École Nationale des Techniques Avancées
Sciences et Technologies de l'Information et de la Communication

Génie Logiciel et Programmation Orientée Objet

TETRIS 3D

Binôme :
Henrique MORAU RODRIGUES
Gabriel PIRES SOBREIRA

Palaiseau - France
6 mars 2021

Table des matières

Table des matières **1**

1 Introduction **2**

1.1 But du Projet 2

1.2 Compréhension du Besoin 2

2 Compilation **2**

3 Description Fonctionnelle **3**

4 Description de l'Architecture **5**

4.1 Les classes 6

4.1.1 Cube 6

4.1.2 Blocks 6

4.1.3 Grid 8

4.1.4 Manager 11

4.1.5 Text 12

4.1.6 Menu 13

4.1.7 Player 14

4.2 La Dynamique du Jeu 15

4.2.1 Mouvement 15

4.2.2 Scènes 16

4.2.3 Clavier 17

5 Futurs Changements et Mises à Jour **17**

Références **18**

1 Introduction

1.1 But du Projet

Le but de ce projet s'agit d'implanter complètement le jeu TETRIS et d'étendre ce jeu à de nouvelle variante, en 3D de type BLOCKOUT avec programmation orientée objet en langage C++.

Pour gérer l'affichage graphique en 3D, il a été utilisé la bibliothèque GLUT (*OpenGL Utility Toolkit*) qui fournit diverses routines pour créer l'interface d'un programme entièrement avec la librairie OpenGL (*Open Graphics Library*).[1][2]. Il a été utilisé aussi pour faire l'affichage des textes dans la fenêtre, et prend les inputs du clavier.

1.2 Compréhension du Besoin

Afin d'atteindre l'objectif proposé, il a été implémenté une hiérarchie dans le programme. Étant donné que pour la réalisation de ce projet, il a été utilisé une classe pour faire la gestion de tous les autres, pour changer les états de jeu. En plus il serait chargé d'appeler chaque scène (menu, le jeu, e la fin du jeu).

En plus, a été fait la mise en œuvre d'une classe pour les pièces du jeu afin de simplifier le code, en raison d'une utilisation fréquente. Ainsi comme une classe pour les cubes qui sont partie composant de chaque pièce, et encore une classe pour faire la gestion du texte dans la fenêtre. Tous ces classes utilisent comme base fonctions de la bibliothèque GLUT, parce qu'elle possibilité une grande simplification sur le code permettant de faire l'affichage sur la fenêtre d'une scène 3D de manière plus simple.

Toujours, en ce qui c'est relation sur qui serait nécessaire dans le projet dans son ensemble, il a été une classe responsable pour la logique du jeu qui est responsable pour gérer le tableau de jeu, la pièce qui tombe et les pièces qui restent sur le tableau, et ainsi s'assurer que toutes les contraintes du jeu sont respectées.

2 Compilation

Pour une bon usage du programme et du joue, il faut utilisé le logiciel *Visual Studio* (la version utilisé a été le "*Community 2019*") pour bien faire la compilation.

Le projet a été construit dans le logiciel en utilisant l'allocation dynamique des librairies, donc la bibliothèque GLUT est dans le dossier du projet, il n'y a donc pas besoin de télécharger de bibliothèque.

