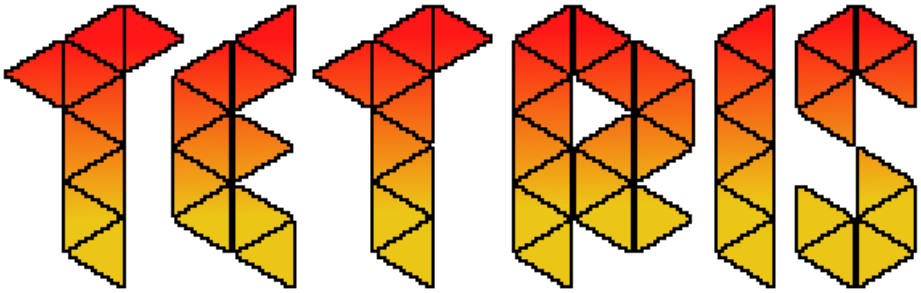
Projet L3-POO2

2013



Paul Gerst

Stan Wilhelm

Répartition

Moteur du jeu + IA : Paul, Stan
Logique : Stan
Animations : Paul

Introduction

Le sujet de ce projet de Programmation Orientée Objet porte sur l'implémentation du jeu Tetris, ainsi que d'une intelligence artificielle (IA). Nous avons choisi de réaliser ce projet en
C++, avec l'utilisation de la bibliothèque SDL2 pour l'affichage. Nous allons commencer par présenter le fonctionnement du jeu, avant d'expliciter l'algorithme de l'IA.

I. Présentation du jeu

L'implémentation du jeu est relativement complexe puisque plusieurs de ses composants ne se limitent pas à la prise en charge du jeu Tetris. En effet, avant de coder le jeu, plusieurs outils ont été mis en place facilitant la réalisation d'un jeu 2D. La plupart de ces éléments prennent appui sur les conseils donnés dans le livre « *Game Coding Complete* » de Mike McShaffry et David Graham. Nous allons donc commencer par décrire ces outils, avant d'en donner l'utilisation pour ce projet.

1. Description des outils

Les outils créés pour ce projet prennent la forme de plusieurs classes, dont chacune des fonctionnalités va nous être donnée :

SDLApp : Cette première classe définit le point central du jeu. Elle ne peut être instanciée qu'une fois (pattern de singleton) au travers d'un pointeur global. Cette classe commence par charger les configurations depuis un fichier XML, puis initialise SDL2 et les principaux composants du jeu. Elle s'occupe ensuite de mettre à jour ces composants et de leur transmettre les inputs. Il est aussi à noter que le jeu tourne à un rythme maximal de 60 frames par seconde, afin de ne pas encombrer inutilement le processeur.
(Fichiers sources contenus dans Src/GameApp)

GameState : Les différents états du jeu sont définis dans ce qu'on appelle des GameStates. La classe permettant d'assurer les transitions entre ses états est le GameStateManager et s'occupe aussi de mettre à jour l'état courant. La plupart de la logique du jeu Tetris est donc contenue dans les classes héritées de BaseGameState.
(Fichiers sources contenus dans Src/GameState)

Resource : Le chargement de ressources telles que des images se fait grâce au ResourceManager. Cette classe permet de garder les ressources chargées en RAM afin d'éviter de nombreux chargements inutiles. Cet outil permet présentement de charger des images ou des polices de caractères. Cependant, afin que les ressources n'occupent pas trop de place, il est possible de rafraichir la liste des ressources chargées (par exemple entre deux GameState).
(Fichiers sources contenus dans Src/Resource)

Graphics : L'affichage des éléments graphiques se fait grâce au GfxManager. Cette classe contient une liste d'éléments graphiques qu'elle affichera selon leur valeur « layer ». En effet, cette valeur définit à quelle « couche » l'élément graphique appartient, laissant ainsi la possibilité de gérer facilement la façon dont les éléments doivent se chevaucher. Les différentes classes d'éléments permettent l'affichage de texte, d'image et de rectangle tout en gérant l'allocation de ressources. De nombreux effets ont été ajoutés pour permettre des animations telles que le changement de couleur, la transparence, l'étirement, la rotation, etc.
(Fichiers sources contenus dans Src/Graphics)

