École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
3e année
2011 - 2012

Rapport de Projet

I.A. pour le jeu Snake et implémentation sur la PolyClock V1.3

Encadrant
Carl ESSWEIN
carl.esswein@univ-tours.fr

Étudiants
Thibault DREVON
thibault.drevon@etu.univ-tours.fr
Kévin GUÉGAN
kevin.guegan@etu.univ-tours.fr

Université François-Rabelais, Tours

DI3 2011 - 2012

Version du 7 juin 2012

Table des matières

Table des figures

Introduction

Le projet qui nous a été confié est un projet visant à améliorer un I.A. du jeu populaire connu sous le nom de Snake. En effet, l'I.A. est déjà présent dans la PolyClock mais comporte des bugs et ne possède pas un algorithme très évolué. Il fait donc suite a plusieurs projets antérieurs qui vont de la conception de la PolyClock à la création d'applications comme le Snake.

La PolyClock est une boite composée d'un microcontroleur, Arduino, et d'un écran à Leds controlé par celui-ci. Elle a été conçue lors d'un projet antérieur.

Nous avons donc pour but de corriger les bugs que l'on rencontre et de trouver un algorithme pour l'I.A. qui soit assez satisfaisant. Il nous a fallu évidemment par la suite implémenter cet algorithme dans la PolyClock en transcrivant celui-ci en langage C.

Présentation de l'existant

2.1 Description du jeu Snake

Snake, de l'anglais signifiant "serpent", est un jeu vidéo populaire créé au milieu des années 1970, disponible de par sa simplicité sur l'ensemble des plate-formes de jeu existantes. Il s'est de nouveau fait connaître dans les années 1990 avec l'émergence du nouveau support de jeu qu'est le téléphone portable. En effet, nous avons pu le découvrir sous le fameux nokia, qui a dès lors relancé la mode du Snake.

Le joueur contrôle une longue et fine créature semblable à un serpent, qui doit slalomer entre les bords de l'écran et les obstacles qui parsèment le niveau. Pour gagner chacun des niveaux, le joueur doit faire manger à son serpent un certain nombre de fruits, allongeant à chaque fois la taille de celui-ci.

Alors que le serpent avance sans s'arrêter, le joueur ne peut que lui indiquer une direction à suivre (en haut, en bas, à gauche, à droite) afin d'éviter que la tête du serpent ne touche les murs ou son propre corps. Le niveau de difficulté est contrôlé par l'aspect du niveau (simple ou labyrinthique), le nombre de pastilles à manger, l'allongement du serpent et sa vitesse.

Dans le cadre de notre projet, le Snake est sous son aspect simple, c'est-à-dire que les seuls obstacles sont les bords de l'écran, caractérisés par des murs. Le Snake n'a qu'un seul fruit à manger, et la vitesse est choisie par nos soins. Bien évidemment, celle-ci n'influe pas sur la difficulté, car les calculs de l'I.A. dépassent largement la vitesse du serpent. Le plateau de jeu a, quant à lui, une taille de 46x14 (murs non compris).

2.2 Description de la PolyClock

La PolyClock est à la base une horloge conçu lors d'un projet d'électronique. L'horloge est composée d'une carte avec un kit Arduino et d'un écran à Leds. Les Leds fonctionnent comme une matrice, controlée par le micro processeur qui commande l'affichage. Sa composition est la suivante :

- 2 matrices à LED vertes 24x16
- Arduino Duemilanove 328
- Horloge temps réel DS1307
- Quartz 32.768 kHz

– Boutons poussoir
– Câble USB

Lors du projet de fabrication, un boitier a été conçu car aucun support n'était présent.

La Polyclock s'allume automatiquement lorsqu'elle est branchée en USB. Pour naviguer dans celle-ci, il faut utiliser les boutons suivant.

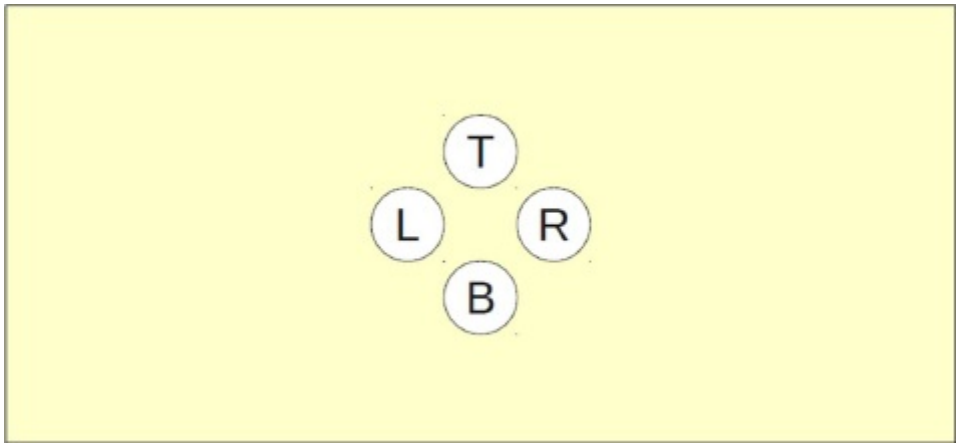

FIGURE 2.1 – Représentation des boutons

Pour passer à l'application suivante, il faut appuyer sur le bouton T.

La PolyClock est composée d'applications d'ambiances et jeux telles que :

FIGURE 2.2 – Photo du Pong

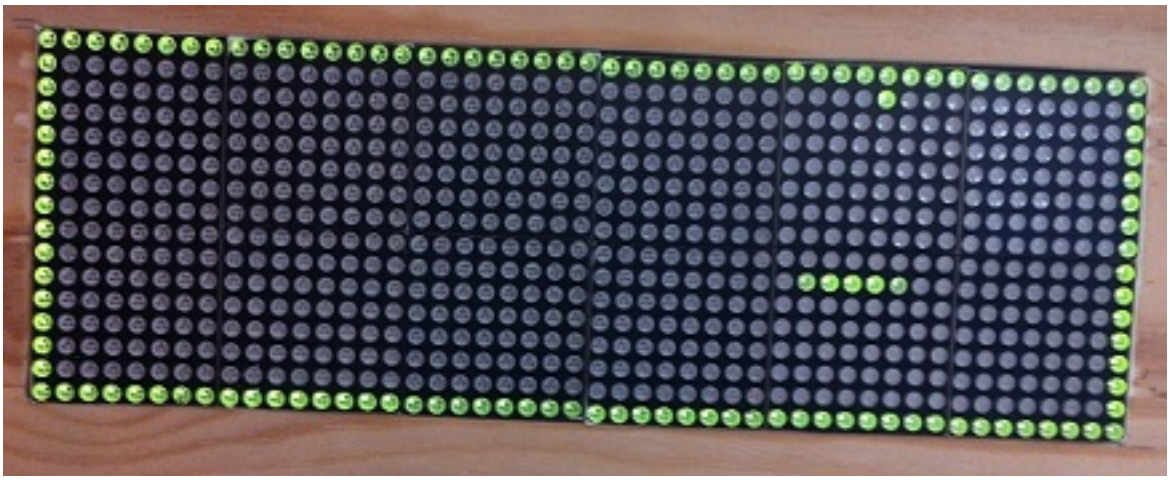

FIGURE 2.3 – Photo du Snake

FIGURE 2.4 – Photo du Rain

FIGURE 2.5 – Photo du Tetris

2.3 Description de l'environnement Arduino

Arduino est un circuit imprimé dont les plans sont publiés en licence libre sur lequel se trouve un microcontrôleur qui peut être programmé pour analyser et produire des signaux électriques, de manière à effectuer des tâches très diverses comme la charge de batteries, la domotique (le contrôle des appareils domestiques - éclairage, chauffage...), le pilotage d'un robot, etc. C'est une plateforme basée sur une interface entrée/sortie simple et sur un environnement de développement utilisant la technique du Processing/Wiring.