3 Description Fonctionnelle

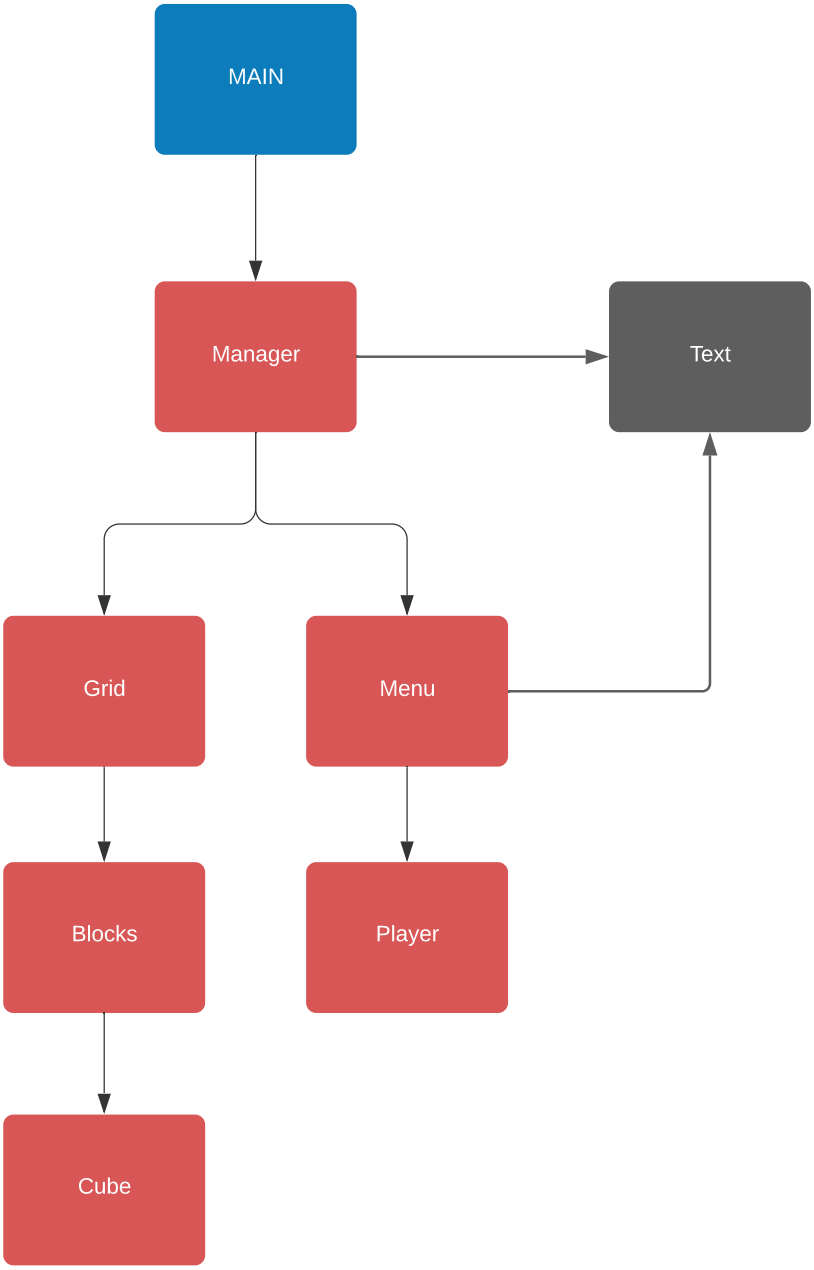

FIGURE 1 – Diagramme de classes.

La partie fondamentale du jeu est la classe **Cube**, c'est celle qui définit la position, et rend chaque cube, en plus de vérifier s'il est dans une position valide.

Après l'arrivée de la classe **Blocks**, il est responsable de la combinaison de plusieurs cubes, d'assembler une pièce, avec que des fonctions ont été créées pour définir, vérifier et rendre les blocs (pièces) d'une manière qui appelle la fonction existante dans la classe **Cube** pour chaque cube qu'ils composez le bloc (pièce).

Vient ensuite la classe **Grid**, elle se compose de différent parties : la première est un vecteur de cubes pour délimiter la grille (grid) et empêcher les pièces de quitter le terrain de jeu, car comme chaque pièce est composée de cubes et d'étui dans la position dans laquelle le bloc se déplacera s'il y a un cube, il est empêché ; la deuxième partie est un vecteur de blocs arrêtés (et donc les cubes de ces blocs), ce vecteur est composé de tous les blocs tombés et non détruits ; une troisième partie est le bloc (pièce) qui tombe. Il y a dans Grid aussi, une partie fondamentale pour le bon fonctionnement du jeu, qui est le mouvement des pièces, bien comme les rotations.

