprit pour représenter ce tableau de Tetris est évidemment un tableau d’entiers à deux dimensions respectivement de taille 40 et 10 représentant chaque case contenant 0 ou 1 selon son occupation.  
Cette solution bien que simple est très facile d’utilisation présentait un problème majeur dans notre optique d’occupation minimale en mémoire.

La solution que nous avons finalement choisie consiste à supprimer la deuxième dimension du tableau dont nous parlions plus haut par une matière très simple : encoder les 10 booléens de chaque étage du tableau de jeu en mot de 10 bits.

  
Figure 2 : Encodage des étages du tableau

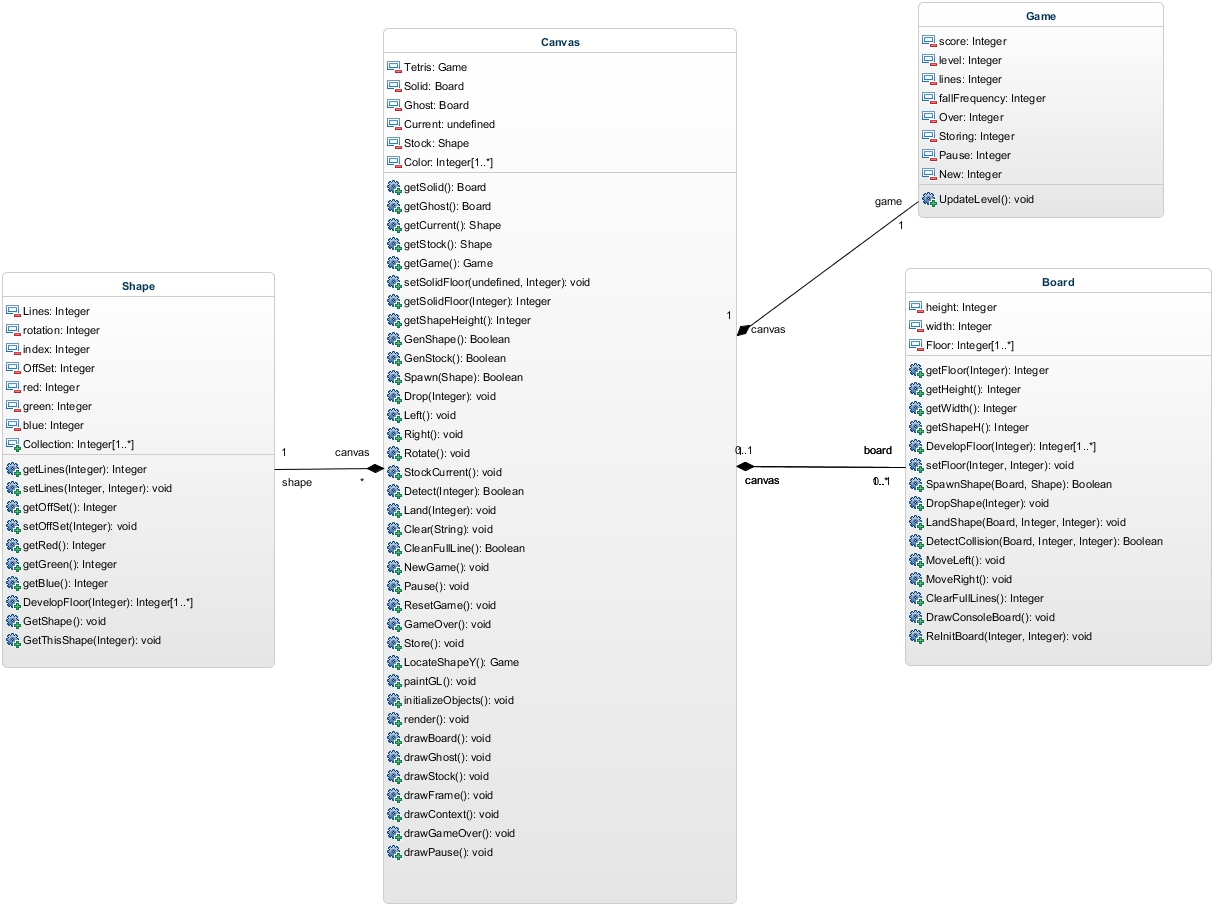
Ce procédé permet de stocker dix booléens en un seul entier en termes de puissances de deux et ainsi supprimer ce besoin de deuxième dimension dans notre tableau.  
Nous pouvons donc instancier la classe **Board** représentant le tableau de jeu avec ce type d’attribut.

