# Rapport pour le projet C.M.I.

Mattéo Delabre
Groupe 1A1

29 février 2016

# Table des matières

# Introduction

L'objectif du module Projet C.M.I. de second semestre est de développer un jeu en groupe, ce qui permet de s'initier à la gestion d'un projet, à la communication et à la collaboration au sein d'une équipe, mais aussi de mobiliser les connaissances acquises en matière d'algorithmique au premier semestre.

Ce rapport présente ainsi trois jeux vidéos qui se prêtent à cet exercice : le jeu de la vie, le jeu de go et un jeu original basé sur les jeux de plateformes. Pour chacun de ces jeux, on étudie son principe, mais aussi les manières de le représenter dans une machine et les algorithmes pouvant être utilisés pour simuler le jeu.

Cette étude est également l'occasion de réfléchir sur les éventuelles difficultés qui seront rencontrées pendant le développement. Est également présentée une organisation possible pour la création du jeu.

Enfin, les trois jeux sont comparés à la lumière de leur intérêt algorithmique, de leur difficulté de développement et du temps nécessaire pour les créer.

# Jeu 1

# Jeu de la vie

Le jeu de la vie, inventé par John Conway en 1970, est l'exemple le plus populaire d'automate cellulaire. [1] Un automate cellulaire est un ensemble de cellules pouvant être dans plusieurs états différents et dont l'état suivant est entièrement déterminé à partir de l'état actuel. [2]

Il présente un intérêt théorique car il a été démontré que le jeu de la vie est Turing-complet, c'est-à-dire que n'importe quel algorithme peut y être implémenté.

## 1.1 Principes

Le jeu de la vie est composé d'une grille infinie de cellules pouvant être soit mortes soit vivantes. Le jeu est constitué d'états : un état du jeu est l'ensemble des états (vivante ou morte) de chacune de ses cellules à un moment précis. Seul l'état initial est fourni, les états suivants sont calculés de proche en proche selon les règles suivantes :

1. si une cellule a trois voisines à l'état $n$, elle est vivante à l'état $n + 1$ ;
2. si une cellule a deux voisines à l'état $n$, elle persévère dans son état ;
3. si une cellule a moins de deux ou plus de trois voisines à l'état $n$, elle est morte à l'état $n + 1$. [3]

Les figures 1.1, 1.2 et 1.3 montrent trois situations typiques du jeu de la vie sur des portions de grilles. Dans ces figures, une case ■ représente une cellule qui vient de naître, une case ■ représente une cellule qui va mourir à l'état suivant et une case ■ représente une cellule vivante.

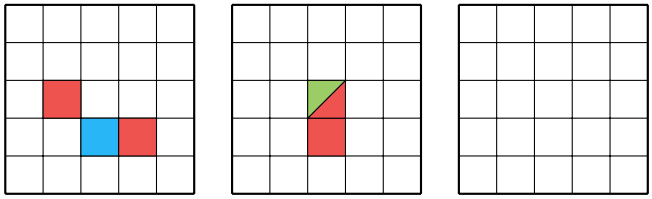


FIGURE 1.1 – Mort par sous-population

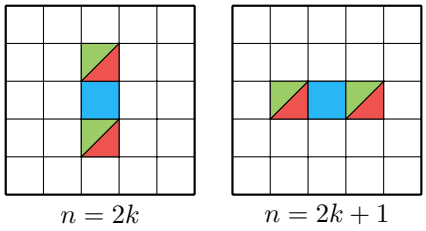


FIGURE 1.2 – Un oscillateur, une configuration qui se répète indéfiniment

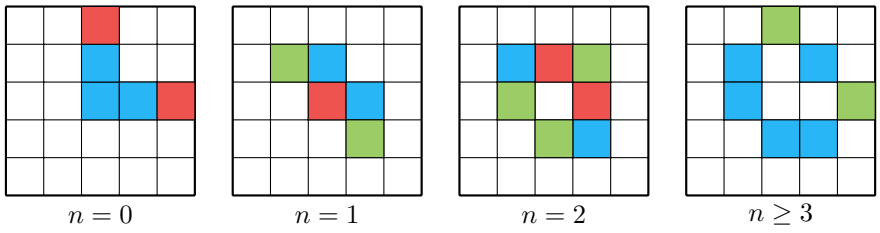


FIGURE 1.3 – Stabilisation de la configuration en quatre états

## 1.2 Modélisation

Les cellules peuvent prendre uniquement deux états, vivante ou morte. Pour cette raison, il est naturel de représenter une cellule par un booléen, `true` pour une cellule vivante et `false` pour une cellule morte. Pour la représentation d'une grille de cellules, on utilisera donc un tableau de tableaux de booléens.

Les tableaux seront de taille suffisamment grande pour éviter qu'une structure n'arrive à la bordure. En effet, le comportement à la bordure d'une grille n'est pas défini dans les règles du jeu de la vie puisqu'il est censé se dérouler sur une grille infinie.

## 1.3 Algorithmes

Quatre algorithmes doivent être définis pour gérer la grille.