Après cela, il y a la classe **Manager**, elle est composée de 3 parties qui utilisent des différent classes : l'une est la **Grid**, qui a déjà été expliquée, avec tous les mouvementes et la dynamique de le joue ; l'autre est la classe **Text**, qui est chargée de rendre des informations de jeu à l'utilisateur sur les écrans, telles que son score, son niveau et son nom d'utilisateur ; Enfin la classe **Menu**, qui appelle la classe **Player** qui est chargée de stocker les données du joueur telles que le nom, le niveau et le score, et ainsi dans la classe **Menu**, le rendu de l'écran initial du jeu est effectué où l'utilisateur peut entrer son nom pour et commencez, en utilisant également la classe **Text** pour afficher les textes. Enfin, dans le fichier **main**, une instance de manager est créée pour démarrer le jeu.

La sélection de la scène (dans la classe **Manager**) qui est présenté au utilisateur c'est d'accord avec un indice (1 pour le menu, 2 pour le jeu et 3 pour le fin du jeu) l'indice initialise comme 1 pour commencer dans le menu.

Ainsi, l'utilisateur lors du démarrage du jeu entrera dans le menu et donc créer un jouer avec la classe Player où il pourra entrer son nom et démarrer le jeu. Il passera alors de la scène du menu à la scène du jeu (où c'est créé tous les matrices et vecteurs pour sauvegarder des pièces en utilisant le principe de *pushback*, *pop* and *clear*). Dans la scène du jeu, toutes les informations pertinentes pour le joueur seront disponibles. Lorsque le joueur perd, il basculera la scène vers le jeu sur lequel l'utilisateur peut choisir entre redémarrer le jeu ou revenir au menu (et les vecteurs sont vidés pour qu'un autre jeu puisse commencer).

