Université de Franche-Comté

Projet d'Initiation à l'Ingénierie
L1 - CMI

Analyse des Jeux

BOITEUX Elouan
BENALI Samia
GEHANT Aurélie
LITAMPHA Benoît

UNIVERSITÉ DE FRANCHE-COMTÉ

CMI Figure
CURSUS MASTER EN INGÉNIERIE

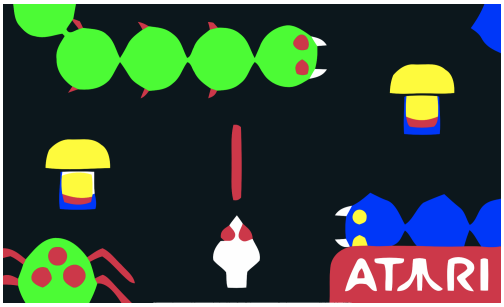
2023/2024

Table des matières

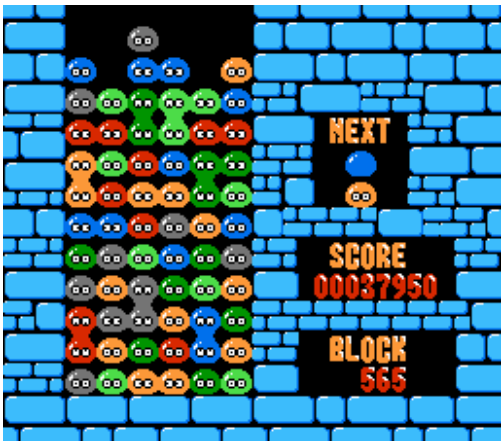
Introduction

Les jeux vidéo ont été, depuis leur création, un moyen fascinant d'explorer des mondes virtuels, de défier nos compétences et de nous immerger dans des expériences interactives. Dans ce rapport, nous allons plonger dans deux titres emblématiques : Centipède, un classique de l'arcade, et Puyo Puyo, un puzzle game coloré et stratégique. Notre objectif est de décortiquer et d'analyser les mécanismes de ces jeux afin de pouvoir, par la suite, les recréer à notre manière.

Since their beginnings, video games have been a fascinating way to explore virtual worlds, challenge our skills and immerse ourselves in interactive experiences. In this report, we take a look at two iconic titles: Centipede, an arcade classic, and Puyo Puyo, a colourful and strategic puzzle game. Our aim is to analyse the mechanics of these games so that we can recreate them in our own way.



Centipède



Puyo Puyo

Chapitre 1

Centipède

Introduction

Centipède est un jeu vidéo développé par Atari sorti sur les bornes d'arcade en 1981. C'est le premier jeu développé par une femme : Dona Bailey. C'est un shoot'em up fixe : un jeu où l'on incarne un personnage armé face à une horde d'ennemis. Ici, le joueur est un petit nain armé qui doit protéger son jardin contre les puces, les scorpions, les araignées et bien évidemment un mille-pattes géant (qui se décomposera en plusieurs mille-pattes lorsqu'il sera touché par les tirs du nain).

Ce jeu peut être retrouvé sur de multiples consoles de jeu comme l'Atari2600, la GameBoy ou plus récemment, la PS4.

1.1 Règles du jeu

Dans le jeu Centipède, le but est de tuer le mille-pattes (qui se scindera en deux lorsqu'il sera touché par un tir du nain) de chaque niveau pour pouvoir passer au niveau supérieur. Le personnage incarné est un nain, il devra éviter les entités (mille-pattes, scorpions, puces, araignées) pour éviter de perdre des vies. Le nain a trois vies par niveau et, s'il perd les trois vies, la partie est terminée et le score est enregistré. Si le joueur relance la partie, il redémarrera au niveau 1 avec un score à 0. Enfin, le jeu est conçu pour ne jamais se terminer (car il y a une infinité de niveaux), avec une difficulté croissante au fur et à mesure de la progression. Ainsi, Centipède offre une expérience de jeu infinie.

1.2 Description des modèles de données

1.2.1 Les structures générales

```
1
2   Structure Position
3       reel x
4       reel y
5   finStructure
6
```

Cette structure permet de donner des informations sur les positions `x` et `y` des entités. C'est une structure générale car on l'utilise dans d'autres structures (celles des entités). Les déplacements se font verticalement et horizontalement, d'où la nécessité d'utiliser deux attributs (`x` et `y`).

```
1
2       Structure Dimension
3           reel x
4           reel y
5       finStructure
6
```

Cette structure permet de donner des informations sur les dimensions `x` et `y` des entités. C'est une structure générale car on l'utilise dans d'autres structures (celles des entités). On définit grâce à l'attribut `x` la largeur et l'attribut `y` la longueur de l'entité.