Arduino peut être utilisé pour construire des objets interactifs indépendants (prototypage rapide), ou bien peut être connecté à un ordinateur pour communiquer avec ses logiciels (ex. : Macromedia Flash, Processing, Max/MSP, Pure Data, SuperCollider).

2.3.1 Matériel

Un module Arduino est généralement construit autour d'un microcontrôleur Atmel AVR (ATmega328 ou ATmega2560 pour les versions récentes, ATmega168 ou ATmega8 pour les plus anciennes), et de composants complémentaires qui facilitent la programmation et l'interfaçage avec d'autres circuits. Chaque module possède au moins un régulateur linéaire 5V et un oscillateur à quartz 16 MHz (ou un résonateur céramique dans certains modèles). Le microcontrôleur est pré-programmé avec un bootloader de façon à ce qu'un programmateur dédié ne soit pas nécessaire.

Les modules sont programmés au travers d'une connexion série RS-232, mais les connexions permettant cette programmation diffèrent selon les modèles. Les premiers Arduino possédaient un port série, puis l'USB est apparu sur les modèles Diecimila, tandis que certains modules destinés à une utilisation portable se sont affranchis de l'interface de programmation, relocalisée sur un module USB-série dédié (sous forme de carte ou de câble).

2.3.2 Logiciel

Le logiciel de programmation des modules Arduino est une application Java, libre et multi-plateformes, servant d'éditeur de code et de compilateur, et qui peut transférer le firmware et le programme au travers de la liaison série (RS-232, Bluetooth ou USB selon le module). Il est également possible de se passer de l'interface Arduino, et de compiler les programmes via l'interface en ligne de commande.

Le langage de programmation utilisé est le C++, compilé avec avr-g++ 2, et lié à la bibliothèque de développement Arduino, permettant l'utilisation de la carte

et de ses entrées/sorties. La mise en place de ce langage standard rend aisé le développement de programmes sur les plates-formes Arduino, à toute personne maitrisant le C ou le C++.

Plusieurs logiciels et matériels compatibles Arduino (bien que non officiels) ont été produits commercialement par d'autres fabricants :

- Les kits Freeduino Bare Bones Board et Really Bare Bones Board compatibles Arduino, fabriqués et vendus par Modern Device Company.
- Le Freeduino Through-Hole, compatible Arduino, module évitant l'utilisation de composants CMS et destiné à une fabrication personnelle ; fabriqué et vendu par NKC Electronics (sous forme de kit).
- Le Boarduino, un clone peu cher du Diecimila, avec des connecteurs pour une utilisation sur plaque de test.
- Des versions Fundamental Logic en kit ou montées, le MaxSerial Freeduino compatible Arduino Diecimila, interface RS-232, alimentation 3.3V optionnelle sur la carte ; et le mini iDuino pour une utilisation sur plaque de test.

Développement de l'I.A.

3.1 Analyse du jeu

3.1.1 Repérer les points fondamentaux

Pour assurer une bonne conception des structures et des algorithmes nécessaires à notre application, il faut avant tout explorer, comprendre et analyser les mécanismes du jeu. Quelques parties permettent de distinguer les éléments suivants :

L'entité principale du jeu est le serpent. Il possède une queue de taille variable mais qui ne peut que grandir et non rétrécir. A chaque mouvement, le serpent choisit l'une des quatre directions cardinales et avance d'une case dans cette direction, c'est-à-dire que sa tête arrive sur la case en question et que la case sur laquelle se trouvait le bout de sa queue devient vide. Les autres parties de son corps ne bougent pas.

Les mouvements du serpent impliquent que son univers n'est pas continu mais discret. Il ne peut bouger que d'une case à une autre et ne peut pas rester entre deux cases.

L'environnement dans lequel le serpent évolue, le plateau, est un rectangle de largeur et hauteur définies dont le serpent ne peut s'échapper. Hauteur et largeur utiles sont réduites chacune de 2 puisque les murs font parties du plateau mais sont inaccessibles pour le serpent.

Il existe aussi un fruit/cible, disposé de façon aléatoire sur le plateau, mais qui ne peut apparaitre ni sur les murs ni sur un morceau de la queue du serpent. Lorsque le serpent mange un fruit, un autre est immédiatement généré et la taille du serpent augmente. Cette augmentation se déroule d'une façon précise : au coup suivant, la tête avance mais le bout de la queue n'est pas retiré.

Enfin, la partie s'arrête lorsque le serpent entre en contact avec un morceau de sa queue ou avec un mur. Il faut donc conserver les positions de tous les points de la queue du serpent.

L'IA, puisque son objectif est de faire survivre et croître le serpent le plus possible, devra trouver le coup le plus adapté à chaque situation et modifier la direction de ce dernier en conséquence. Le détail de cette importante tâche est fait plus loin.

Note : Dans sa version originale, le Snake est un jeu " temps réel ". Le serpent n'attend pas une instruction du joueur pour avancer. En fait, le joueur dispose d'un laps de temps bien défini pour changer la direction courante du serpent. Passé ce délai, le serpent avance d'une case dans la dernière direction choisie. Notre version du Snake n'appliquera pas ce mode de fonctionnement cependant.

3.1.2 Regrouper les fonctionnalités au sein d'entités

Compte tenu de la description faite ci-dessus, on peut proposer un regroupement logique des fonctionnalités du jeu sous quelques entités :