**Initialisation de la grille** Allocation d'une grille de taille donnée et initialisation de tous les booléens à `false`.

**Détermination du nombre de voisins vivants** Reçoit une grille $G$ et une position de cellule $(x, y)$.

1. Soit $total := 0$.
2. Pour $i$ allant de $x - 1$ à $x + 1$ :
   (a) pour $j$ allant de $y - 1$ à $y + 1$, si $(i, j)$ est une position valide et $(i, j)$ est une cellule vivante de $G$ et $(i, j) \neq (x, y)$, alors $total := total + 1$.
3. Renvoyer $total$.

**Calcul de l'état suivant** Reçoit une grille $G$ et calcule la grille représentant l'état suivant du jeu.

1. Soit $G'$ une grille de même taille que $G$.
2. Pour $i$ allant de 0 à la taille horizontale de $G$ :
   (a) pour $j$ allant de 0 à la taille verticale de $G$ :
      i. appeler l'algorithme **Détermination du nombre de voisins vivants** sur la grille $G$ et la case $(i, j)$ et stocker le résultat dans $voisins$ ;
      ii. si $voisins = 3$, $G'(i, j) :=$ `true` ;
      iii. sinon, si $voisins = 2$, $G'(i, j) := G(i, j)$ ;
      iv. sinon, $G'(i, j) :=$ `false`.
3. Renvoyer $G'$.

**Affichage de la grille** Doit parcourir chaque cellule d'une grille donnée pour afficher à l'écran l'état de celle-ci. L'algorithme dépendra principalement du type d'affichage choisi (terminal, fenêtré).

Le programme principal se charge d'appeler l'algorithme 1, puis d'appeler de manière répétée les algorithmes 3 et 4 pour afficher les états suivants.

## 1.4 Spécifications

### 1.4.1 Version initiale

On choisira le langage C++, qui possède les structures requises dans la section précédente, et est enseigné dans le cursus. Il n'y a pas de difficulté algorithmique particulière qui justifie le choix d'un langage différent, sachant que le choix d'un tel langage pourrait ralentir le développement.

L'affichage de la grille se fera dans le terminal. Un algorithme naïf sera choisi pour le calcul de l'état suivant, se contenant de parcourir chaque cellule et de calculer son état, sans recherche d'optimisation.

L'état initial sera choisi par l'utilisateur parmi une série de préconfigurations codées *en dur* dans le programme.

Le type `std::vector<std::vector<bool>>` de la librairie standard C++ sera utilisé pour représenter les grilles car il permet une allocation dynamique automatiquement gérée.

### 1.4.2 Améliorations possibles

L'affichage peut s'effectuer en fenêtré. Les bibliothèques SDL (en C) ou SFML (en C++, orienté objet) peuvent être choisies et seront utilisées dans l'algorithme d'affichage pour le dessin des cellules, et dans le programme principal pour initialiser la fenêtre.

On peut laisser l'utilisateur choisir l'état initial en cliquant sur les cellules. L'utilisateur pourra contrôler la génération grâce à un bouton marche/arrêt, pas à pas, ou remise à zéro. On peut également fournir une série de préconfigurations parmi lesquelles choisir dans l'interface.

Enfin, un algorithme plus performant peut être utilisé, réduisant les calculs inutiles. Par exemple, l'algorithme *Hashlife,* [4] utilisant des *quadtrees* et une table de hachage peut améliorer considérablement les performances (mais peut être difficile à implémenter).

## 1.5 Organisation

Pour la version initiale du programme, il y a quatre algorithmes à mettre en place. Ces algorithmes reprennent des concepts vus en cours. On pourra y consacrer 10 heures.

Pour l'affichage fenêtré, il faudra se renseigner sur l'A.P.I. des bibliothèques utilisées, concevoir une interface (placement des boutons, de la grille) et l'implémenter. On pourra prévoir 10 heures.

L'implémentation de l'algorithme *Hashlife* requièrera des renseignements sur l'utilisation des *quadtrees* et des tables de hachages ainsi que sur l'algorithme en lui-même. Prévoir 20 heures.

La rédaction du rapport s'effectuera en continu pendant la création du jeu. La figure 1.4 présente un diagramme de Gantt résumant la répartition du travail.

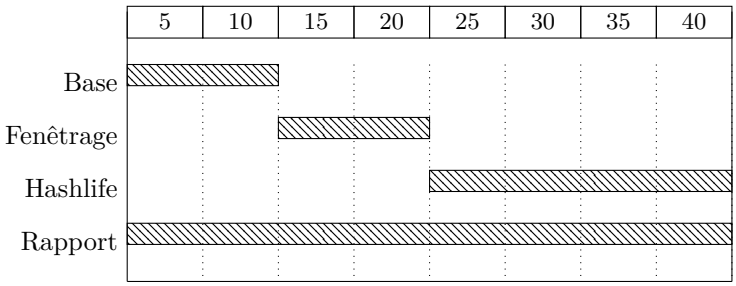


FIGURE 1.4 – Développement du jeu de la vie sur 40 heures

# Jeu 2

# Jeu de go

Joué en occident depuis le XIX siècle, le jeu de go est originaire d'Asie et vieux de plusieurs milliers d'années. La société étasunienne de Go le décrit comme « le plus ancien jeu toujours joué sous sa forme originale. » [5] Comme les échecs, le jeu de go est déterminé, à information complète et parfaite : à tout moment tous les joueurs ont la même information à disposition pour décider de leur coup et il n'y a pas de hasard. [6]