Event : La communication à travers tous ces outils est rendue plus facile grâce à la classe EventManager. Pour fonctionner, cette classe utilise la bibliothèque « Fast Delegate » écrite par Don Clugston et permet l'utilisation de delegates. Cette classe permet concrètement de déclencher des évenements (classes dérivées de IEvent). N'importe quel objet aura la possibilité d'« inscrire » une de ses méthodes à un type d'évenement. Dès lors qu'un évenement de ce type sera déclenché, toutes les méthodes inscrites seront appelées.

(Fichiers sources contenus dans Src/Event)

Process : Le concept de Process est au centre de la logique du jeu et est geré par la classe ProcessManager. Un Process effectuera une action au cours du temps (grâce au Manager qui les mettera constamment à jour) et pourra éventuellement terminer. Un exemple simple de Process est le DelayProcess dont le but est uniquement de rester actif pendant un certain temps (précisé en paramètres) avant de quitter. Il sera aussi possible de chaîner les Process. En effet, il est possible d'attacher un Process « fils » à un autre Process « père ». Lorsque le Process « père » terminera, le Process « fils » sera lancé.

Un exemple simple d'utilisation de Process :
FadeInProcess → DelayProcess(3000) → FadeOutProcess

Dans cet exemple, le FadeInProcess (correspond à un fondu) sera déclenché, puis une temporisation de 3 secondes s'effectuera, avant de lancer le FadeOutProcess.

(Fichiers sources contenus dans Src/Process)

Network : Ce dernier outil est en lien direct avec l'IA, puisqu'il permet d'établir une communication entre deux applications via l'utilisation de sockets et du protocole TCP. Une première classe Server accèpte les connexions extérieures tandis qu'une classe Client permet de se connecter. Les deux classes peuvent s'échanger des évenements qui seront ensuite déclenchés au moyen de la classe EventManager.

(Fichiers sources contenus dans Src/Network)

2. Description de l'implémentation du Tetris

Tout d'abord, quelques classes ont été créées afin de contenir les états des composants principaux du jeu Tetris (comme les pièces, la grille) et sont contenues dans le dossier Src/TetrisLogic. Ces classes sont ensuite utilisées dans des Process, appelés par le GameState principal du jeu : MainGameState.

Le jeu repose sur une principe de boucle. Une boucle de jeu est définie par l'arrivée d'une pièce en haut de la grille et se termine alors que la pièce est posée sur la grille et que les potentielles lignes pleines sont effacées. Une boucle est donc consitutée de Process qui sont chainés. Voici un schéma expliquant son fonctionnement :

FallingPieceProcess se charge de faire tomber graduellement la pièce. Ce Process répond aux inputs, permettant ainsi de faire bouger et tourner la pièce lors de sa chute. Un appui sur la touche Espace **(1)** lancera cependant **DroppedPieceProcess** qui fera tomber la pièce bien plus rapidement. Dans les deux cas, dès que la pièce touche la grille **(2)**, le Process suivant est lancé.

DeleteLinesProcess va vérifier si la grille contient des lignes pleines qu'il est nécessaire d'effacer. Si ce n'est pas le cas **la boucle se termine**. Dans le cas contraire, une animation est lancée pour détruire les blocs de la ligne pleine. Une fois l'animation terminée **(3)**, **FallingLinesProcess** se chargera de faire descendre toutes les lignes qu'il faut réajuster à cause de la disparition des lignes précédentes. Ceci fait, **la boucle se termine**.

De cette manière, à chaque fois qu'une boucle de jeu se termine, MainGameState vérifie que le jeu n'est pas terminé et relance une nouvelle boucle. Le score est mis à jour au moyen d'un évenement qui est déclenché lorsque des lignes sont effacées. Il est aussi à noter que la prochaine pièce à apparaître est affichée pour le joueur, mais l'information n'est pas disponible pour l'IA.