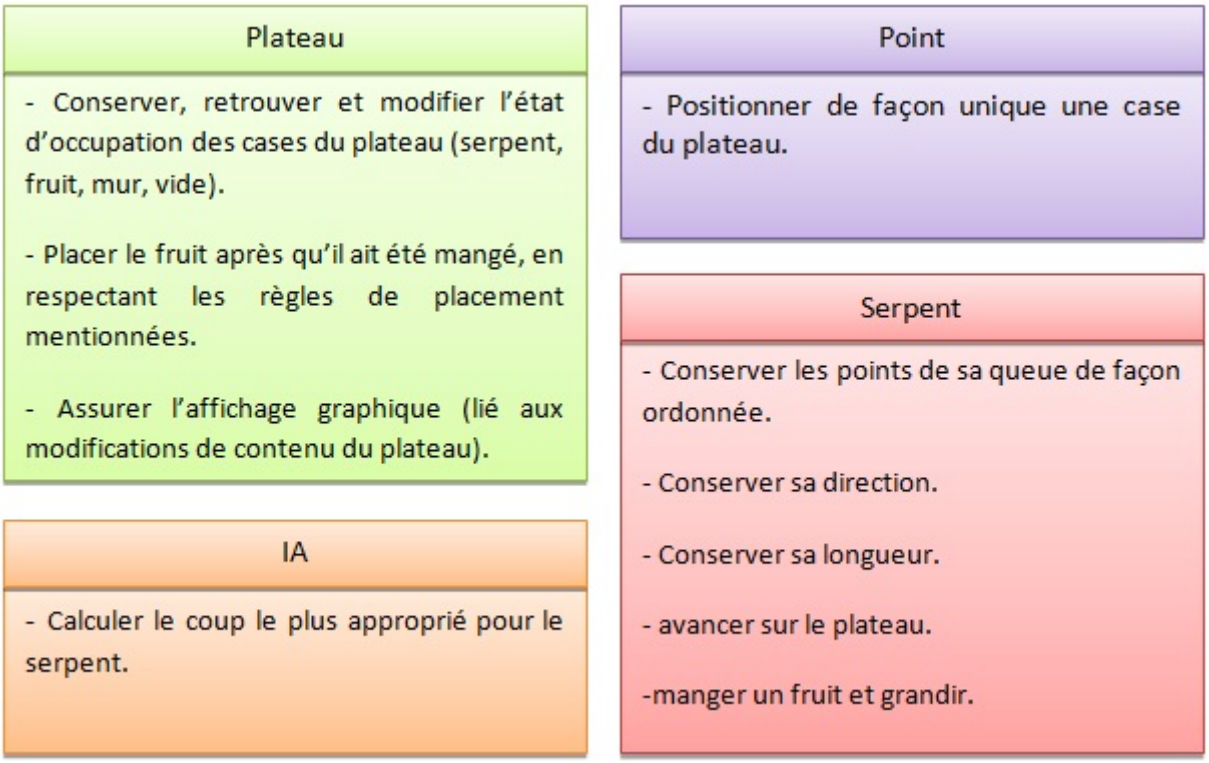

FIGURE 3.1 – Les entités

Ces entités représentent des structures de données et les fonctionnalités (méthodes) qui leur sont associées. Elles ne seront pas les seules structures à être implémentées, mais constituent les objets propres à notre application. Il est ensuite nécessaire de détailler algorithmiquement parlant ces structures. Le détail se fera en deux parties :

1. Les structures et algorithmes relatifs au déroulement du jeu d'une part
2. L'IA seule d'autre part.

3.2 Structures et algorithmes du jeu

3.2.1 Le Point

Le Point est la structure la plus simple du programme et est relativement courante dans le monde informatique. Précisons simplement par ailleurs le système de représentation choisi : On se placera dans le système de représentation classique des écrans et des matrices, c'est-à-dire l'ordonnée vers le bas et l'abscisse vers la droite. Le premier point (coin haut gauche) aura pour coordonnées (0, 0) et le dernier (coin bas droite) (BOARD_WIDTH - 1, BOARD_HEIGHT - 1) avec BOARD_WIDTH et BOARD_HEIGHT respectivement largeur et hauteur du plateau à définir. La structure choisie est donc :

```
Point : record
        X : entier non signé ;
        Y : entier non signé ;
End ;
```

FIGURE 3.2 – Structure du point

Le point ne dispose en outre d'aucune primitive particulière associée.

3.2.2 Le plateau

La fonction principale du plateau est de conserver l'état de ses cases pour d'une part informer le serpent (déterminer la fin de la partie) et l'IA (calcul basé sur l'état du plateau) et d'autre part pour pouvoir placer le fruit en respectant les règles de placement. Le contenu du tableau ne peut prendre que des valeurs finies et exclusives (pas de contenu multiple). La largeur du plateau est définie par la constante BOARD_WIDTH. La hauteur du plateau est définie par la constante BOARD_HEIGHT. La structure choisie est donc la suivante :

```
Board : record
        Matrix: tableau [0..BOARD_WIDTH – 1][0..BOARD_HEIGHT -1] d'entiers;
        {Les valeurs autorisées sont :
        BRD_EMPTY ( 0 ), indique que la case est vide.
        BRD_SNAKE ( 1 ), indique que la case contient un morceau de serpent.
        BRD_WALL ( 2 ), indique que la case contient un mur.
        BRD_FRUIT ( 3 ), indique que la case contient un fruit.}
        Fruit : Point ; {pour localiser rapidement le fruit}
        emptyCasesCount : entier non signé ;
        {Indique le nombre de cases vides restantes dans le plateau, pour le placement du fruit}
End ;
```

FIGURE 3.3 – Structure du plateau

Pour chaque primitive, on donnera une description plus ou moins brève en fonction de sa complexité ainsi que ses spécifications.

Board_getContent

Cette fonction récupère sous forme d'un entier le contenu d'une case du plateau spécifiée par un Point.

Module: Board_getContent(aBoard, aPoint)

Entrée :
aBoard : Pointeur sur Board
aPoint : Point

Préconditions :
aBoard est correctement initialisé.

Sortie :
BoardContent : entier ; Valeur qui indique le contenu de la case du plateau au point aPoint

Postconditions :
BoardContent = (matrix[aPoint.x][aPoint.y] si aPoint possèdes des coordonnées valides, BRD_WALL sinon).

FIGURE 3.4 – La fonction Board_getContent

Board_setContent

Primitive permettant de modifier le contenu d'une case du plateau. L'affichage à l'écran est également modifié en conséquence.

Module : Board_setContent(aBoard, aPoint, value)

Entrée :
aBoard : Pointeur sur Board
aPoint : Point
value : entier {une des valeurs valides pour le contenu du plateau}

Préconditions :
aBoard est correctement initialisé.
aPoint possède des coordonnées valides.

Sortie :
Aucune

Postconditions :
Matrice de aBoard mise à jour.
emptyCasesCount de aBoard mise à jour.
LED correspondante sur la matrice à LED allumée ou éteinte en fonction de value.

FIGURE 3.5 – La fonction Board_setContent

Board_setFruit

Méthode permettant de placer le fruit aléatoirement sur le plateau en évitant de le placer sur le serpent ou sur un mur. L'algorithme est grossièrement le suivant :

- ALEAT ← (un entier aleatoire) modulo emptyCasesCount
- De (1, 1) à (BOARD_WIDTH - 2, BOARD_HEIGHT - 2), si la case est vide, décrémenter ALEAT
- Si ALEAT = -1, placer le fruit à la position courante (avec Board_setContent).

Module : Board_setFruit(aBoard)

Entrée :
aBoard : Pointeur sur Board

Préconditions :
aBoard est correctement initialisé.

Sortie :
Aucune

Postconditions :
Matrice de aBoard mise à jour.
Fruit de aBoard mis à jour.
Nouvelle LED allumée correspondant à la position du fruit.

FIGURE 3.6 – La fonction Board_setFruit

Board_Init

Cette fonction a pour but d'initialiser le plateau (Board). C'est-à-dire, dans l'ordre :

- Mettre à BRD_EMPTY (0) le contenu de chacune des cases de la matrice.
- Effacer l'écran à LED.
- Placer les cases de mur (affichage + stockage dans la matrice)
- Initialiser emptyCasesCount à (BOARD_WIDTH - 1)×(BOARD_WIDTH - 1)
- Placer le premier fruit à l'aide de la méthode Board_setFruit.

