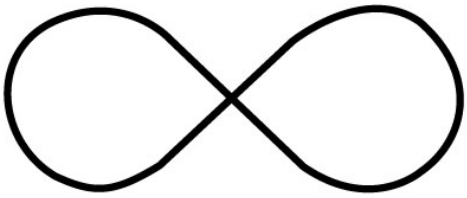
Université Paris Dauphine

MASTER INFO MIAGE
- PARIS -

Master MIAGE : **P**rogrammation **J**ava **A**vancée

Projet

Infinity Loops Game



INFINITY

réalisé par | OUAZZANI CHAHDI NIZAR
M'CHICH MOHAMMED

Table des matières

1 Introduction

Le projet Java consistait à développer un jeu de puzzle permettant de créer des motifs de boucles complexes. le jeu Infinity Loops est considéré comme étant un jeu simple et surtout INFINIE. Nous détaillons dans ce rapport les fonctionnalités développées, l'architecture du jeu ainsi que le diagramme de classe associé au projet.

L'application, comme le stipule le cahier de charge, doit comporter un générateur de niveaux,un vérificateur qui vérifie la résolution ou la non-résolution d'une grille donnée et un solveur.

2 Architecture de l'application

2.1 Architecture

Afin de réaliser notre application , nous avons fait le choix de développer l'application en utilisant l'architecture MVC.Cette architecture permet d'avoir une certaine clarté de l'application et une facilité de manipulation (une mise a jour est souvent facile a faire).

Le jeu faisant sujet du présent rapport se compose principalement de deux types d'objets :

— Le premier représente les pièces telles que décrites sur le sujet du projet

— le deuxième représentant la grille qui contient ces pièces pour former un niveau du jeu

De ce fait, l'architecture que nous avons retenue pour représenter les fonctionnalités du jeu s'articulait autour de deux classes : La classe *Pièce* qui implémente les fonctionnalités attendues de l'objet Pièce (notamment sa rotation) et la classe *Grid* qui implémente les fonctionnalités d'une grille de niveau (en l'occurrence, la génération d'un niveau, la vérification du niveau et la recherche d'une solution du niveau).

En plus de ces deux classes, nous avons choisie d'en créer d'autres afin de pouvoir faciliter la maintenabilité et l'évolutivité du code en modulant et découplant les différentes composantes du jeu. Ainsi, nous avons créé une interface implémentant le contrat pour créer une pièce afin d'assurer l'abstraction du code (l'utilisation des fonctionnalités de la pièce lors du développement de la grille est indépendante du choix d'implémentation de ces fonctionnalités dans la classe Piece).

D'autre part, une classe de lecture et écriture ReaderWriter afin d'assurer la gestion des flux entrées/sorties dans le format préconisé dans le sujet du projet a été implémentée ainsi qu'une classe Solver (et d'une classes interne Node) qui se chargera de la résolution d'un niveau. Par ailleurs, d'autres packages ont été conçus afin de se charger de la création de l'interface graphique et de la réalisation des tests sur les fonctionnalités implémentées

2.2 Construction des pièces

La manière de construire une pièce était soumise à deux contraintes : d'une part les contraintes de génération d'une grille (qui concerne plus particulièrement les connexions que peut avoir une pièce selon ses voisins) et d'autre part les contraintes de lecture et écriture des fichiers dans le format demandé (qui prenait en compte les paramètres de type et d'orientation d'une pièce donnée). Cela s'est traduit par l'implémentation de deux constructeurs. Le premier créait la pièce en fonction des contraintes auxquelles elle est soumise et prenait en paramètre un code traduisant ses contraintes (il aurait été possible de créer un constructeur qui prend directement la liste des contraintes mais cela compliquerait plus son implémentation). Nous fournissons ci-dessous la table de conversion des contraintes vers une pièce.