Malgré la simplicité apparente du jeu, le nombre de combinaisons possibles s'élève à plus de $10^{600}$, ce qui en fait l'un des objectifs non-atteints les plus anciens en matière de recherche en intelligence artificielle. Le programme *AlphaGo* de *Google* tentera en mars prochain de rivaliser avec le meilleur joueur de go au monde, Lee Sedol. [7, 8]

## 2.1 Principes

### 2.1.1 Matériel

Le jeu se joue sur un plateau de $18 \times 18$ cases (soit $19 \times 19$ intersections) appelé *goban.* Deux joueurs s'affrontent en posant à tour de rôle des pierres blanches et noires.

### 2.1.2 Règles

Lorsqu'un joueur doit jouer, il peut soit poser une pierre de sa couleur sur une des intersections du *goban,* soit passer son tour. Si les deux joueurs passent successivement, la partie est terminée. [9]

On définit une chaîne comme étant une zone de pierres interconnectées horizontalement ou verticalement (mais pas diagonalement). Les *libertés* d'une telle chaîne sont le nombre d'intersections vides autour d'elles (horizontalement, verticalement, mais pas diagonalement).

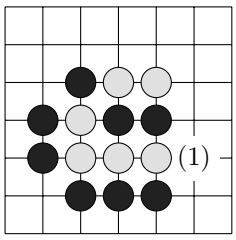


FIGURE 2.1 – La chaîne blanche sera capturée par la pose d'une pierre noire en (1)

Si, en plaçant une pierre, un joueur supprime la dernière liberté d'une chaîne adverse, la chaîne en question est capturée et retirée du plateau, comme montré en figure 2.1. Un joueur peut placer sa pierre n'importe où sur le *goban,* pour peu que cela ne supprime pas toutes les libertés d'une de ses chaînes et que cela ne répète pas une position précédente du jeu.

### 2.1.3 Fin du jeu

À la fin de la partie, c'est-à-dire après que les deux joueurs ont passé leur tour, on décompte les points. Chaque pierre présente sur le *goban* rapporte un point, ainsi que chaque intersection vide à l'intérieur du territoire d'un joueur. Un territoire d'un joueur est défini comme étant une zone inoccupée du plateau, séparée du reste uniquement par des pierres de la couleur attribuée à ce joueur. La figure 2.2 montre des exemples de territoires.

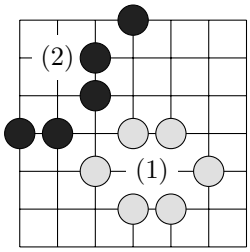


FIGURE 2.2 – Un territoire du joueur attribué aux blancs en (1) et un territoire du joueur attribué aux noirs en (2)

## 2.2 Modélisation

Seules les $19 \times 19$ intersections de la grille sont utilisées, pas les cases. On pourra donc utiliser un tableau de tableaux de taille $19 \times 19$. À tout moment, chaque case peut soit être vide, soit occupée par une pierre noire, soit occupée par une pierre blanche. On optera donc pour un tableau de tableaux d'entiers,

en représentant par 0 l'état vide, 1 la présence d'une pierre noire et 2 la présence d'une pierre blanche.

On maintiendra en tout temps une liste des chaînes actives, avec une liste des intersections occupées par ces chaînes et le nombre de leurs libertés. Après chaque coup, la grille sera hachée et le résultat sera ajouté dans une liste appelée la liste des positions précédentes.

## 2.3 Algorithmes

**Initialisation** Allocation d'une grille de taille $19 \times 19$, initialisée à 0.

**Placement d'une pierre** Étant donnés une position et la couleur de la pierre à placer.

1. Si la pierre se situe en dehors de la grille, le coup est invalide. Terminer l'algorithme.
2. Faire une copie de la grille et de la liste des chaînes. Dans le reste de l'algorithme, on opérera uniquement sur ces copies, sauf mention contraire.
3. À la position choisie dans la grille, affecter l'entier correspondant à la couleur du joueur (1 ou 2).
4. Hacher la grille et comparer l'empreinte à la liste des positions précédentes. S'il y a correspondance, le coup reproduit un état de jeu déjà atteint, donc le coup est invalide. Terminer l'algorithme.
5. Identifier toutes les chaînes voisines horizontalement et verticalement à la pierre placée. Mettre à jour les libertés et les pierres composant ces chaînes.
6. Si une chaîne voisine de la couleur adverse n'a plus aucune liberté, effacer toutes ses pierres dans la grille et supprimer la chaîne de la liste des chaînes. Mettre à jour les libertés des chaînes voisines.
7. Si une chaîne voisine de la couleur du joueur jouant le coup n'a plus aucune liberté, le coup est invalide. Terminer l'algorithme.
8. Appliquer les grilles copiées dans les grilles originales.
9. Ajouter l'empreinte de la grille à la liste des positions précédentes.