Module : Board_init(aBoard)

Entrée :
aBoard : Pointeur sur Board
Préconditions :
Aucune
Sortie :
Aucune
Postconditions :
aBoard est initialisé et prêt à être utilisé.

FIGURE 3.7 – La fonction Board_Init

3.2.3 Le serpent

La structure du serpent doit pouvoir contenir toutes les informations relatives à son état (vie ou mort, a mangé un fruit, direction) ainsi que celles permettant de le localiser facilement sur le plateau (position de la tête et référence au plateau auquel il appartient). Ces informations sont facilement stockables et ne nécessitent aucune attention particulière.

Le problème se pose pour ce qui est de la queue du serpent et de la conservation des positions de chacun de ses morceaux. Ces points doivent être conservés de façon ordonnée de telle manière qu'on supprime toujours à chaque coup le dernier élément de sa queue. Les précédents détenteurs de la PolyClock travaillant sur le Snake avaient choisi de conserver les Points du serpent dans un tableau, le premier élément de ce tableau étant le bout de la queue et le dernier la tête. A chaque mouvement, ils effaçaient le bout de queue, décalaient chaque élément n vers la position n - 1 et ajoutaient la tête.

Ce choix est discutable dans la mesure où sa complexité temporelle croit proportionnellement avec la taille du serpent. En partant du principe que notre IA peut s'avérer gourmande en ressource, autant limiter la complexité là où cela est

possible. Une solution intéressante est l'implémentation d'une file. Puisqu'à chaque mouvement, nous supprimons le bout de la queue et ajoutons une nouvelle tête sans toucher au reste du corps, cette structure de données semble assez appropriée.

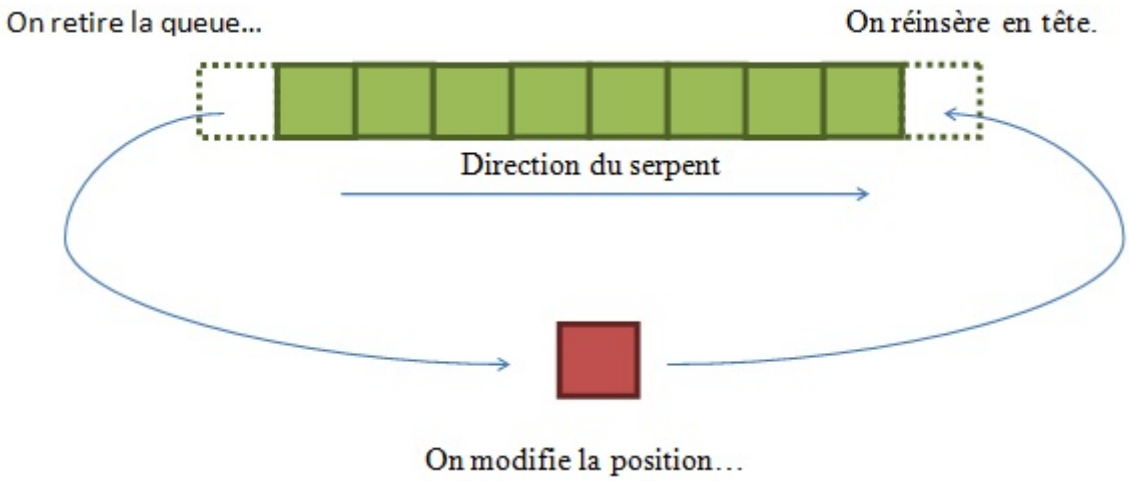

FIGURE 3.8 – Le schémas du serpent

La complexité du mouvement du serpent est alors en O(1) (constante), ce qui s'avère plus intéressant. La structure de données pour une file (ici appelée **Queue**) ainsi que les primitives associées, étant donné qu'elles sont disponibles dans tout bon cours d'algorithmique et sur Wikipédia, ne seront pas détaillées ici.

```
Snake : record
        position: Point; {la position de la tête du serpent}
        tail : Queue ; {les points successifs du corps du serpent}
        direction : entier non signé
        {Les valeurs autorisées sont :
        DIRECT_UP( 0 ), haut.
        DIRECT_RIGHT( 1 ), droite.
        DIRECT_DOWN( 2 ), bas.
        DIRECT_LEFT( 3 ), gauche.}
        environment : pointeur sur Board ; {l'environnement dans lequel évolue le serpent}
        hasEaten : booléen ; {VRAI si le serpent vient de manger}
        hasDied : booléen ; {VRAI si le serpent est mort}
End ;
```

FIGURE 3.9 – La structure du serpent

Pour chaque primitive, on donnera une description plus ou moins brève en fonction de sa complexité ainsi que ses spécifications.

Snake_init

Initialise une structure de type Snake, en plaçant notamment dans le plateau les deux premiers points de son corps, en initialisant la file des points de sa queue et en mettant à FAUX les booléens de sa structure. Par défaut le serpent se dirige vers la droite.

Module : Snake_init(aSnake, aPoint, board)
Entrée :
aSnake : pointeur sur le Snake à initialiser.
aPoint : position initiale de la tête du serpent.
aBoard : Pointeur sur le Board sur lequel évolue le serpent
Préconditions :
Aucune
Sortie :
Aucune
Postconditions :
aSnake est initialisé et prêt à être utilisé.

FIGURE 3.10 – La fonction Snake_init

Snake_finalize

Libère la mémoire réservée pour tail (la Queue qui gère les points du corps du serpent).

Module : Snake_finalize (aSnake)
Entrée :
aSnake : pointeur sur le Snake à finaliser.
Préconditions :
aSnake est correctement initialisé.
Sortie :
Aucune
Postconditions :
La mémoire allouée pour la gestion du corps du serpent est libérée.

FIGURE 3.11 – La fonction Snake_finalize

Snake_eat

Utilisée pour notifier que le serpent a mangé un fruit. Modifie son attribut hasEaten et place un nouveau fruit sur le plateau.

Module : Snake_eat(aSnake)
Entrée :
aSnake : pointeur sur le Snake qui vient de manger un fruit.
Préconditions :
aSnake est correctement initialisé.
Sortie :
Aucune
Postconditions :
Attribut hasEaten du serpent mis à TRUE. Nouveau fruit placé.

FIGURE 3.12 – La fonction Snake_eat

Snake_Move

L'algorithme principal du serpent. Il se découpe comme suit :
– Modifier la position du serpent en fonction de la direction courante
– Tester la nouvelle position → Si mort, arrêter
– Placer la nouvelle tête sur le plateau.
– Si le serpent n'a pas mangé, retirer le dernier élément de la queue
– Mettre hasEaten à FAUX.

Module : Snake_Move(aSnake)
Entrée :
aSnake : pointeur sur le Snake à déplacer.
Préconditions :
aSnake est correctement initialisé.
Sortie :
Aucune
Postconditions :
Soit le serpent est mort soit sa position a été mise à jour et son corps déplacé.