II. Présentation de l'IA

L'IA ne présentant aucun mécanisme d'affichage, seuls les outils concernant les évenements et la communication en réseau sont utilisés pour ce programme. Son fonctionnement est simple : dès qu'une nouvelle boucle de jeu est propagée sur le réseau, l'IA simule chaque coup possible en attribuant un score à l'état de la grille résultant. La meilleure configuration est ensuite sauvegardée et le coup à jouer est envoyé au jeu.

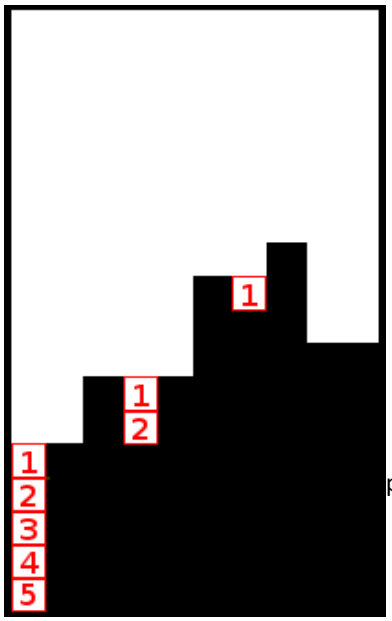
Cette IA repose donc sur sa fonction d'évaluation de la grille. Elle se découpe en plusieurs critères :

Hauteur de la pièce : La hauteur moyenne de chaque bloc de la pièce est calculée. Ce test a pour but de motiver l'IA à placer une pièce le plus bas possible.

Nombre de trous : Un trou est défini par un emplacement vide de la grille où il est impossible de poser directement un bloc. Plus précisément, cela signifie qu'il existe au moins un bloc occupé au dessus de lui. Ce test a pour but de limiter la présence de trous, puisque ces trous représentent une perte d'espace et empêchent l'élimination de lignes.

Nombre de lignes : Ce test compte le nombre de lignes que ce coup permettrait de détruire. Ce test est clairement là pour que l'IA essaye de remplir des lignes.

Nombre de tubes : Un tube (ou « pipe » comme défini dans les sources) définit une suite consécutive d'emplacements vides dont la largeur est d'un bloc. Ce test a pour but de limiter le nombre de tubes puisqu'ils ne sont atteignables que par un nombre restreint de pièces. Il aura aussi été ajouté que la valeur d'un tube augmente en fonction de sa profondeur. Voici un schéma explicitant ce calcul :



Dans cette configuration, le nombre d'emplacements pleins est indiqué en noir.

Le calcul du nombre de tubes pour cette grille serait donc :

nbTubes = (1+2+3+4+5)
+ (1+2)
+ 1

= 19

Cette façon de calculer permet de limiter les tubes trop profonds, tout en tolérant de manière relative le reste du dénivelé.

Au final, chaque test est multiplié par un coefficient et leur somme nous donne l'évaluation de la grille. Ces coefficients jouent un rôle très important puisqu'ils influencent la priorité d'un test par rapport à un autre. Les coefficients que nous avons choisi sont les suivants :

Hauteur : 1
Nombre de trous : -2.5
Nombre de lignes : 1
Nombre de tubes : -1

Ces coefficients ont montré de bons résultats puisque seul le test comptant les trous a une priorité par rapport aux autres. Le test du nombre de tubes prend naturellement de l'importance lorsque des trous trop profonds se creusent. Cette IA a éliminé plus de 15000 lignes avant que nous ne l'arrêtions nous-même.

Conclusion

Ce projet nous aura permis de gagner en confiance avec le langage C++ et la Programmation Orientée Objet. Nous aurons de plus appris à créer des outils qui seront réutilisables lors de développements futurs.

La partie concernant l'IA nous aura obligé à nous intéresser de plus près à un mécanisme de communication entre applications indépendantes. Elle nous aura aussi montré la difficulté de se confronter à un problème dont la solution n'est pas déterministe.