**Décompte des points** On initialise deux compteurs pour les points du joueur attribué aux noirs et les points du joueur attribué aux blancs. Pour parcourir les territoires des joueurs on utilise une variante de l'algorithme de remplissage par diffusion. [10]

1. Parcourir la grille et attribuer un point par pierre placée à chaque joueur.
2. Créer une grille $G$ de même taille que la grille de jeu, initialisée à 0.

3. Pour chaque case $C$ dans la grille de $(0,0)$ à $(18,18)$, si la case n'est pas vide ou si $G[C] = 1$, passer à la case suivante, sinon :
   (a) initialiser une liste $L$ contenant uniquement $C$ ;
   (b) initialiser un compteur *cases* à 0 ;
   (c) initialiser une variable *couleur* vide ;
   (d) tant que la liste $L$ n'est pas vide, faire :
      i. prendre $D$ la première case de $L$ ;
      ii. supprimer le premier élément de $L$ ;
      iii. si la case $D$ est vide, passer $G[D]$ à 1, ajouter les cases au nord, au sud, à l'est et à l'ouest de $D$ dans $L$ si elles ne sont pas telles que $G[X] = 1$, et incrémenter *cases* ;
      iv. sinon, si *couleur* est vide, $couleur := couleur(D)$ ;
      v. sinon, si $couleur \neq couleur(D), couleur := mixte$ ;
   (e) si $couleur = noir$, ajouter *cases* points au joueur attribué aux pierres noires. Si $couleur = blanc$, ajouter *cases* points au joueur attribué aux pierres blanches.

**Affichage** La grille sera parcourue case par case. Chaque valeur différente de 0 provoquera le placement d'un pion sur l'intersection correspondante de la couleur correspondante. Le programme devra être réceptif aux clics sur la grille et appeler l'algorithme de placement d'un pion en conséquence.

## 2.4 Spécifications

### 2.4.1 Version initiale

On choisira le langage C++, qui possède les structures requises dans la section précédente, et est enseigné dans le cursus. Il n'y a pas de difficulté algorithmique particulière qui justifie le choix d'un langage différent, sachant que le choix d'un tel langage pourrait ralentir le développement.

On préfèrera un affichage fenêtré au vu des nombreuses interactions qui seront facilitées par l'usage de la souris. On proposera une grille de taille fixe $19 \times 19$. Les deux joueurs utiliseront la même fenêtre, plaçant leur pierre à tour de rôle.

Les types standards, comme `std::vector` ou `std::unordered_map`, seront utilisés pour représenter les structures abordées dans la section traitant de la modélisation.

### 2.4.2 Améliorations possibles

Une interface plus travaillée pourra être proposée avec un choix parmi différentes tailles standard de grille telles que $9 \times 9$ et $13 \times 13$. Les joueurs pourront s'affronter en réseau.

## 2.5 Organisation

L'implémentation des algorithmes demandera de la documentation sur les hachages, les tableaux associatifs et autres structures abordées ci-avant. Il y a deux algorithmes principaux : le décompte des points et le placement d'une pierre. On consacrera à la mise au point des algorithmes initiaux et à leur perfection 40 heures au total.

Pendant le développement des algorithmes, le fenêtrage pourra être conçu avec l'interface de choix des grilles et de présentation des résultats. On pourra y consacrer 10 heures.

La mise en réseau du jeu se fera lorsque les algorithmes seront suffisamment robustes. Elle nécessitera de la documentation sur les *sockets* avec la SFML. On pourra y consacrer 15 heures.

La rédaction du rapport s'effectuera en continu pendant la création du jeu. La figure 2.3 présente un diagramme de Gantt résumant la répartition du travail.

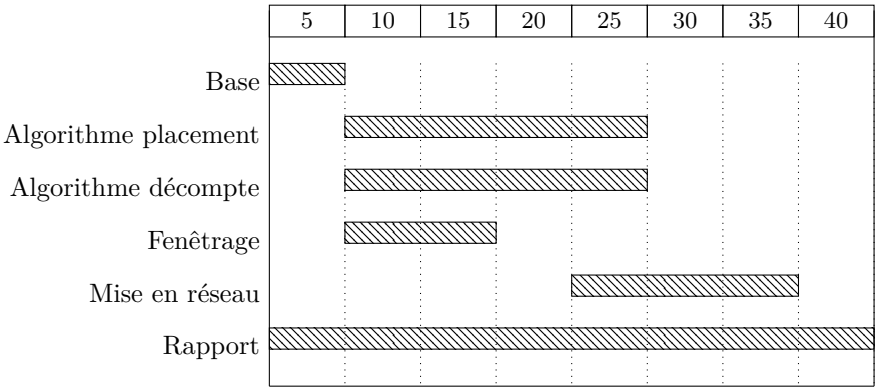


FIGURE 2.3 – Développement du jeu de go sur 40 heures

# Jeu 3

# Jeu de plateformes coopératif

Les jeux de plateformes sont un des premiers genres à avoir émergé dans le monde du jeu vidéo. Ils consistent à faire progresser un personnage en le faisant aller de plateforme en plateforme, d'où leur nom.