FIGURE 3.13 – La fonction Snake_Move

Une fois tous ces algorithmes correctement mis en place, le jeu est fonctionnel mais le serpent, sans guide, s'obstine à heurter le mur de droite. Il est donc temps de s'intéresser à l'IA.

3.3 L'Intelligence Artificielle

3.3.1 Repérer les objectifs et les tactiques

Comment concevoir une IA efficace pour le Snake ? Qu'est-ce d'abord qu'une IA efficace ? La réponse peut paraitre simple : atteindre une taille optimale. Autrement dit, manger le plus de fruits. Certes oui, mais cela implique-t-il de foncer immédiatement vers le fruit, au risque de mourir au coup suivant ? Certes non. Il faut également préserver la vie du serpent. Et ces deux objectifs se contredisent parfois.

La pratique montre également qu'un certain nombre de bonnes habitudes tactiques permettent d'obtenir un score plus élevé. Par exemple, quand le serpent commence à devenir grand, on a naturellement tendance à ne pas laisser de trou en gardant le serpent collé à lui-même, de telle sorte qu'il optimise l'espace qui lui reste.

Une étude approfondie permet de dégager les objectifs (primordiaux ou secondaires) suivants :

- Survivre immédiatement (ne pas choisir un coup fatal si un autre coup est possible)
- Chercher le fruit
- Eviter les murs, ils constituent une issue de secours en cas de danger
- Eviter de rentrer dans des " cul-de-sac ", c'est-à-dire des zones trop petites pour le serpent dans lesquelles l'issue fatale est possible voire inévitable
- Quand le serpent est en danger du fait de sa taille ou de l'espace restreint dans lequel il se trouve, favoriser les coups qui le gardent collé à lui-même.

Ces objectifs dégagés, il reste à leur donner une priorité relative. On ne pourrait en effet pas concevoir qu'éviter les murs soit aussi important que de survivre immédiatement. De plus le contexte peut influer sur les priorités : si atteindre le fruit entraine une mort quasi immédiate, il est certain que ce n'est pas la meilleure décision à prendre.

Le choix qui est fait ensuite peut être discuté. C'est en tout cas en suivant la hiérarchie de priorité suivante que notre version de l'IA choisit la direction à suivre.

Il convient d'apporter quelques précisions au logigramme. On définit par cul-de-sac une zone close (entourée de mur ou de morceaux de serpent) dans lequel le serpent s'engage et dont la surface est inférieure à sa taille.Plus une direction potentielle monte haut dans le logigramme, plus elle est intéressante. L'algorithme conserve les caractéristiques du meilleur coup trouvé jusqu'alors et le compare avec le coup étudié sur le moment. Si le coup étudié présente de meilleurs résultats, il remplace le meilleur coup et ainsi de suite jusqu'à ce que tous les coups possibles aient été étudiés.

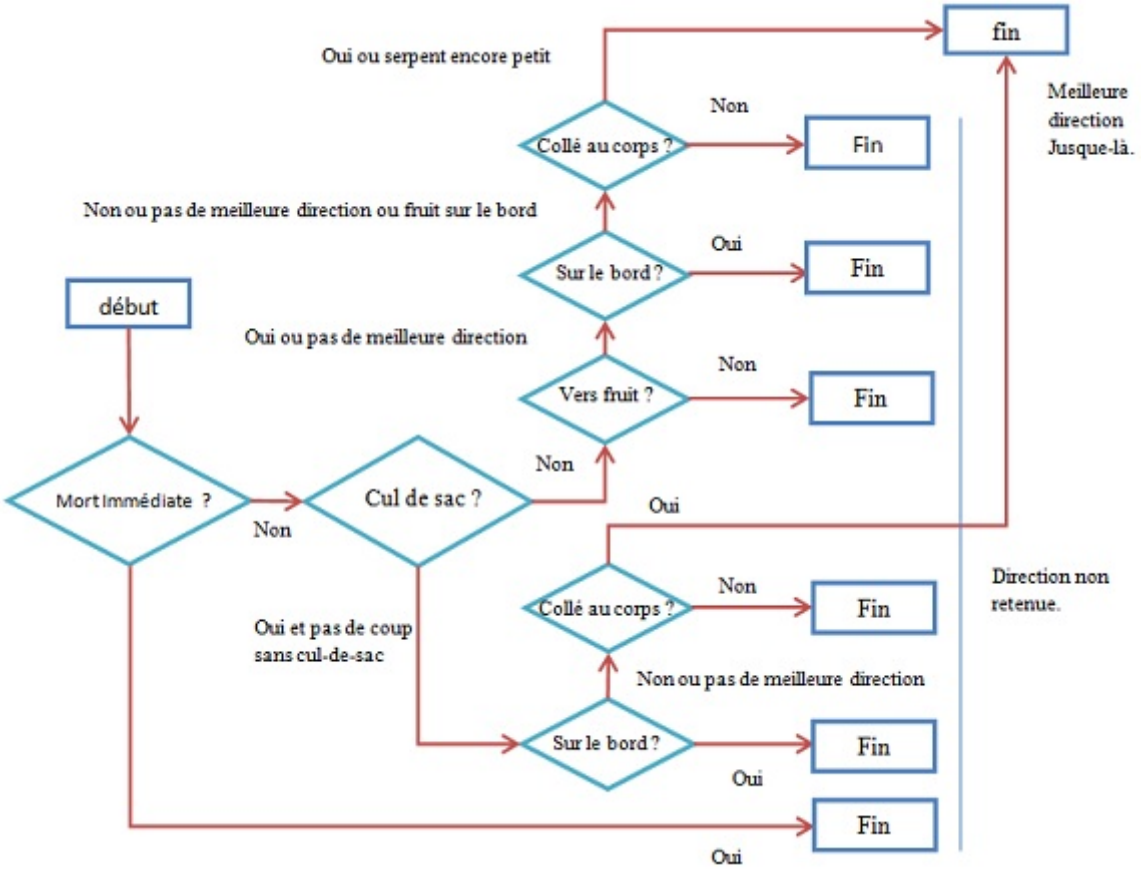

FIGURE 3.14 – Logigramme de l'I.A.

3.3.2 Détacher et définir les primitives

Une fois cette analyse effectuée, la route à suivre est plus claire. De plus, les différents losanges du logigramme suggèrent d'eux-mêmes la marche à suivre pour découper de façon méthodique cet imposant algorithme en petites primitives essentielles détaillées ci-après.

Le test de mort immédiate et celui pour savoir si le coup est sur le bord sont tellement simples qu'ils ne nécessitent pas de fonctions individuelles (on pourra utiliser une MACRO C pour le test de bordure)

SnakeIA_isStuckToTail

L'objectif de cette méthode est de déterminer si le coup fait que le serpent reste collé à lui-même (pour ne pas gaspiller d'espace).

Module : SnakeIA_isStuckToTail(aBoard, aPoint)
Entrée :
aBoard: tableau sur lequel évolue le snake et qui garde le contenu des cases.
aPoint : le coup à tester.
Préconditions :
aBoardest correctement initialisé.
Sortie :
RESULT : booléen
Postconditions :
RESULT = VRAI si le coup fait que le serpent est collé à sa queue.