4 Description de l'Architecture

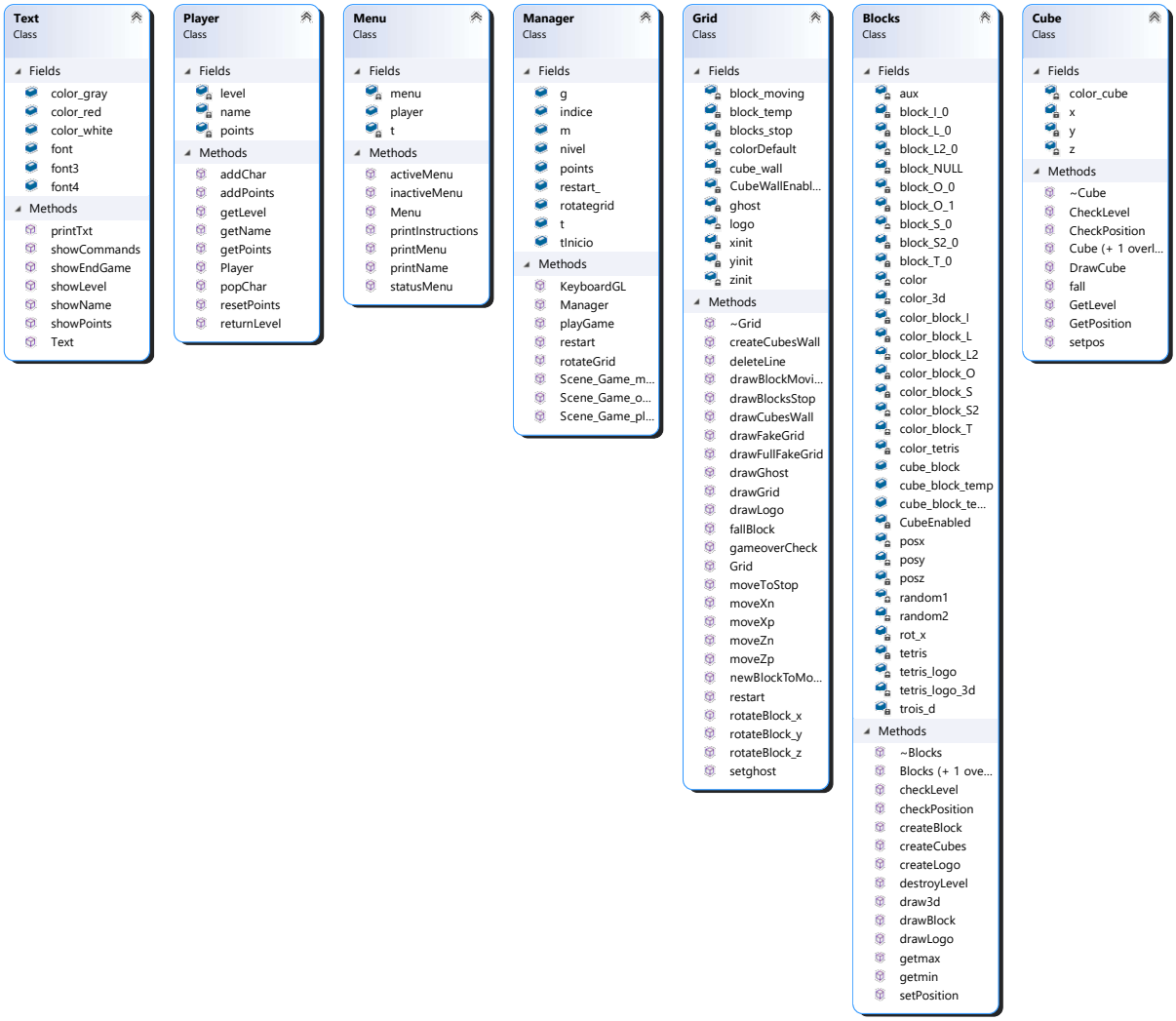

FIGURE 2 – *Fields and Methods.*

4.1 Les classes

4.1.1 Cube

La classe **Cube** a deux constructeurs :

— Cube()
Cet constructeur ne initialise rien, se seulement pour crée une instance de **Cube**

— Cube(float x_, float y_, float z_, GLfloat color[3])
Cet constructeur crée une instance de **Cube** dans la position, x, y, z et une colleur.

La classe **Cube** a plusieurs fonctions :

— void setpos(float x_, float y_, float z_)
Définit la position de l'objet cube.

— void fall()
Fait tomber le cube (par rapport à l'axe y).

— bool CheckPosition(float x_, float y_, float z_)
Vérifie si la position x, y, z passée est la position x, y, z du cube ou non.
Return true si est la position et false se non.

— bool CheckLevel(float _y)
Vérifie si la position y passée est la position y du cube ou non. Return true si est la position et false se non.

— float GetLevel()
Return la position y du cube.

— std : :vector<float> GetPosition()
Return les positions du cube sous la forme d'un vecteur.

— void DrawCube(bool g)
dessine le cube dans les positions et couleurs respectives si g est false, est une cube 'solid', sinon, est une cube 'wired'.

4.1.2 Blocks

La classe **Blocks** a deux constructeurs :

— Blocks()
Cet constructeur ne initialise rien, se seulement pour crée une instance de **Blocks**

— Blocks(float x_, float y_, float z_)
Cet constructeur crée une instance de **Cube** dans la position, x, y, z, et appelle les fonctions createBlock() createCubes().

La classe **Blocks** a plusieurs fonctions :
— bool createBlock()
Crée une pièce ou block (tetromio), en fonction des variables aléatoires random1 et random2 en utilisant une matrice 3x3x3 pour faciliter le positionnement.
Return true si une pièce a été créé.

— void createCubes()
Crée une vecteur de cubes pour la piéce créé. Des cubes sont stockés dans le vecteur cube_block.

— void drawBlock(bool g)
Dessine la pièce, en utilisant la fonction drawCube() (class Cube) sur chaque cube du vecteur de cubes. si g est false, est une block 'solid', sinon, est une block 'wired'.

— bool checkPosition(float x, float y, float z)
Vérifie si la position x, y, z passée est la position x, y, z de chaque cube du vecter qui répresent la piéce en utilisant la fonction CheckPosition() (class Cube) sur chaque cube du vecteur de cubes.
Return true si est la position et false se non.

— bool setPosition(float x, float y, float z, std : :vector<Blocks> blocks, std : :vector<Cube> wall_cube)
Vérifie si la position x, y, z passée est la position x, y, z de chaque cube du vecteur de blocks (qui contient des cubes) qui représent des autres piéces dans la 'grid' et du vecteur de cubes qui représent la 'grid' et donc si c'est possible, modifie des positions de chaque cube et donc de la piéce. En utilisant la fonction CheckPosition() GetPosition() et setpos() (class Cube) sur chaque cube.
Return true si la position de chaque cube ont été modifié et false si non.