Conn_Nord	Conn_sud	Conn_ouest	Conn_west	Code corresp	Pièce corresp
false	false	false	false	0*1+0*2+0*4+0*8=0	(0,0)
true	false	false	false	1*1+0*2+0*4+0*8=1	(1,0)
false	true	false	false	0*1+1*2+0*4+0*8=2	(2,0)
false	false	true	false	0*1+0*2+1*4+0*8=4	(4,0)
false	false	false	true	0*1+0*2+0*4+1*8=8	(8,0)
true	false	true	false	1*1+0*2+1*4+0*8=5	(2,0)
false	true	false	true	0*1+1*2+0*4+1*8=10	(2,1)
true	true	false	true	1*1+1*2+0*4+1*8=11	(3,0)
true	true	true	false	1*1+1*2+1*4+0*8=7	(3,1)
false	true	true	true	0*1+1*2+1*4+1*8=14	(3,2)
true	false	true	true	1*1+0*2+1*4+1*8=13	(3,3)
true	true	true	true	1*1+1*2+1*4+1*8=15	(4,0)
true	true	false	false	1*1+1*2+0*4+0*8=3	(5,0)
false	true	true	false	0*1+1*2+1*4+0*8=6	(5,1)
false	false	true	true	0*1+0*2+1*4+1*8=12	(5,2)
true	false	false	true	1*1+0*2+0*4+1*8=9	(5,3)

Le deuxième constructeur prenait en paramètre le type et l'orientation de la pièce afin de la construire. Ce constructeur est particulièrement utile lors de la création d'une grille à partir d'un fichier contenant une grille dans le format demandé par le sujet du projet.

2.3 Construction de la grille

Une grille se définissait par trois paramètres : sa hauteur, sa largeur et ses pièces. Nous avons choisi de représenter les pièces par un tableau à deux dimensions qui se justifiait par la facilité d'accès aux éléments (étant fait en O(1)) ce qui allégerait la complexité des algorithmes.

2.3.1 Générateur de niveaux

Nous avons suivi les étapes proposées lors de la génération d'une grille. Nous avons tout d'abord commencé par traduire les contraintes auxquelles est soumise chaque pièce selon

son emplacement dans la grille (dans le cas d'absence de contraintes nous choisissons aléatoirement une connexion True/False dans la direction étudiée), puis avons créé la pièce satisfaisant ces contraintes à partir du constructeur prenant en paramètre le code traduisant les états des connexions N,S,E,O. Une fois la création des pièces achevées, on applique sur chaque pièce un nombre aléatoire de rotation afin de mélanger les positions initiales