FIGURE 3.15 – La fonction Snake_IA_isTargetDirection

Snake_IA_isTargetDirection

Cette méthode doit déterminer si le coup rapproche ou non du fruit.

Module : Snake_IA_isTargetDirection(directVectorToTarget,aDirection)
Entrée :
directVectorToTarget: Point, en fait il s'agit de la distance « à vol d'oiseau » de la position courante du serpent (avant le coup) au fruit.
aDirection : entier non signé. Code direction (voir la définition de Snake) pour effectuer le calcul.
Préconditions :
0 <= aDirection<= 3
Sortie :
RESULT : booléen
Postconditions :
RESULT = VRAI si un coup dans cette direction rapproche du fruit.

FIGURE 3.16 – La fonction Snake_IA_isTargetDirection

Snake_IA_computeSurface

Cette méthode est censée calculer la surface disponible lorsqu'un serpent s'engage dans une " zone ". Prenons un exemple :

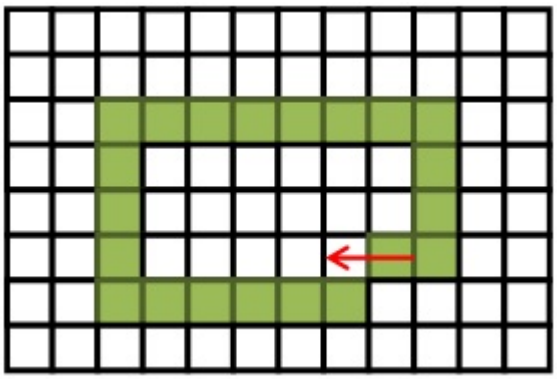

FIGURE 3.17 – Exemple de situation d'engouffrement

Si le serpent fait un pas de plus dans cette direction, il s'engagera dans une zone close potentiellement dangereuse. L'estimation du risque se fait en calculant le nombre de cases à l'intérieur de la surface et en comparant à la longueur du serpent (A noter que cette méthode ne fait que le calcul, pas la comparaison).

Pour effectuer son travail, la dite fonction dispose d'une file, (une Queue) ainsi que d'une matrice booléenne de dimension identique à celle du plateau de jeu. La file sert à mettre en attente les cases à explorer. Quant à la matrice, elle garde la trace des cases déjà explorées.

On initialise l'algorithme en mettant le compteur de surface à 0, en insérant le coup à tester dans la file et en mettant à vrai le booléen de la matrice situé aux coordonnées du coup puis :

Tant que la file n'est pas vide, on retire son premier élément(une case). On teste le contenu de ses quatre voisins. Si le contenu est vide **et** si la case n'est pas marquée comme explorée dans la matrice, alors on la marque explorée, on incrémente le compteur de surface et on ajoute la case à la fin de la file.

Pour éviter une recherche trop longue et inutile, on arrêtera de chercher si le compteur de surface atteint une valeur maximale, fixée à la longueur courante du serpent.

Grâce à cette méthode, l'IA sera en mesure d'éviter que le serpent ne s'engouffre dans des zones trop dangereuses.

Module : Snake_IA_computeSurface(aBoard, testPoint, maxComputation)
Entrée :
aBoard: pointeur sur le plateau de jeu du serpent.
testPoint : Point; le coup à tester, point de départ du calcul de surface.
maxComputation : entier non signé ; valeur de calcul maximale. Si elle est atteinte le calcul s'arrête.
Préconditions :
aBoard est correctement initialisé.
testPoint possède des coordonnées valides.
Sortie :
SURFACE : entier non signé correspondant la surface disponible dans la zone du coup.
Postconditions :
SURFACE = MIN(surface disponible dans la zone, maxComputation)

FIGURE 3.18 – La fonction Snake_IA_computeSurface

Snake_IA_getNextMove

Pour finir, la méthode première qui appelle toutes les autres. Elle retourne la direction dans laquelle le serpent devrait aller, selon l'IA.

Module : Snake_IA_getNextMove(aSnake)
Entrée :
aSnake : pointeur sur le Snake à examiner.
Préconditions :
aSnake est correctement initialisé.
Sortie :
DIRECTION : entier non signé correspondant à un code indiquant une direction à suivre.
Postconditions :
DIRECTION est le coup le plus approprié à jouer par le serpent du point de vue de l'IA.

FIGURE 3.19 – La fonction Snake_IA_getNextMove

3.3.3 La seule façon de faire ?

La méthode choisie pour déterminer le prochain coup du serpent est une méthode " à la volée ", au coup par coup. On aurait pu aborder le problème différemment en essayant par exemple de déterminer un chemin complet et sûr vers le fruit puis de l'appliquer sans recalculer. Et recommencer tant que faire se peut.

Le problème de cette technique (outre sa complexité) est son temps de calcul et sa consommation mémoire. Le microprocesseur reste peu performant et le Snake est tout de même censé se rapprocher d'un jeu temps réel. Néanmoins, il est possible, voire probable, que cette méthode décroche de meilleurs résultats si elle est correctement conçue.

3.4 Les limites du microprocesseur

3.4.1 Le problème

Après avoir conçu et implémenté les algorithmes, on aurait pu croire que la partie était terminée, mais il n'en fut rien. La PolyClock refusait obstinément de ne serait-ce qu'afficher les murs du plateau, au tout début du jeu. Un mystère d'abord percutant, d'autant que la même implémentation sous Windows (avec CodeBlocks, mode console) fonctionnait parfaitement. Des recherches dans la documentation d'Arduino ont fini par apporter l'explication de cette avarie : La mémoire vive.

Sur le microprocesseur Arduino Duemilanove 328, il n'est possible de disposer que de 2ko de mémoire vive au maximum. Si l'on retire la mémoire utilisée par le clJur et celle réservée par les autres fonctionnalités de la PolyClock, on tombe aux alentours du ko, ce qui il faut le reconnaitre est un peu restrictif.

Après quelques essais, il est aussi apparu que les méthodes d'allocation mémoire (malloc et free) codées par Arduino ne semblaient pas performantes et diminuaient encore la capacité.

3.4.2 De l'optimisation

La première chose à faire dans ce cas est de réduire toutes les variables de taille superflue. Par exemple, on utilise souvent des entiers i ou j comme compteur de boucle. Or dans le cas des algorithmes du Snake, ces compteurs ne montaient jamais au-dessus de 255. Un entier est codé sur 4 octets et un char (min 0, max 255) sur 1 octet seulement. En appliquant le même principe aux structures et aux variables locales, on économise déjà quelques dizaines d'octets.

Mais devant l'ampleur du problème, de petits ajustements comme ceux cités précédemment sont vains. Il faut voir plus grand. On pouvait par ailleurs remarquer que la rapidité du processeur était bien plus que suffisante pour faire tourner le programme en dépit de sa faible RAM.

Il s'est donc avéré que, pour une fois, l'objectif de l'optimisation n'était pas de diminuer la complexité temporelle, mais de diminuer la complexité spatiale.Il fallait repenser les structures et les algorithmes de telle sorte qu'ils soient moins gourmands en mémoire même si cela impliquait une perte en vitesse. Et de plus, il fallait trouver un moyen de stocker plus efficacement les données existantes.