Le troisième jeu étudié a un principe original, inspiré des jeux de plateformes, de coopération et de réflexion. Il s'agira pour deux joueurs de faire traverser deux balles aimantées à travers un niveau, certains éléments du décor étant eux aussi aimantés.

La conception du jeu s'appuiera sur cet élément physique pour proposer des niveaux sous forme de casse-têtes, dans lesquels les deux participants devront s'entraider et réfléchir pour parvenir à la fin.

## 3.1 Principes

Deux joueurs doivent s'entraider pour faire avancer leur balle à travers des niveaux. Les joueurs exploitent pour ce faire les mécanismes physiques définis dans la section 3.1.1.

Le jeu est constitué d'une suite de niveaux, chaque niveau étant une grille de blocs en deux dimensions. Ces grilles sont définies en avance par le programmeur. Les blocs interagissent avec les balles comme défini dans les sections 3.1.1 et 3.1.3.

Les joueurs valident un niveau en faisant parvenir toutes les balles dans une zone d'arrivée prédéfinie. Ils passent ainsi au niveau suivant. Le but du jeu est de terminer tous les niveaux.

### 3.1.1 Joueurs

Les deux joueurs contrôlent chacun une balle. Ces balles ont pour propriétés leur position dans le plan, leur vitesse et leur charge électrique. Les balles évoluent par interactions avec le joueur et par interactions physiques.

Un joueur peut interagir avec sa balle de trois manières : il peut lui donner de la vitesse vers la gauche de la fenêtre, la droite de la fenêtre, ou inverser la polarité de la balle. Le joueur ne peut pas faire « sauter » sa balle.

L'évolution des balles, en dehors du contrôle des joueurs, est conditionnée par les phénomènes physiques suivants :

— une force de gravité qui agit en tout point et attire les balles vers le bas, le haut, la gauche ou la droite de la fenêtre. L'orientation de cette force peut être modifiée par les actions d'une balle en cours de jeu ;
— une force de réaction qui agit de telle sorte que les balles ne puissent pas traverser les blocs ;
— des forces de frottements lorsque la balle se situe au contact d'un bloc. L'intensité de cette force varie en fonction des types de blocs ;
— une force d'interaction coulombienne entre les éléments du jeu qui sont polarisés (c'est-à-dire les balles et certains blocs tels que définis en section 3.1.3).

### 3.1.2 Caméra

À tout moment les deux balles peuvent se situer n'importe où dans le niveau. Il faut toutefois faire en sorte qu'elles soient toutes deux visibles à tout moment.

Pour ce faire, la caméra est centrée sur la position moyenne des deux balles si celles-ci sont suffisament proches. Sinon, l'écran est divisé en deux et chaque partie est centrée sur chacune des balles.

### 3.1.3 Blocs

Le niveau est une grille de blocs. Les blocs possèdent une position sur la grille déterminée par une paire d'entiers et une charge électrique qui peut être annulée pour que le bloc n'attire aucun objet. Certains blocs peuvent modifier le sens de la gravité lorsqu'une balle rentre en collision avec eux. Enfin, les blocs sont statiques et ne sont pas soumis à la physique du jeu.

## 3.2 Modélisation

Les balles sont modélisées par une classe `Ball`. Cette classe est dotée des propriétés `position`, `velocity`, `mass` et `charge`. Le vecteur `position` représente la position de la balle dans le plan. Le vecteur `velocity` représente la vitesse de la balle. Le flottant `mass` représente la masse de la balle, qui sera utilisée dans le calcul de l'accélération. Enfin, le flottant `charge` représente sa charge électrique.

Les blocs sont modélisés par une classe `Block` munie des propriétés `position` et `charge` qui représentent la position du bloc sur la grille de jeu et sa charge. La charge du bloc est nulle s'il n'est pas polarisé. La classe sera étendue pour représenter des types de blocs spécialisés.

Enfin, une classe principale `Engine` est chargée de coordonner les éléments du jeu et d'organiser le dessin des *frames*. Elle est dotée d'un tableau à deux dimensions d'instances de `Block` qui représente la grille de jeu, d'un tableau à une dimension d'instances de `Ball` qui stocke toutes les balles dans le niveau, et d'une horloge qui mesure le temps écoulé entre chaque *frame*.

La figure 3.1 présente les classes utilisées pour la modélisation. Les méthodes de ces classes sont détaillées dans la section suivante.