— int checkLevel(float _y)
vérifier combien de cubes sont au niveau 'y' indiqué.

Return le nombre de cubes au niveau indiqué.

— void destroyLevel(float _y)
Supprime tous les cubes au niveau y indiqué, en conservant les mêmes positions que ceux en dessous de ce niveau et en descendant (en diminuant la position y) de ceux au-dessus en utilisant la fonction GetLevel() et fall() (class Cube) sur chaque cube.

— std : :vector<float> getmin(std : :vector<Cube> vec)
Recherche la plus petite position de chaque axe (x, y, z) occupé par les cubes d'une pièce.
Return les positions dans un vecteur de taille 3, étant [0] le x, [1] le y et [2] le z.

— std : :vector<float> getmax(std : :vector<Cube> vec)
Recherche la plus grand position de chaque axe (x, y, z) occupé par les cubes d'une pièce.
Return positions dans un vecteur de taille 3, étant [0] le x, [1] le y et [2] le z.

— void createLogo()
Crée les cubes que représent le logo "TETRIS" et "3D" dans le menu.

— void drawLogo()
Dessine les cubes que représent le logo "TETRIS" dans le menu en utilisant la fonction DrawCube(false) dans chaque cube.

— void draw3d()
Dessine les cubes que représente le logo "3D" dans le menu en utilisant la fonction DrawCube(false) dans chaque cube.

4.1.3 Grid

La classe **Grid** a un constructeur :

— Grid()
Crée la 'grid' (CubeWallEnabled[12][21][12]) et donne la valeu 1 pour les cubes qui sont dans les "murs" ou "wall" appelle les fonctions createCubesWall() et newBlockToMove().

La classe **Grid** a plusieurs fonctions :

— void createCubesWall()
Parcoure la matrice de la 'grid' (CubeWallEnabled[12][21][12]) et créer un vecteur de cubes (cubewall) avec des cubes lorsque la valeur de position est 1.

— void newBlockToMove()
Créé une piéce (block) avec le constructeur de Blocks() avec les positions initiales (xinit, yinit, zinit).

— void fallBlock()
Faire la movimentation du block qui tombe à une position en bas. Cas ce n'est pas possible, la pièce s'arrete et une nouvelle pièce doit être en haut du tableu. Appelle la fonction setPosition() de Block et appelle la fonction moveToStop().

— void moveToStop()
Ajoute le block_moving derniere position dans le vecteur blocks_stop e crée une nouvelle block_moving dans la position initiale. Appelle la fonction newBlockToMove().

— void rotateGrid()
Faites pivoter les positions de référence pour que la Grid réel soit dessinée dans une autre position. Chaque somme dans 'rotateGrid' représente 90 degrés supplémentaires, donc en ajoutant 90 quatre fois, ce serait 360, revenant à 0.

— void moveXn()
Si possible, change la position du block en moins un dans l'axis X. Appelle la fonction setPostion() de Blocks.

— void moveXp()
Si possible, change la position du block en plus un dans l'axis X. Appelle la fonction setPostion() de Blocks.

— void moveZn()
Si possible, change la position du block en moins un dans l'axis Z. Appelle la fonction setPostion() de Blocks.

— void moveZp()
Si possible, change la position du block en plus un dans l'axis Z. Appelle la fonction setPostion() de Blocks.

— void rotateBlock_x()
Faire la rotation du block au tour de l'axis X. Appelle les fonction getmin(), getmax() de Blocks et les fonctions GetPosition() et setpos() de Cube.

— void rotateBlock_y()
Faire la rotation du block au tour de l'axis Y. Appelle les fonction getmin(), getmax() de Blocks et les fonctions GetPosition() et setpos() de Cube.

— void rotateBlock_z()
Faire la rotation du block au tour de l'axis Z. Appelle les fonction getmin(), getmax() de Blocks et les fonctions GetPosition() et setpos() de Cube.

— void setghost()
Faire la projection de l'endroit où tombera la píèce. Appelle la fonction setPos() de Blocks.