2.3.2 Vérificateur

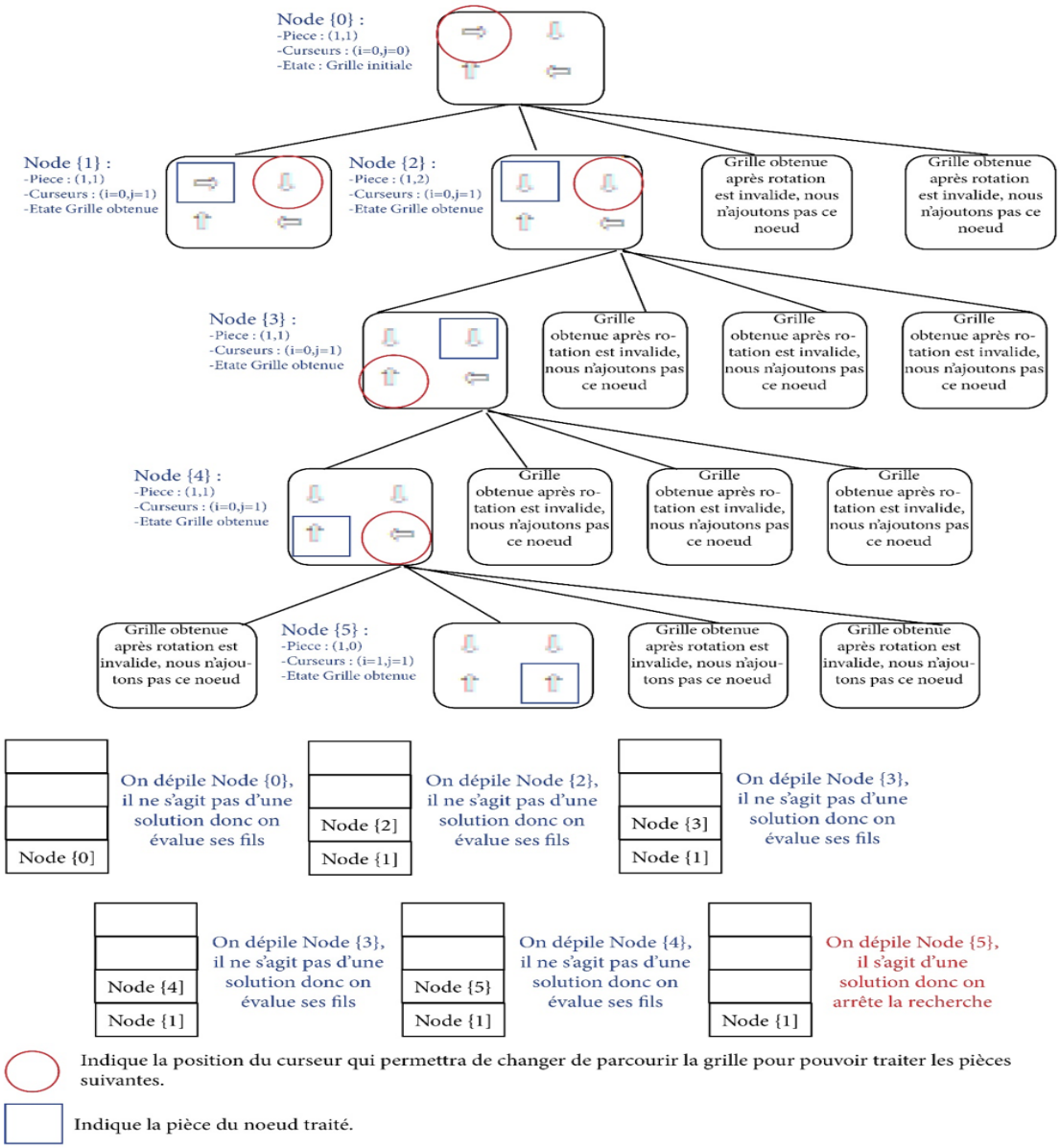
Nous avons créé le vérificateur de telle sorte qu'il puisse vérifier la validité d'une partie de la grille et de sa totalité en introduisant deux paramètre largeur courante et longueur courante. L'introduction de vérification d'une partie de la grille a été motivée par le besoin de vérification lors du processus de résolution dans la grille qui sera traité par la suite. La méthode appliquée pour vérifier se faisait en vérifiant, pour chaque pièce de la grille, si elle respectait toutes les contraintes de connexion (en fonction de sa position dans la grille et des voisins qu'elle a). Si toutes les pièces respectent toutes les contraintes, le niveau représente donc une solution.

2.3.3 Solveur

La méthode de résolution de la grille se rapprochait des algorithmes de recherche dans des arbres que nous avons traité en cours d'intelligence artificielle. C'était donc une occasion pour mettre en œuvre l'un de ses algorithmes. Afin d'implémenter cet algorithme nous avions besoin d'expliciter la représentation de 4 éléments : Le nœud de l'arbre de recherche, l'état correspondant à ce nœud, la méthode d'évaluation des fils du nœud (l'action à prendre afin d'assurer la transition entre deux nœuds) et la condition d'arrêt de la recherche. La stratégie retenue pour parcourir l'arbre est celle exposée dans le sujet du projet.

Dans le cas de notre problème, nous avons choisi de représenter un nœud de l'arbre par une pièce et sa position dans la grille, l'état du noeud sera la grille obtenue avec la pièce de ce nœud, la fonction de transition permettra de créer au plus 4 fils au nœud en appliquant des rotations à la pièce du nœud (un fils sera créé si et seulement si l'état du fils est partiellement valide càd des positions (0,0) jusqu'à la position de la pièce du nœud, ceci sera vérifié par la fonction Check de la grille). Le déroulement de l'algorithme se fera en ajoutant et retirant des nœuds dans une stack, la condition d'arrêt sera l'aboutissement a un état résolu d'un nœud retiré de la pile.

Nous présentons dans ce qui suit un exemple de déroulement de l'algorithme de recherche.



Avec cette configuration, l'arbre aura au plus n^4 nœuds où n est le nombre de pièce dans la grille et **4** représente le nombre de fils maximal que peut avoir chaque nœud. Nous remarquons dès lors la complexité en mémoire engendrée par ce problème, il était donc nécessaire d'ajouter d'autres contraintes afin de minimiser la création de nœuds lors du développement des fils (nous avons implémenté les deux amélioration algorithmes demandées dans le sujet).