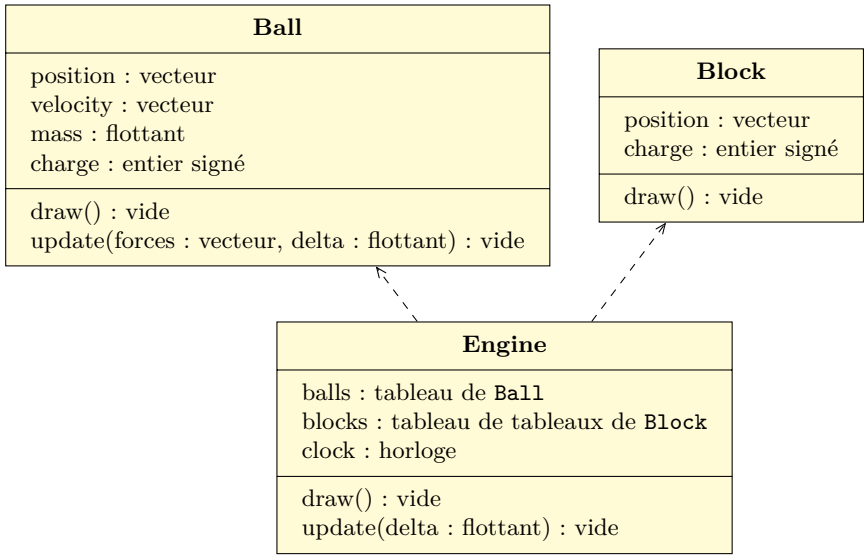


FIGURE 3.1 – Diagramme des classes utilisées

## 3.3 Algorithmes

### 3.3.1 Physique

Les balles sont équipées de propriétés représentant leur position, leur vitesse, leur masse et leur charge comme vu dans la section précédente. On utilise l'intégration explicite d'Euler pour calculer la nouvelle position de chaque balle à chaque *frame*. [11] Cette méthode a l'avantage d'être simple à implémenter et rapide.

**`Engine::update(delta)`** L'algorithme reçoit le temps écoulé depuis la dernière *frame*. Il calcule les forces à appliquer à chaque balle, puis appelle la procédure `Ball::update(forces, delta)` en conséquence.

1. Pour chaque `ball` dans `balls`.
2. Initialiser un vecteur `forces` au vecteur nul.
3. Ajouter le vecteur $(0, g)$, $(0, -g)$, $(g, 0)$, $(-g, 0)$ selon la direction de la gravité au vecteur `forces`.
4. Si la touche pour faire aller la balle `ball` à gauche est enfoncée, ajouter le vecteur $(-m, 0)$ au vecteur `forces`.
5. Si la touche pour faire aller la balle `ball` à droite est enfoncée, ajouter le vecteur $(m, 0)$ au vecteur `forces`.
6. Pour chaque autre élément polarisé, appliquer une force d'attraction portée par la droite passant par les deux éléments et de norme $c \times \frac{\texttt{charge}_1 \times \texttt{charge}_2}{\texttt{distance}^2}$.
7. Gérer les collisions et les frottements.
8. Appeler `Ball::update(forces, delta)` sur `ball`.

**`Ball::update(forces, delta)`** L'algorithme reçoit le vecteur somme de toutes les forces appliquées à la balle et un flottant qui contient le temps écoulé depuis la dernière *frame*. Il calcule la position suivante de la balle.

1. `acceleration` $:= \frac{\texttt{forces}}{\texttt{mass}}$.
2. Ajouter `acceleration` $\times$ `delta` à `velocity`.
3. Ajouter `velocity` $\times$ `delta` à `position`.

Les constantes $g$, $m$ et $c$ devront être définies et ajustées au cours de la conception du jeu pour que la simulation paraisse naturelle.

### 3.3.2 Dessin

La scène du jeu est composée de trois couches, l'une pour l'arrière-plan du jeu, une autre pour la grille de blocs, et la dernière pour les balles. Ces couches sont représentées sur les figures 3.2 et 3.3. Chaque objet susceptible d'être dessiné à l'écran possède une méthode `draw()`.

`Ball::draw()` Dessine la balle à sa position sur l'écran.

`Block::draw()` Dessine le bloc à sa position sur l'écran.

`Engine::draw()` Cet algorithme du moteur appelle les différentes fonctions de dessin dans un ordre spécifique pour que les trois couches de la scène du jeu soient correctement affichées.

1. Dessiner l'arrière-plan du jeu.
2. Dessiner la grille de blocs en appelant `Block::draw()` sur les blocs de `blocks` qui sont visibles à l'écran.
3. Dessiner le premier-plan en appelant `Ball::draw()` sur les balles de `balls` qui sont visibles à l'écran.

### 3.3.3 Moteur

**Initialisation du moteur `Engine`**

1. Créer et configurer la fenêtre d'affichage du jeu.
2. Initialiser `clock` à zéro.
3. Tant que la fenêtre est ouverte :
   (a) traiter tous les événements relatifs à la fenêtre (notamment l'appui sur une touche, la fermeture, le redimensionnement) ;
   (b) calculer le temps écoulé `delta` depuis la dernière *frame* et réinitialiser `clock` à zéro ;
   (c) appeler l'algorithme `Engine::update(delta)` ;
   (d) appeler l'algorithme `Engine::draw()`.

## 3.4 Spécifications

### 3.4.1 Version initiale

On choisira le langage C++, qui prend en charge le paradigme objet et dont la syntaxe est enseignée dans le cursus. On utilisera la librairie SFML pour son A.P.I. simple et puissante.