Nous avions besoin d’un QGL Widget pour créer la scène et dessiner l’avancement de la partie.  
La classe **Canvas** héritant de cette dernière et de la classe GWindow fournie dans le framework IN55 possède les attributs permettant de faire fonctionner le jeu afin de dessiner chaque action en temps réel.

En effet nous avons fait le choix de faire tourner le jeu en C++ entièrement, actions comprises, OpenGL n’étant utilisé que pour dessiner l’interface visuelle, aucun calcul de matrice n’est ainsi nécessaire.

Un timer permet de redessiner la scène à intervalles réguliers.

# Diagramme de classe

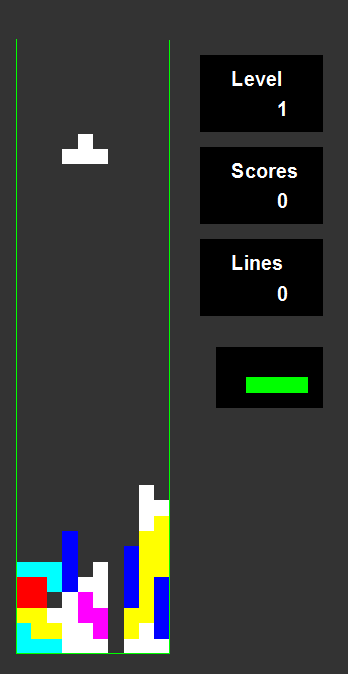


# Fonctionnement

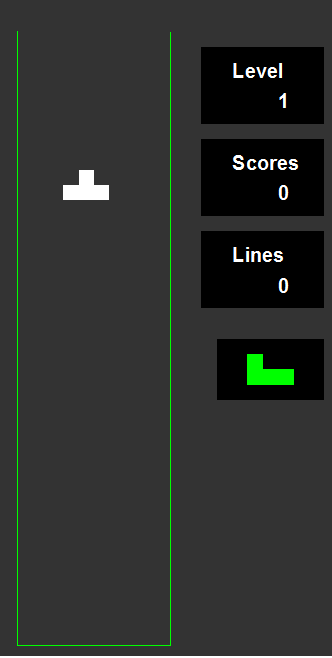
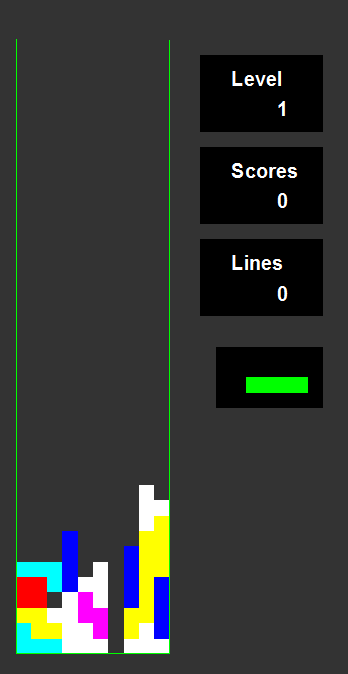
Choix de développement  
Suivant les conseils de notre professeur suiveur nous n’avons opéré aucune opération géométrique en OpenGL nous contentant de coder le jeu entièrement en C++ en back-end et dessiner l’état actuel du jeu dans une fenêtre avec OpenGL.

Back-End  
Nous avons séparé le fonctionnement du jeu entre les deux instances de la classe **Board** que nous avons nommés **Ghost** et **Solid** pour des raisons que nous allons maintenant expliquer.

Etant donné que notre structure est un tableau d’entiers représentant, nous avons réfléchi a un moyen de faire descendre la pièce dans le **Board** sans modifier la position des pièces déjà atterries.

  
Figure 3 : Rendu Espéré

Notre solution fut donc de composer avec deux tableaux l’un, **Solid**, comportant les pièces déjà atterries et gérant les collisions en coopération avec le **Ghost** qui lui ne comporte que la pièce en train de descendre dans le tableau.