— int deleteLine()
Verifier si un plan a été complété et si oui, le détruire. Appelle les fonctions checkLevel() et destroyLevel() de Blocks.
Return le nombre de plans détruits.

— bool gameoverCheck()
Verifier si le jeu est fini. Appelle la fonction getmax() de Blocks.
Return True si le jeu est fini , false sinon.

— void drawFakeGrid(float width, float height, int dir)
Cet grid s'appelle Fake, parce que elle est seulement une représentation pour l'user avoir une idée des limites du tableau, mais en fait elle n'est pas responsable de limiter le tableau. Cet fonction va seulement dessinée un carré dans la fake grid avec orientation pour le plan xy, xz ou yz.

— void drawFullFakeGrid(int x, int y, int z)
Cet grid s'appelle Fake, parce que elle est seulement une representation pour l'user avoir une idée des limites du tableau, mais en fait elle n'est pas responsable de limiter le tableau. et cet fonction va seulement dessinée la grille qui est dessiné pour l'utilisateur avoir une notion où sont les limites du tableau. Appelle la fonction drawFakeGrid().

— void drawBlocksStop()
Dessiné tous les bolcks dans le vecteur blocks_stop. Appelle la fonction drawBlock() de Blocks.

— void drawBlockMoving()
Dessiné le block_moving. Appelle la fonction drawBlock() de Blocks.

— void drawGhost()
Dessiné le ghost. Appelle la fonction drawBlock() de Blocks.

— void drawGrid()
Dessiné tous les composants qui son important pour l'utilisateur voir dans la grid. Appelle les fonctions drawBlockMoving(), setghost(), drawGhost() et drawBlocksStop() de Grid.

— void drawCubesWall()
Dessiné la grid qui est vraiment utiliser pour limiter le tableau, cet function est seulement utilisé pour vérifier si tous ce passe bien. Appelle la fonction DrawCube() de Cube.

— void drawLogo()
Dessiné la Logo du jeu. Appelle la fonction drawLogo() et createLogo() de Blocks

— void restart()
Clear le vecteur blocks_stop

4.1.4 Manager

La classe **Manager** a un constructeur :
— Manager()
Crée le 'manager'. il faut faire la gestion du jeu.

La classe **Manager** a plusieurs fonctions :
— void playGame()
Vérifier l'index et rend la scène appropriée appelle les fonctions Scene_Game_play(), Scene_Game_menu() ou Scene_Game_over() de Manager

— void KeyboardGL(unsigned char c, int x, int y)
Vérifier l'index et activer les fonctions du qui sont approprié appelle les fonctions

inactiveMenu() et activeMenu() de menu, les fonctions popChar() et addChar() de Player, les fonctions moveXp(), moveXn(), moveZp(), moveZn(), rotateBlock_x(), rotateBlock_y(), rotateBlock_z() et fallBlock() de Grid et la fonction restart() de Manager.

— void restart()
Reset le jeu pour recomencer. Appelle la fonction resart() de Grid et la fonction resetPoints() de Player.

— Scene_Game_play()
Préparer la scène pour jouer appelle les fonctions returnLevel(), addPoints(), getName(), getLevel(), getPoints() de Player, les fonctions fallBlock(), drawFullFakeGrid(), drawGrid() et deleteLine() de Grid, les fonctions showName(), showLevel(), showPoints(), showCommands() de Text.

— Scene_Game_over()
Préparer la scène pour le fin de jouer appelle les fonctions drawFullFakeGrid() et drawGrid() de Grid, les fonctions showName(), showLevel(), showPoints(), showCommands() de Text, la fonction statusMenu() de Menu.

— Scene_Game_menu()
Préparer la scène du Menu. Appelle la fonction drawLogo() de Grid, les fonctions printMenu() et statusMenu() de Menu.

4.1.5 Text

La classe **Text** a un constructeur :
— Text()
Seulement pour crée une instance de Text.

La classe **Text** a plusieurs fonctions :
— void printTxt(float x, float y, std : :string text, GLfloat color_text[3], void* font)
Variables : x est la position dans l'axis X ; y est la position dans l'axis Y ; text est le texte qui sera afficher dans la fenêtre ; color_text est la couleur de le texte ; font est la font de la letre. Afficher un text dans la fenêtre.