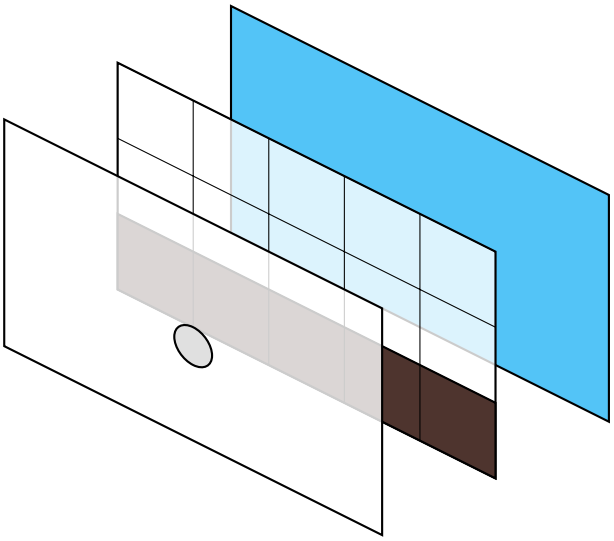

FIGURE 3.2 – Vue explosée des trois couches de rendu du jeu

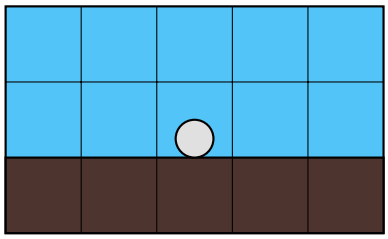

FIGURE 3.3 – Une configuration similaire à celle de la figure 3.2, vue de face

Les deux joueurs partageront le clavier d'une même machine.

On utilisera le type `sf::Vector2d` pour ce qui est vectoriel (position, vitesse, accélération, forces). Les tableaux utiliseront le type standard `std::vector`. Le type `sf::Clock` sera utilisé pour l'horloge du moteur. On choisira entre des flottants simple précision ou double précision en fonction des besoins.

Des recherches supplémentaires sont nécessaires notamment concernant les algorithmes de détection et de réaction aux collisions. [12, 13]

### 3.4.2 Améliorations possibles

L'algorithme physique utilise la méthode explicite d'Euler. Cette méthode engendre une erreur de simulation d'autant plus grande que l'espacement entre les *frames* est élevé, ce qui signifie que la physique du jeu se comportera différemment selon les performances de la machine. On pourra opter pour des méthodes plus précises comme l'intégration de Verlet [14] ou la méthode de Runge-Kutta classique. [15]

On pourra faire en sorte que le jeu s'exécute en réseau entre deux machines, une pour chaque joueur. Cela requièrera une synchronisation de la simulation physique et la mise au point d'un protocole de communication entre les deux machines.

## 3.5 Organisation

Dans un premier temps le moteur physique et graphique devra être conçu sur la base des algorithmes fournis dans le rapport (hormis l'algorithme de collision, qui devra faire l'objet de plus de recherches). Durant la mise au point de ce moteur, un niveau de test sera créé permettant l'appréciation et l'ajustement des variables physiques. On y consacrera 40 heures au total.

Ce niveau de test permettra également l'essai de différents mécanismes de *gameplay.* Ces éléments seront en parallèle intégrés dans les niveaux finaux. La conception et le test des niveaux se fera sur 80 heures.

Dans le même temps, l'univers graphique, notamment les textures, la décoration de l'interface, ainsi que la musique et les bruitages du jeu seront réalisés. Après la réalisation des éléments graphiques, on pourra concevoir l'interface du jeu. On y consacrera 50 heures.

Enfin, les tests finaux du jeu s'effecturont sur 5 heures. On pourra éventuellement demander l'aide de personnes extérieures pour ces essais.

La rédaction du rapport s'effectuera en continu pendant la création du jeu. La figure 3.4 présente un diagramme de Gantt résumant la répartition du travail.