2.4 Diagramme de classe

Le Diagramme de classe ci-dessous représente les classes intervenant dans le système.C'est la représentation statique des éléments qui composent le jeu ainsi que leurs relations.

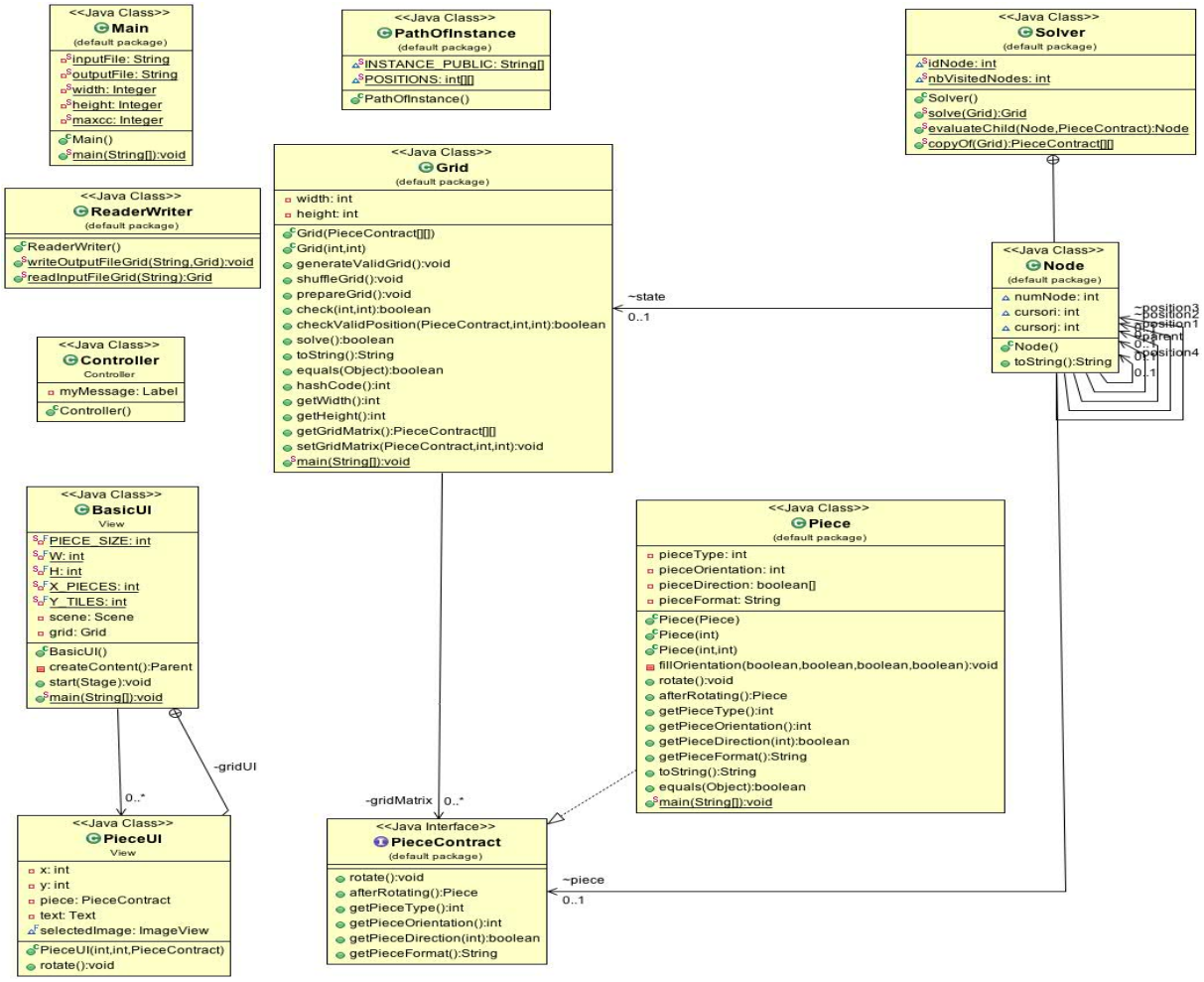


FIGURE 1 – Diagramme de Classe

3 Difficultées rencontrées

3.1 Difficultées d'implémentation

3.1.1 Point de départ

Il consistait à trouver un fil directeur pour la conduite du projet et dresser une timeline afin d'assurer son avancement. Cette étape était plutôt délicate puisqu'il n'y avait pas d'exemples similaires à ce projet pour servir de référence et d'autre part parce que le sujet dressant les spécifications n'était pas très explicite sur quelques points et laissait plutôt une certaine liberté qui nécessité un effort de réflexion pour comprendre ce qui est demandé avant de chercher une solution.