— void showPoints(int points)
Variable : points sont les points du joueur. Afficher les points du joueur dans la

fenêtre. Appelle la fonction printTxt() de Text.

— void showLevel(int level)
Variable : points sont les points du joueur. Afficher les points du joueur dans la fenêtre. Appelle la fonction printTxt() de Text.

— void showName(std : :string name)
Variables : name est le nom du joueur. Afficher le nom du joueur dans la fenêtre. Appelle la fonction printTxt() de Text.

— void showCommands()
Afficher les commandes dans la fenêtre. Appelle la fonction printTxt() de Text.

— void showEndGame()
Afficher le fin de jeu dans la fenêtre. Appelle la fonction printTxt() de Text.

4.1.6 Menu

La classe **Menu** a un constructeur :

— Menu()
Seulement pour crée une instance de Menu.

La classe **Menu** a plusieurs fonctions :

— void printName()
Faire l'afichage du nom du joueur dans la fenêtre. Appelle la fonction printTxt() de Text et la fonction getName() de Player.

— void printInstructions()
Faire l'afichage des commandes dans la fenêtre. Appelle la fonction printTxt() de Text.

— void printMenu()
Faire l'afichage du Menu dans la fenêtre. Appelle les fonctions printName() et printInstructions() de Menu.

— void activeMenu()
Set menu égale à true.

— void inactiveMenu()
Set menu égale à false.

— bool statusMenu()
Return la variable menu (peut être true ou false).

4.1.7 Player

La classe **Player** a un constructeur :

— Player()
Seulement pour crée une instance de Player.

La classe **Player** a plusieurs fonctions :

— int returnLevel()
Vérifier le nombre de points du joueur et retourner le niveau. Return le niveau du joueur.

— void addChar(char c)
Variable : c est un character. Ajoutera le charactére c au nom du joueur.

— void popChar()
Dépouiller le dernier caractère du nom du joueur.

— void addPoints(int num_lines)
Variable : num_lignes est le nombre de lignes qu'ont été déleté. Ajouter un nombre de points en fonction du nombre de lignes détruites.

— void resetPoints()
Reset la pontuation a zero.

— int getLevel()
Return level.

— int getPoints()
Return Points.

— std : :string getName()
Return nom.

4.2 La Dynamique du Jeu

4.2.1 Mouvement

Un point essentiel pour la bonne constitution du jeu, était l'idée qu'il y a deux grilles (*FakeGrid* et *Grid*), l'une d'elles est celle conçue, étant juste pour montrer à l'utilisateur, les limites de l'environnement de jeu. L'autre grille, qui serait la matrice de tout l'environnement ([12] [21] [12]), est la clé de son fonctionnement.

Les murs (*Wall*) de la grille (*Grid*) (x = 0, x = 12, z = 0, z = 12 et y = 0) ont été remplis avec des blocs qui ne sont pas dessinés avec l'environnement de jeu normal, mais lorsqu'ils sont dessinés, ils seraient comme indiqué dans la Figure 3.