   
 Figure 4: Board Ghost Figure 5: Board Solid

L’utilisation d’une seconde board se justifie notamment par le fait de devoir trouver à tout instant la pièce qui est jouée. Au premier abord on pourrait penser qu’il suffit de prendre la partie la plus haute du tableau, cependant, on observe un cas particulier posant problème : lorsque la grille est remplie sur la droite et la gauche, et la pièce descend entre ces 2 colonnes (On peut imaginer une barre tombante dans notre figure 5). A cet instant, l’utilisation d’un champ de bits ne permet pas de distinguer la pièce qui tombe du reste du jeu.

A cet exemple s’ajoute un egestion plus simple des rotations et movements ce qui justifie donc pleinement l’emploi d’une seconde instance de board.

Pour obtenir une pièce on utilise la méthode **getShape**() de la classe **Shape** qui génère un nombre aléatoire entre 0 et 6 lui-même correspondant à une des 7 pièces du jeu.

On invoque cette instance de pièce dans le board Ghost en début de partie ou après un atterrissage avec **SpawnShape**() qui, si l’espace est libre, fait apparaître la pièce dans les deux premières lignes du tableau.

Détection des Collisions  
La méthode **DetectCollision**() renvoie un booléen et donne ou non l’autorisation à la pièce de descendre à la ligne suivante.  
Elle compare pour un nombre de lignes égal à la hauteur de la pièce en chute (hauteur qui dépend de la rotation) si les cases correspondantes dans Solid sont libres ou non. Une fois ce test effectué si aucune case de Solid nécessaire au déplacement n’est occupée, la fonction renvoie true et donne ainsi le feu vert à la pièce pour descendre à la ligne suivante.

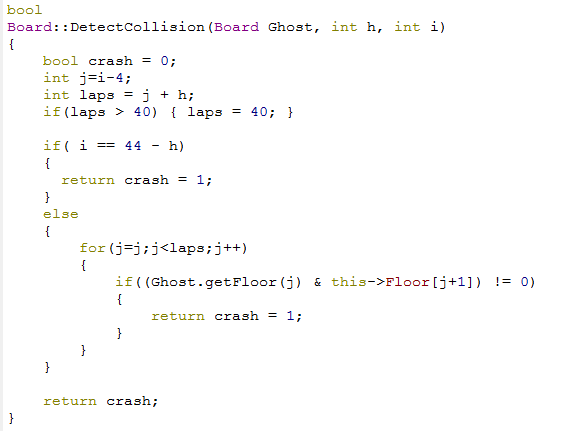
Par exemple, si nous voulons tester si la barre bleue placée horizontalement actuellement en ligne 7 du tableau **Ghost** peut être accueillie en ligne 8 du tableau **Solid** le code agit de cette manière :

Admettons que **Ghost** [7] = 120 c’est-à-dire que la barre est horizontalement placée et son extrémité gauche se situe en case 4 (c’est ainsi qu’elle est générée en haut de tableau par défaut), regardons ce que donne ce nombre en décomposition binaire :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 0 | **0** | **0** | **1** | **1** | **1** | **1** | **0** | **0** | **0** |
| X | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** | **X** |

En deuxième liste du tableau, nous avons **Solid** [8] on se rend compte que si tous les bits de couleur rouges ne sont pas à 0, **Solid** ne peut pas accueillir **Ghost**, la pièce est donc tombée le plus bas possible. Si tous les bits rouges valent 0, la valeur des bits bleus n’a aucune incidence sur la collision.

La méthode **DetectCollision** signale également quand la pièce a atteint le fond du tableau.  
  
Comme montré ci-dessous, cette opération binaire simple est modélisée par l’opérateur & :

  
 Figure 6: Détection des Collisions

Insérée dans la méthode **Detect**() de la classe **Canvas**, elle permet de créer une boucle principale de jeu ( game loop ) de cette manière :