3.1.2 Modélisation d'une pièce

Le tout premier défi du projet était de bien choisir les structures de données pour modéliser une pièce puisque cela en découlait la suite du projet et une mauvaise modélisation

conduirait certainement à reprendre le code. Nous nous sommes donc basés sur les spécifications du sujet (plus particulièrement la notion de type et d'orientation d'une pièce) pour entamer cette étape, puis nous nous sommes rendu compte qu'il était nécessaire d'introduire la notion de connexions de chaque pièce afin d'implémenter les contraintes.

3.1.3 Générateur et vérificateur de niveau

La difficulté principale que nous avons rencontré lors de la génération de niveaux était de bien comprendre les différentes contraintes et de les transformer au niveau de l'implémentation de la classe pièce afin de pouvoir les utiliser par la suite au cours de la génération. Après plusieurs tentatives de modélisation des contraintes et à l'aide du tutoriel [1] nous avons réussi à générer une grille valide en suivant la méthode proposée sur le sujet.

La deuxième difficulté était de créé une pièce qui satisfaisait les contraintes, la solution que nous avons proposée était le constructeur qui prenait un code représentant les contraintes. L'étape d'écriture de fichier représentant le niveau nécessitait uniquement de lire les membres type et orientation de chaque pièce dans la grille.
Le code du vérificateur découlait naturellement des concepts implémentés sur le générateur. Nous n'avons pas trouvé de difficulté particulière pour le développer

3.1.4 Solveur

La première difficulté rencontrée lors du développement du solveur était la compréhension de la méthode proposée pour ce sujet et de faire le rapprochement avec les algorithmes de recherche dans un arbre. La deuxième difficulté, qui était plus compliquée à résoudre, était de modéliser l'arbre de recherche et plus particulièrement de bien définir la transition qui permet de créer des nœuds fils. La méthode que nous avons utilisée pour résoudre les bugs rencontrés dans cette étape était de simuler les fonctionnements sur des grilles simples (3x3 ou 2x2) pour lesquelles on pouvait construire les arbres à la main, comparer le comportement attendu de l'algorithme et son comportement réel, puis ajuster le code au fur et à mesure.

3.1.5 Utilisation de Github, Maven et répartition du travail

Ayant déjà pris en main l'outil Git à travers les TD/TP, nous n'avons pas eu de problème particulier avec son utilisation, mise à part les Merge Conflict qui se produisait puisqu'on modifiait des portions du codes simultanément. Toutefois, nous parvenions à résoudre ces conflits en suivant les étapes indiquées sur le guide d'utilisateur de Github. Nous nous sommes également rencontrés à mainte reprise afin de réfléchir sur les difficultés rencontrées et éventuellement développer les solutions ensemble.

L'utilisation de Maven était par ailleurs un autre défi à surmonter et dont le résultat se manifestait directement sur la compétition de benchmarking. Le problème que nous avons rencontré était que notre *pom.xml* générait deux fichiers Jar et que le script du benchmark utilisait le Jar qui n'intégrait pas toutes les dépendances. Après quelques tentatives nous avons réussis à générer le Jar qui comportait les dépendances uniquement en excluant

la génération du default-jar de la phase de Build. Il est nécessaire de mentionner que ce projet était l'occasion pour se familiariser avec structure d'un projet sur Maven et les différentes branches du fichier pom.xml.

3.2 Difficultés conceptuelles

Au cours du projet, nous avons pensé à respecter les consignes (données lors du cours) pour développer un 'code propre'. Nous avons donc voulu introduire au mieux les notions d'encapsulation, d'abstraction et de polymorphisme. La création d'une interface PieceContrat définissait d'une part le contrat d'implémentation d'une pièce ce qui permettait de découpler la tâche de développement de la grille et de la pièce et d'autre part de mettre en œuvre la notion de polymorphisme puisqu'on utilisait l'implémentation réelle du type PieceContract qui était la classe Piece (le compilateur appelle les membres selon leur type réel).