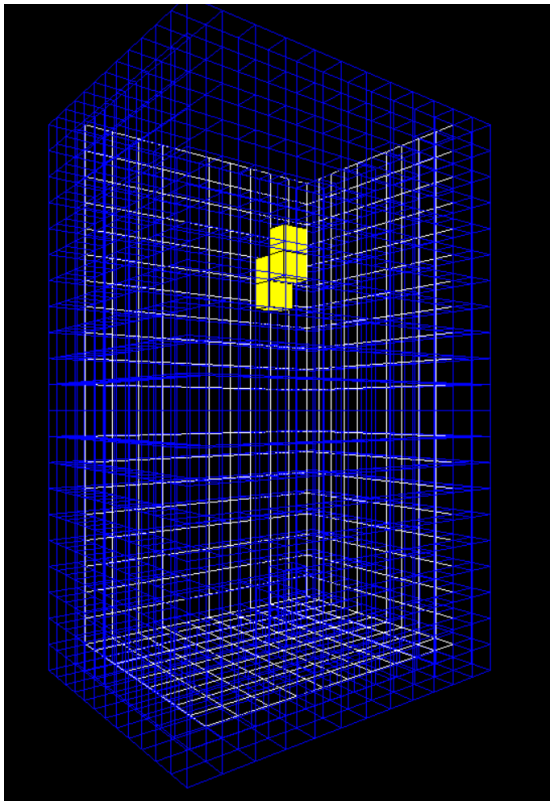

FIGURE 3 – Représentation de la grille non dessiné de *Grid*

Avec l'algorithme présent dans le **Grid**, lorsque on souhaite déplacer les cubes, ou faire des rotations des blocks (pièces), l'algorithme vérifie toujours si la prochaine position souhaitée est remplie par un cube. Ainsi, avec les "murs" constitués de cubes, seule cette vérification suffit, et l'analyse par rapport aux limites de l'environnement n'est pas nécessaire, réduisant ainsi le nombre de fonctions à élaborer. Une idée simplifiée représentant cette interaction des cubes de la pièce avec les autres se trouve dans le pseudocode présent dans l'Algorithme 1.

Algorithme 1 : La dynamique de mouvement des pièces

```
Result : Mouvement : move ou rotate la pièce
for Chaque Cube du vecteur de la pièce do
    for Chaque Cube du "mur" ou des pièces arrêtés do
        if proche position de le cube de la pièce == position de quelque d'autre
         cube then
            pas de mouvement de la pièce;
        else
            faire le mouvement de la pièce;
        end
    end
end
```

4.2.2 Scènes

Il y a trois scènes possibles, étant pour chaque moment du jeu : le menu, le jeu en fait et quand il y a *gameover*.

Pour effectuer cette sélection, il existe un algorithme dans **Manager** chargé de sélectionner la scène à afficher. L'algorithme est représenté par pseudocode 2 et est basé sur la modification de la valeur d'une variable "index" qui représente chacune de ces scènes.

Algorithme 2 : Changement de scène

```
Result : Changer les scènes
switch 'index' dans Manager do
    case 'index' que représente le menu do
        affiche la scène du menu;
        variable booléenne pour le menu = true;
        selon la commande clavier, menu = false et démarre le jeu;
    end
    case 'index' que représente le play do
        affiche la scène du "game play";
        vérifie se il y a le status de "game over" pour changer de 'index';
    end
    case 'index' que représente le "game over" do
        affiche la scène du "game over";
        selon la commande clavier, changez la variable booléen 'menu' ou utilise la
         fonction "restart" pour effacer le vecteur de bloc et redémarrer le jeu;
    end
end
```

4.2.3 Clavier

Pour suivre l'évolution des scènes, un algorithme similaire a été réalisé et le pseudocode 3 le représente.

Algorithme 3 : Clavier

```
Data : Entrées dans le clavier
Result : Changer de clavier avec des scènes
switch 'index' dans Manager do
    case 'index' que représente le menu do
        clavier avec des fonctionnes du menu;
        (default pour la entrée du nom du jouer);
    end
    case 'index' que représente le play do
        clavier avec des fonctionnes du "game play";
    end
    case 'index' que représente le "game over" do
        clavier avec des fonctionnes du "game over";
    end
end
```

5 Futurs Changements et Mises à Jour

En tant que futurs changements ou mises à jour, nous pouvons inclure le développement d'un mode de jeu multijoueur, de sorte qu'en plus de pouvoir jouer avec quelqu'un d'autre, il y a aussi la possibilité de jouer avec un bot. Pour cela, il faudrait ajouter une bibliothèque qui permettrait la gestion d'un serveur puis apporter des modifications aux classes de jeu afin que vous puissiez recevoir cette mise à jour.

En termes de gameplay, des audios pourraient être places au long du jeu et aussi la possibilité de faire pivoter le champ de vision sur la grille de jeu pour donner une meilleure expérience au joueur.

Références

[1] M. Kilgard, feb 1996.
https ://www.opengl.org/resources/libraries/glut/spec3/node1.html
.

[2] J. V. Oosten, feb 2011.
https ://www.3dgep.com/introduction-opengl/#Enter_the_3rd_Dimension
.