3.5 L'optimisation

3.5.1 Les matrices booléennes

L'algorithme de calcul de surface de l'IA nécessite une matrice booléenne pour son calcul. Celle-ci était d'abord implémentée en C comme ceci :

```
Char matrix[BOARD_WIDTH - 2][BOARD_HEIGHT - 2] {on éliminait d'office les murs}
```

FIGURE 3.20 – Définition de la Matrice

Soit un tableau de 46 × 14 octets soit 644 octets. Nul doute que la PolyClock ne pouvait le supporter. Il est évident que cette consommation de mémoire était

un gâchis. En effet, un booléen n'a besoin que d'un bit pour être codé. Ainsi la consommation réelle est de (46 × 14 / 8) = 81 octets.

Heureusement, le langage C offre des possibilités de calcul, de modification et de comparaison au bit à bit, décrites ci-dessous. Pour autant, les variables doivent avoir une taille multiple d'un octet.

Opérateur	Dénomination	Effet	Syntaxe	Résultat
&	ET bit-à-bit	Retourne 1 si les deux bits de même poids sont à 1	9&12(1001&1100)	8(1000)
|	OU bit-à-bit	Retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux)	9|12(1001|1100)	13(1101)
^	OU bit-à-bit exclusif	Retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux)	9^12(1001^1100)	5(0101)
«	Décalage à gauche	Décale les bits vers la gauche (multiplie par 2 à chaque décalage). Les zéros qui sortent à gauche sont perdus, tandis que des zéros sont insérés à droite	6«1(110«1)	12(1100)
»	Décalage à droite avec conservation du signe	Décale les bits vers la droite (divise par 2 à chaque décalage). Les zéros qui sortent à droite sont perdus, tandis que le bit non nul de poids plus fort est recopié à gauche	6»1(0110»1)	3(0011)

Ce qui amène à l'astuce suivante : la hauteur du plateau de jeu est de 14 cases (16 - 2 pour les murs). Or un entier court, ou short int, est codé sur 16 bits. En utilisant des entiers courts comme des colonnes de notre plateau et en faisant des opérations bit à bit, il est possible de mimer une matrice booléenne.

Bien sûr, l'accès aux éléments de la matrice s'en trouve un peu complexifié, mais l'économie réalisée à l'aide de cette technique n'est pas négligeable.

Suffisamment intéressant en tout cas pour considérer d'adapter également la matrice de la structure Board. Certes, ce n'est pas à l'origine un tableau de booléens. Rappelons qu'une case peut contenir un mur, un bout de serpent, un fruit ou être vide. Mais :

– Les murs sont extérieurs au plateau de jeu et de position fixe. Il suffit de dire que tout élément en dehors du plateau utile (hors du rectangle ((1, 1), (1, Y_MAX - 1), (X_MAX - 1, 1) (X_MAX - 1, Y_MAX -1))), est un mur. Résultat, une légère diminution des algorithmes d'ajout et de consultation et une économie car on ne conserve plus en mémoire le bandeau du mur.

0	0	0	0	0
1	1	1	0	1
1	0	0	1	0
0	0	0	0	0
1	1	1	0	1
1	0	0	1	0
0	0	0	0	0
1	1	1	0	1
1	0	0	1	0
0	0	0	0	0
1	1	1	0	1
1	0	0	1	0
0	0	0	0	0
1	1	1	0	1
1	0	0	1	0
0	1	1	0	1

FIGURE 3.21 – Simulation d'une matrice booléenne

– Il existe déjà un attribut de la structure Board qui conserve la position du fruit et il ne peut y avoir qu'un fruit au même moment.

On se retrouve donc avec une matrice booléenne : la case contient un bout de serpent ou est vide. On peut appliquer le changement de structure.

Economie totale réalisée : 1164 octets.

3.5.2 Les champs de bits

Dans le cadre de l'optimisation, le C dispose d'une autre petite merveille : les champs de bits.Un champ de bits permet de regrouper plusieurs variables de taille inférieure à un octet au sein du même octet. La syntaxe de déclaration est très proche de celle d'une structure et on accède d'ailleurs aux sous-variables d'un champ de bits à l'aide d'un " . " comme pour les structures. La seule différence est qu'après la déclaration d'un attribut du champ de bit, on doit spécifier sa taille en bit après un " : ".

FIGURE 3.22 – Structure envisagée

Une possibilité comme cela devait être exploitée. Le plus gros point noir en mémoire dans l'application à ce moment était la conservation des points du corps du snake. Puisque l'utilisation des fonctions d'allocations et de libération de la mémoire étaient inutilisables, il a fallu abandonner la structure de file et il n'était pas possible de faire autrement que de réserver un tableau de points pour chaque serpent.

Seulement, si l'on considère qu'un bout de la queue du serpent peut être modélisé à l'aide de deux char (x et y) et d'un entier court (index dans le tableau du point suivant), on se retrouve avec 4 octets par point. L'algorithme ne dépassant jamais le score de 300 points sous Windows et en gardant une marge de sécurité, il fallait disposer d'une zone de 350 points soit 1300 octets. Hélas, des tests montraient que la PolyClock ne déroulait correctement le programme qu'avec une taille réservée de 140 points au mieux.

Une longue réflexion a par bonheur fait surgir l'idée suivante : La répartition des points du corps du serpent sur le plateau n'est pas purement aléatoire. En fait, chaque point est situé immédiatement à la gauche, à la droite, au dessus ou au dessous du précédent. 4 possibilités seulement. En connaissant le point de départ, il suffit donc de 2 bits pour trouver les coordonnées du point suivant. Pour retrouver dans le tableau du corps du snake l'index du point suivant, le tableau devant comporter environ 350 entrées, 9 bits suffisent.

Avec 11 bits + le point initial, il est possible de coder un point de la queue du serpent. Mais 11 n'est pas un multiple de 8, d'ailleurs 11 est premier. En regroupant 8 points donc, on obtient la structure suivante de 88 bits soit 11 octets qui permet de représenter correctement 8 bouts de serpent.

Pour chacun des huits points contenus dans la structure, posX_rel est la direction suivie entre le coup précédent et ce coup et posX_nextPos est l'index dans le tableau du Snake du prochain point.