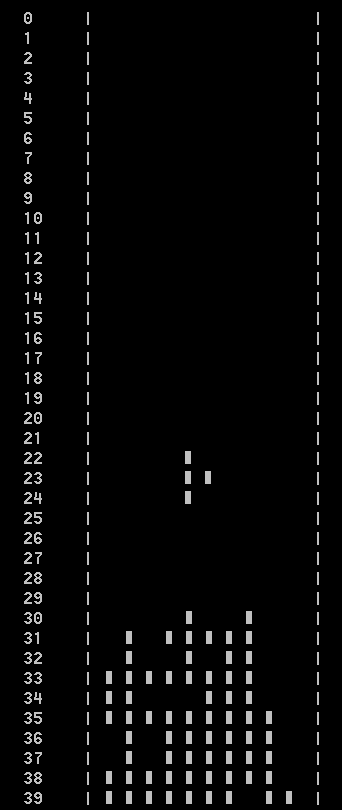


Dans cette boucle nous faisons tomber la pièce quand la détection des collisions le permet, on laisse ici un eventListener pour permettre à l’utilisateur d’appliquer des mouvements à la pièce.  
Dans le cas où la détection renvoie une interdiction de descendre la pièce plus bas, nous rentrons dans l’autre cas : L’atterrissage cette pièce.

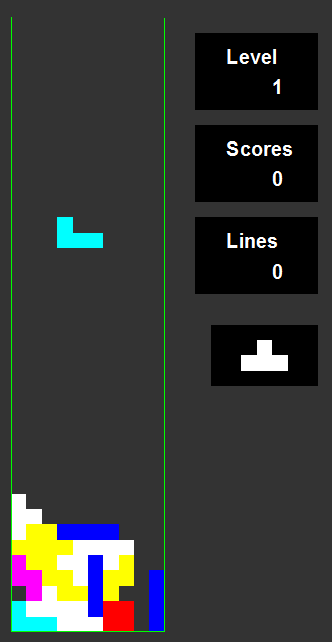
La méthode **Land**() implémente le passage de la pièce du tableau **Ghost** où elle ne peut descendre plus bas et son accueil dans le tableau **Solid** où elle rejoint les pièces précédemment tombées pour créer l’empilement.   
Il ne sert alors plus à rien de continuer cette boucle, une nouvelle pièce est alors générée et ce processus est répété jusqu’à l’appel d’une nouvelle partie par le joueur ou la fin de la partie.

# Etapes de Développement

Visuel  
Avant la création de la partie visuelle en OpenGL, nous avions entièrement réalisé le jeu en C++ avec un rendu console qui nous a permis un débogage en profondeur :

  
Figure 7: Tetris Console  
  
La partie visuelle se contente d’observer le comportement Back-End et redessiner la scène à chaque intervalle de 30 ms provoqué par le déclenchement d’un timer.

Nous disposons de quatre fonctions  principales:

**drawFrame**()  
Cette méthode dessine le cadre du tableau ainsi que les emplacements réservés au score, aux niveaux etc..   
Elle utilise des primitives basiques telles que GL\_LINE\_STRIP et glRect()pour garder une esthétique simple mais un rendu bien marqué.

**drawGhost**()  
Cette méthode dessine la pièce en train de chuter grâce à la primitive glRect() après réception de ses coordonnées actuelles.

**drawBoard**()  
Celle-ci est plus complexe, en effet nous parcourons ici l’ensemble du tableau Solid pour dessiner un carré 1x1 pour chaque booléen ayant la valeur true.  
Le parallèle avec le tableau de couleur nous permet de récupérer les composantes red green blue de chaque pièce et ainsi créer un visuel coloré et fidèle au contenu du tableau.  
  
 **drawContext**()  
Cette méthode se contente d’afficher des chaines de caractères sous forme de dessin dans la fenêtre OpenGL comme le score, le niveau actuel , ou la fenêtre de Game Over .  
Figure 8: Interface Graphique

# Mouvement et Autonomie

Les mouvements de translation à gauche et à droite étant triviaux à part pour la détection des collisions que nous avons détaillé plus tôt dans ce rapport, nous nous concentrerons uniquement sur la phase de rotation de la pièce.

**Rotation**La rotation de chaque pièce est hard codée dans la définition de la classe **Shape**, la fonction de rotation se contente de basculer la pièce d’une position A à une position B qui représentent en fait le départ et l’arrivée après application de la rotation de 90°.