Annexe

1. Pseudo-code

- Méthode CalculateAI :

```
Pièce : type pièce
Score : type float
ScoreActuel : type float
MeilleurRotation : type int
MeilleurPosition : type int

pour chaque rotation
faire
        pour chaque position
        faire
                faire tomber Pièce
                Score = CalculateScore(Pièce)
                si Score > ScoreActuel
                        ScoreActuel = Score
                        MeilleurRotation = Pièce.rotation
                        MeilleurPosition = Pièce.position
                fin si
        fin pour
fin pour

Pièce.rotation = MeilleurRotation
Pièce.position = MeilleurPosition
faire tomber Pièce
```

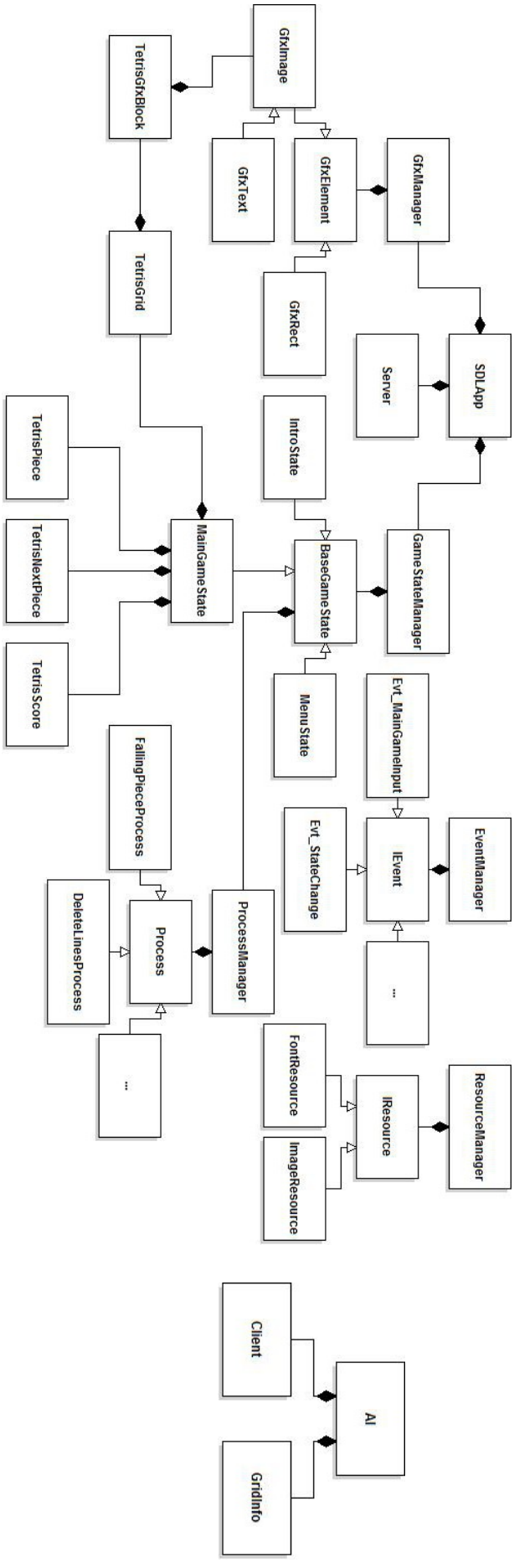
- Méthode CalculateScore :

```
Score : type float
cHeight/cHoles/cRows/cPipes : type float // coefficients

Score = cHeight*AverageHeightTest(Pièce)
          + cHoles*HolesNbTest(Pièce)
        + cRows*RowsNbTest()
        + cPipes*PipesNbTest()

return Score
```

2. UML


GfxImage
TetrisGfxBlock
GfxText
GfxElement
GfxManager
TetrisGrid
GfxRect
Server
SDLApp
TetrisPiece
IntroState
TetrisNextPiece
MainGameState
BaseGameState
GameStateManager
TetrisScore
MenuState
Evt_MainGameInput
FallingPieceProcess
Evt_StateChange
IEvent
EventManager
DeleteLinesProcess
Process
ProcessManager
...
FontResource
IResource
ResourceManager
...
ImageResource
Client
AI
GridInfo