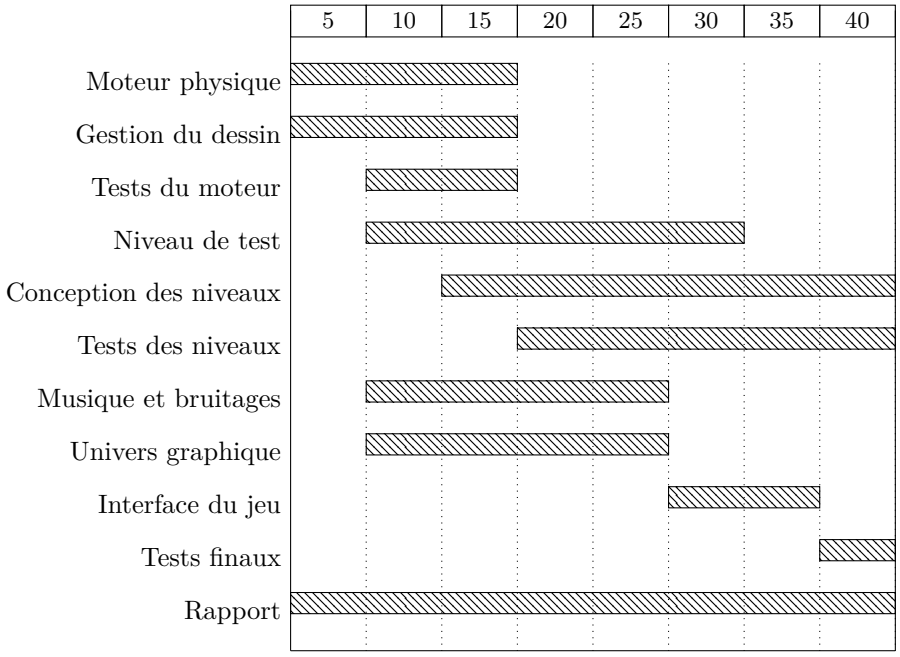


FIGURE 3.4 – Développement du jeu sur 40 heures

# Étude comparative

Le jeu de la vie permet, avec des règles très simples, de produire des motifs extrêmement complexes. Ces motifs peuvent nécessiter de nombreuses générations sur de grandes surfaces, d'où la nécessité d'un algorithme évitant les calculs inutiles. En effet, on peut remarquer que beaucoup de motifs se répètent au cours des générations et qu'il n'est pas utile de calculer l'état de zones isolées des cellules vivantes. C'est l'objectif de l'algorithme *Hashlife*.

D'un point de vue algorithmique, le jeu de go est également intéressant. Le décompte des points demande de pouvoir identifier les territoires des joueurs, ce à quoi un algorithme de remplissage par diffusion se prête bien, comme montré au cours de l'étude algorithmique. Cependant, le décompte peut aussi demander (en fonction des règles que l'on applique) de retirer les pierres « mortes » c'est à dire celles qui n'auraient pas pu être sauvées de la prise même en contiuant de jouer. On ne peut pas imaginer pouvoir concevoir un algorithme qui détecte de telles pierres sans avoir une intelligence artificielle assez performante pour pouvoir simuler les coups suivants. Les intelligences artificielles en matière de jeu de go restent peu performantes, car les techniques classiques de recherche de solutions employées notamment avec les échecs ne sont pas envisageables, au vu du nombre de possibilités.

Le jeu de plateformes présente un intérêt différent. D'une part, la quantité de travail à fournir est largement supérieure à celle des autres jeux présentés, et nécessite donc la coordination d'une équipe. D'autre part, les algorithmes à concevoir sont directement à mettre en relation avec l'enseignement de mécanique classique du premier semestre, ce qui permet de le valoriser. Enfin, un tel jeu a un résultat visuel plus à même d'être présenté en vidéo ou devant un public, puisqu'il est plus dynamique que le jeu de la vie ou le jeu de go.

Chaque jeu présenté dans ce rapport demande une organisation différente car le volume de travail requis va en croissant. En effet, alors que le jeu de la vie peut être développé par une seule personne en 40 heures totales, le jeu de plateformes demande au moins 175 heures et donc la collaboration d'au moins 3 personnes.

# Conclusion

Pour les raisons exposées dans l'étude comparative, j'ai choisi de développer le jeu de plateformes pour le projet C.M.I. de cette année. La réalisation d'un moteur physique reste un bon exercice algorithmique, tout en laissant place à une partie plus créative à travers la réalisation de l'univers graphique et de la musique.

Pour la création de ce jeu, un volume horaire d'au moins 175 heures est nécessaire. Il faut donc constituer une équipe d'au moins 3 personnes. Je pense que Maëlle Beuret et Rémi Cérès seraient les plus à même à m'accompagner sur ce projet.

# Webographie


[1] Wikipédia. Jeu de la vie. https://goo.gl/RLc7C7.

[2] Wikipédia. Automate cellulaire. https://goo.gl/FZ5S5L.

[3] LifeWiki. Conway's game of life. http://goo.gl/JXXQxz.

[4] Wikipédia. Hashlife. https://goo.gl/nFW2oM.

[5] American Go Association. A brief history of go. http://goo.gl/1NY9Xt.

[6] Wikipédia. Classification des jeux. https://goo.gl/guXBAO.

[7] SciencePost. Jeu de go : l'affrontement entre le meilleur joueur humain du monde et la machine sera retransmis en direct. http://goo.gl/pD8zKG.

[8] Jay Burmeister. An introduction to computer go. http://goo.gl/WMPpHU.

[9] Fédération Française de Go. Règle française du jeu de go. http://goo.gl/sK15PL.

[10] Wikipedia. Flood fill. https://goo.gl/ERbxi3.

[11] John T. Foster. Brief explanation of integration schemes. http://goo.gl/HRssYN.

[12] StackExchange GameAlchemist. Collision between AABB and circle. http://goo.gl/7E84Ef.

[13] Randy Gaul. How to create a custom 2d physics engine : The basics and impulse resolution. http://goo.gl/G0gdWU.

[14] Wikipedia. Verlet integration. https://goo.gl/Q6giDh.

[15] Wikipedia. The runge-kutta method. https://goo.gl/UHTD8S.