Par ailleurs, nous avons tâché de bien choisir les portées des différents membres de chaque classe afin d'assurer l'encapsulation du code et sa modularité. Enfin, nous avons prêté une attention particulière à ce que notre code puisse se développer de manière lucide afin de garantir sa généricité. Le type de problèmes cités sur le sujet (qui pourront faire l'objet d'extensions du projet) peuvent être implémentés uniquement en ajoutant les contraintes supplémentaires et en changeant les conditions de développement de fils selon les nouvelles spécifications du problème en question

3.3 Gestion d'entrée-sortie

Nous n'avons pas trouvé de difficulté particulière pour l'implémentation de la classe ReaderWriter puisqu'il y avait assez de ressources disponibles sur Internet et qui indiquait les méthodes à suivre. On s'est inspiré particulièrement de ce tutoriel [2] pour réussir cette tâche. Toutefois, un bug sur le lecteur de fichier conduisait à ce que le script de benchmarking ne marchait pas comme prévu sur notre code. Nous avons réussi à résoudre ce bug après consultation du prof.

4 Fonctionnalités non implémentées

4.1 Générateur de niveau avec composantes connexes

Nous n'avons pas pu implémenter cette option puisque nous n'avons pas pu traduire la notion de connexité en fonctionnalité à coder.

4.2 Solveur

L'algorithme que nous avons développé n'étant pas optimisé (absence de méthode de choix de pièce et de multi-thread), sanctionnait fortement l'efficacité de notre solveur. En effet, d'après la compétition de benchmark, notre solveur réalise 52 Timeout et 29 Unknown. Les Timeout sont le résultat d'un algorithme lent ce qui s'expliquerait par l'absence de méthode de choix de pièce (qui est censé diminuer le nombre de nœuds

visités) et les Unknown seraient le résultat de création d'un nombre excessifs de nœuds (dans nos tests cela atteignait 1000000 nœuds avec certains niveaux), ce qui conduirait naturellement à une OutOfMemoryException.

4.3 Méthodes de choix de pièce

Malgré plusieurs tentatives, nous n'avons pas pu saisir le sens de ce qui est demandé dans cette section à travers la description fournie sur le sujet, il nous était difficile d'imaginer comment on pourrait développer l'arbre de recherche toute en choisissant une pièce particulière pour développer ses fils (c'est bien ce que nous avons compris dans cette partie).

4.4 Solveur multi-thread

Nous n'avons pas eu le temps pour développer cette fonctionnalité.

4.5 Interface graphiue

Nous n'avons pas eu le temps pour implémenter l'interface graphique mais nous avons tout de même tenter dímplementer l'interface statique sauf qu'elle produit une exception lorsque nous la testons sur d'autres ordinateurs.

5 Conclusion

Le projet dans sa globalité a été instructif et permettait de maîtriser non seulement les concepts de base de la programmation Java vu en cours, mais également de prendre en main les outils Git et Maven ce qui était un plus considérable.

La structure avec laquelle le projet a été conçu et présenté était dans l'ensemble suffisante pour comprendre ce qui était demandé, et permettait de subdiviser le sujet en plusieurs sous tâches grâce auxquelles on pouvait voir l'avancement du développement à travers la compétition de benchmarking. Le sujet, n'ayant pas imposé de spécifications techniques particulières, nous a donné assez de liberté de choix ce nous faisait réfléchir sur nos choix et leurs conséquences et par conséquent nous permettait de réaliser l'importance des choix architecturales.

Par ailleurs, les indications données dans le sujet n'étant quelques fois pas très claire, plus particulièrement la section qui détaillait la méthode de choix de pièce, menait à un blocage qui durait quelques jours, ce qui était démotivant parfois. Cela aurait été plus intéressant d'illustrer schématiquement ce qui était attendu de l'algorithme dans la partie de choix de pièce.

Références

[1] Wabble, *Unity Infinity Loop Clone Tutorials*, 21jul. 2016 https://www.youtube.com/playlist?list=PLQzQtnB2ciXTv60MCjghfyet-h3RNhjDG

[2] Dreamincode forum , *Problems Writing Into .Dat File*, Dreamincode 16Jan. 2009 http://www.dreamincode.net/forums/topic/81147-problems-writing-into-dat-file/