La structure du Snake a donc évolué conjointement à cette mesure. Les booléens hasDied et hasEaten ont aussi été retirés car ils se sont révélés inutiles à l'utilisation.

```
PackedPoints: record
        pos0_rel : entier non signé sur 2 bits;
        pos1_rel : entiernon signé sur 2 bits;
        pos2_rel : entiernon signé sur 2 bits;
        pos3_rel : entiernon signé sur 2 bits;
        pos4_rel : entiernon signé sur 2 bits;
        pos5_rel : entiernon signé sur 2 bits;
        pos6_rel : entiernon signé sur 2 bits;
        pos7_rel : entiernon signé sur 2 bits;
        pos0_nextPos : entiernon signé sur 9 bits;
        pos1_nextPos : entiernon signé sur 9 bits;
        pos2_nextPos : entiernon signé sur 9 bits;
        pos3_nextPos : entiernon signé sur 9 bits;
        pos4_nextPos: entiernon signé sur 9 bits;
        pos5_nextPos : entiernon signé sur 9 bits;
        pos6_nextPos : entiernon signé sur 9 bits;
        pos7_nextPos : entiernon signé sur 9 bits;
End ;
```

FIGURE 3.23 – Définition d'un ensemble de 8 points

```
Snake : record
        position: Point; {la position de la tête du serpent}
        tailSpace : tableau 1..184 de PackedPoints ; {les points successifs du corps du serpent}
        direction : entier non signé
        {Les valeurs autorisées sont :
        DIRECT_UP ( 0 ), haut.
        DIRECT_RIGHT ( 1 ), droite.
        DIRECT_DOWN( 2 ), bas.
        DIRECT_LEFT( 3 ), gauche.}
        environment : pointeur sur Board ; {l'environnement dans lequel évolue le serpent}
        headIndex : entier non signé {l'index de la tête du serpent dans tailSpace}
        queueIndex : entier non signé {l'index du bout de la queue du serpent dans tailSpace}
        queuePosition : Point ; {la position du bout de la queue su serpent}
        length : entier non signé {la longueur de la queue du serpent}
End ;
```

FIGURE 3.24 – Nouvelle structure du serpent

De manière similaire mais sans commune mesure, les booléens nécessaires à l'algorithme principal de l'IA, SnakeIA_getNextMove, ont été regroupés au sein d'un champ de bits appelé IAFlags_t.

Grâce à ces modifications, il a été possible de monter le nombre de points réservés pour le Snake à 400. Un chiffre bien plus que suffisant.

3.6 Pas de Tron, encore que...

Les problèmes liés à l'optimisation du temps ont englouti la majeure partie du temps dédié à ce projet et il n'a pas été possible de prendre vraiment le temps de réfléchir à un mode deux serpents à la façon " Tron ".

Néanmoins, la conception du programme autorise tout à fait que deux serpents évoluent en même temps sur le même plateau. Ils chercheront le fruit pour grandir, s'éviteront tant que possible. Mais ils ne chercheront pas à piéger leur adversaire. Le résultat visuel reste satisfaisant.

Penser juste à diminuer la taille réservée pour la queue du serpent car deux serpents en lice, c'est deux fois plus de mémoire réservée.

Conclusion

Dans le cadre de ce projet informatique, nous avons réalisé une modification et optimisation d'IA avec portage sur la PolyClock pour le jeu SNAKE.

La maitrise de la PolyClock a représenté le premier challenge, avec la complexité de son infrastructure ainsi que les outils de développements. L'étude du jeu, de ses failles actuelles, ainsi que la recherche d'optimisation, ont constitué le second challenge de ce projet.

Au final, nous avons intégré de nouveaux algorithmes dans l'IA, optimisant notamment les déplacements et la gestion de la mémoire, rationnalisant les structures ainsi que le jeu lui-même.

Afin d'améliorer encore le coté ludique et visuel de la plateforme, nous aurions pu développer un second mode jeu appelé 'Tron' ou double serpent mais la priorité allant à l'optimisation mémoire du jeu en mode mono serpent, le temps a manqué mais ceci pourra peut être faire l'objet d'un autre projet.

Ce projet nous a permis d'appliquer les connaissances qui nous ont été inculquées au cours de notre année à PolyTech, ainsi que de nous initier encore plus concrètement au travail en équipe, aux techniques de gestion de projet, de suivi de planning. Nous avons été confrontés à de nombreux problèmes et avons pu trouver des solutions rationnelles et efficaces afin de les résoudre.

Enfin, ce projet aura été l'occasion de découvrir et d'utiliser des outils inconnus jusqu'alors et de mesurer notre capacité d'adaptation et d'apprentissage.

Annexe : pistes pour le Tron

Bien que nous n'ayons pas eu le temps d'aborder l'algorithme de l'IA spécifique au TRON, nous tenons à laisser derrière nous certaines pistes à explorer :

En conservant le concept d'objectifs primaires et secondaires pour construire l'algorithme, on pourra ajouter dans la liste existante celui de " bloquer l'adversaire ". Cet objectif devrait se trouver à une priorité supérieure à celle de l'objectif de recherche du fruit mais inférieure à celle de la survie immédiate.

On pourra aussi trier les coups possibles en fonctions des éléments suivants :

ATTAQUE : pour chaque coup, calculer la surface de la zone adverse (voir la méthode computeSurface). Si celle-ci est réduite (valeur à déterminer) et que le coup n'est pas trop dangereux, le jouer (on cherche à bloquer le serpent adverse).

DEFENSE : pour chaque coup, s'il est joué, s'il existe un coup de l'adversaire qui crée une zone close trop petite, le rejeter.

Exemple :

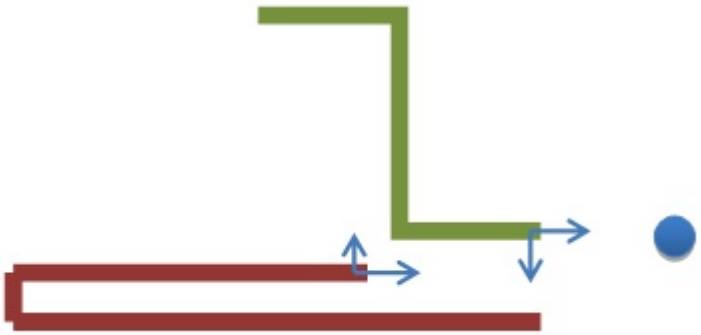

FIGURE A.1 – Exemple pour le Tron

– *Côté défense, serpent rouge* :

Si le serpent rouge joue à droite et que le serpent vert joue en bas, le serpent rouge sera bloqué. Il ne faut donc pas choisir le coup à droite.

– *Côté Attaque, serpent vert* :

Si le serpent vert joue en bas, il crée une zone réduite pour le serpent rouge **et** il peut repartir à droite ensuite pour s'échapper. C'est mieux que d'aller prendre le fruit.

I.A. pour le jeu Snake et implémentation sur la PolyClock V1.3

Département Informatique
3e année
2011 - 2012

Rapport de Projet

Résumé : Ce projet a pour objet la conception et l'implémentation d'une intelligence artificielle pour le Snake, célèbre jeu d'arcade, sur la Polyclock, horloge aux capacités multiples équipée d'un processeur Arduino. Après une première partie sur l'analyse, nous détaillons les fonctionnalités implémentées ainsi que l'optimisation de code nécessaire. Enfin, des pistes pour de prochains travaux sont soulevées.

Mots clefs : Serpent, PolyClock, I.A., Arduino, écran à Leds

Abstract: This project aims at conceiving and implementing an artificial intelligence for the famous arcade game Snake on the Polyclock, an Arduino processor powered clock that has many functionalities. After a first part about analysis, details about the programmed features and the required code optimization are given. At last, some trails to deepen the work are brought out.

Keywords: Snake, PolyClock, Arduino, screen of Leds

Encadrant
Carl ESSWEIN
carl.esswein@univ-tours.fr

Étudiants
Thibault DREVON
thibault.drevon@etu.univ-tours.fr
Kévin GUÉGAN
kevin.guegan@etu.univ-tours.fr

Université François-Rabelais, Tours

DI3 2011 - 2012