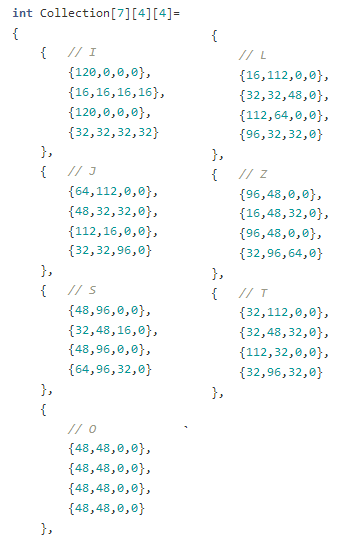


Figure 9: Collection de pièces et leurs positions après chaque rotation

La rotation est une opération compliquée qui demande plusieurs étapes :

1. Vérifier que la rotation est possible, c’est-à-dire qu’aucune collision n’empêchera la pièce de tourner.
2. Stocker le décalage en x de la pièce, c’est-à-dire savoir de combien de cases à gauche ou à droite la pièce se situe avant de subir la rotation puisque les positions hard codées dans la collection ne savent placer la pièce qu’au milieu de la ligne correspondante.
3. Remplacer les coordonnées de la pièce avec celles présentes dans la collection parmi les quatre possibilités en utilisant la rotation suivante, c’est à dire les coordonnées qui suivent celles que la pièce possède actuellement ( la pièce pouvant roter quatre fois avant de reprendre sa position originelle)
4. Utiliser le décalage stocké dans l’étape 3- qui permet de replacer la pièce en son abscisse originelle une fois rotée dans le bon sens.

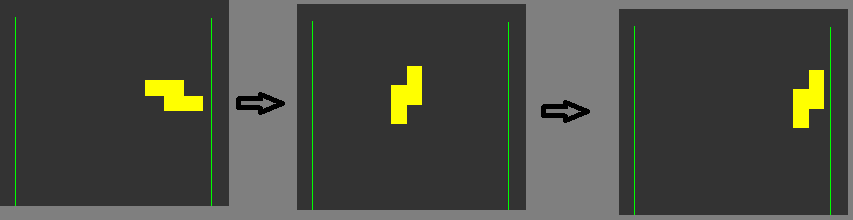


Figure 10: Procédé de Rotation

# Utilisation d’une intelligence artificielle

Notre programme permet l’utilisation d’une intelligence artificielle, toutes les méthodes de jeu sont accessibles à condition de passer en paramètre la board **Ghost** et la board **Solid.** Avec cela, il est possible de connaître la pièce courante ainsi que l’état actuel du jeu.

A titre d’exemple nous avons programmé une intelligence artificielle sensée choisir la colonne la plus vide et y mettre la pièce. Vous constaterez aisément qu’il n’en est pas vraiment le cas en réalité, mais l’essentiel était de montrer qu’il était possible de faire jouer une intelligence artificielle, et cela aisément.

Une fois les calculs et la décision de jeu prise, il convient de renvoyer la valeur du mouvement, pour notre exemple, nous avons 2 choix : « left » et « right ». On peut ajouter les rotations en utilisant un système numérique par exemple :

* 0 : pas de mouvement
* 1 : déplacement à gauche
* 2 : déplacement à droite
* 3, 4, 5 : rotation de 1, 2 ou 3 coups

# Conclusion

Nous tenons tout d’abord à remercier notre tuteur pour cette TO52, Mr Fabrice Lauri pour ses conseils et son écoute lors des réunions.

Cette UV nous a permis de réaliser un clone de Tetris ce qui représente à la fois un challenge technique et un intérêt tout particulier puisqu’il s’agissait du développement de notre premier jeu d’arcade en langage objet.

Le sujet imposait la réalisation de la partie graphique en OpenGL, ce qui nous a freiné dans l’avancée du projet au début, mais nous a permis de progresser en parallèle dans l’UV IN55, même si Tetris ne comporte aucune 3D, la gestion des QGLWidget et l’utilisation du logiciel Qt nous a fait pratiquer et nous a procuré des facilités que nous aurions mis plus de mal à acquérir sans le cadre de cette TO.

Cette première expérience dans les UV hors emploi du temps nous a convaincu avec un sujet intéressant et un mode de travail totalement libre, ce qui nous permet de fournir un résultat qui nous satisfait en termes de contenu et de structures.

Enfin, puisque ce jeu est destiné à être utilisé par une IA et le fait de commencer à coder une interface pour que le programme puisse déplacer les pièces sans joueur humain nous a plu et a ouvert une dimension supplémentaire à ce projet.