1.2.2 Les variables et constantes seules

Les autres variables seules utilisées sont :

```
1
2       constante entier WIDTH
3       constante entier HEIGHT
4       constante entier [3] COULEUR_FOND
5       constante entier DROITE
6       constante entier GAUCHE
7       constante entier HAUT
8       constante entier BAS
9       constante entier STOP
10      booleen enJeu
11      booleen enPause
12      entier {} points
13      constante entier nbMaxChampi
14      Champignon [] listeChampi
15      entier score
16      reel dt
17
```

constante entier WIDTH Cette constante représente la largeur en pixels de la fenêtre.

constante entier HEIGHT Cette constante représente la hauteur en pixels de la fenêtre.

constante entier [3] COULEUR_FOND Cette constante représente la couleur du fond de la fenêtre, elle est représentée sous la forme [R,V,B]. Ces valeurs des codes couleurs RVB font référence à l'intensité des couleurs primaires : rouge, vert et bleu. L'intensité varie de 0 à 255 où 255 est l'intensité maximale.

constante entier DROITE Cette constante est égale à 1 et permet d'indiquer la direction en x pour déplacer le nain. Elle permet d'avoir un code plus lisible.

constante entier GAUCHE Cette constante est égale à -1 et permet d'indiquer la direction en x pour déplacer le nain. Elle permet d'avoir un code plus lisible.

constante entier HAUT Cette constante est égale à -1 et permet d'indiquer la direction en y pour déplacer le nain. Elle permet d'avoir un code plus lisible.

constante entier BAS Cette constante est égale à 1 et permet d'indiquer la direction en y pour déplacer le nain. Elle permet d'avoir un code plus lisible.

constante entier STOP Cette constante est égale à 0 et permet d'arrêter le nain. Elle permet d'avoir un code plus lisible.

booleen enJeu : Cette variable est initialement définie sur `False`. Son état basculera à `True` dès le lancement de la partie, permettant ainsi de naviguer dans la section appropriée de la boucle de jeu. Une fois que le joueur aura épuisé ses trois vies, elle reviendra à `False`, signalant ainsi la fin de la partie et le retour à l'écran d'accueil.

booleen enPause : Cette variable est initialement définie sur `False`. Elle basculera à `True` dès que l'utilisateur appuiera sur le bouton dédié. Cela mettra le jeu en pause, arrêtant tout mouvement des entités jusqu'à ce que la pause soit levée.

entier {} points Cette variable est un dictionnaire, où chaque clé correspond à une action spécifique et sa valeur représente le nombre de points attribués au joueur pour cette action. Par exemple, si le joueur touche le corps du mille-pattes, il gagne 10 points, comme illustré ci-dessous :

```
pointGagne = points["corpsCentipede"]
```

Voici comment est initialisé le dictionnaire :

```
points = {
    "corpsCentipede": 10,
    "teteCentipede": 100,
    "champignon": 1,
    "puce": 200,
    "scorpion": 1000,
    "araignee": {
        "petiteDistance": 300,
        "moyenneDistance": 600,
        "grandeDistance": 900
    }
}
```

entier score Cette variable contient le nombre total de points que le joueur a gagné depuis le début de la partie

constante entier nbMaxChampi Cette constante est définie comme étant égale au nombre maximal de champignons pouvant être présents sur la zone de jeu. Si le nombre de champignons sur la carte est inférieur à `nbMaxChampi`, alors des puces descendent de l'écran, laissant derrière elles des champignons, jusqu'à ce que le nombre de champignons atteigne
`nbMaxChampi`.

Champignon[ ] listeChampi Cette variable de type liste (taille variable) contient tous les champignons actuellement présents sur la zone de jeu. Elle nous permet d'accéder facilement au nombre de champignons présents sur la carte pour savoir si des puces doivent rajouter des champignons sur la zone de jeu :

```
nbMaxChampi > taille(listeChampi)
```

reel dt Cette variable représente le nombre réel qui indique le temps écoulé depuis le dernier passage dans la boucle de jeu. Il est exprimé en millisecondes. À chaque itération de la boucle, cette valeur est mise à jour pour indiquer le temps écoulé depuis le précédent cycle de la boucle. La variable `dt` permettra par la suite de faire progresser le nain sur la carte avec la formule :

$$d = v \times dt$$

1.2.3 Nain

Le personnage que l'on joue est un nain de jardin qui se déplace dans l'aire de jeu. On doit donc connaître sa position grâce à l'attribut `pos` et sa direction grâce au vecteur `dir` (x,y). `x` et `y` pourront prendre les valeurs de -1, 0 ou 1 pour savoir si le personnage doit monter, descendre, aller vers la gauche, vers la droite ou s'il doit s'arrêter. Le nain peut se déplacer en diagonale si son vecteur `dir` n'a aucune valeur à 0. Le nain se déplace à une vitesse constante donnée par l'attribut `vitesse` (ou `vitesse` $\times \sqrt{2}$ si il se déplace en diagonale).

De plus, l'aire de déplacement du nain n'est pas l'aire de jeu complète : il a une limite sur la hauteur, donnée par l'attribut `hauteur`, et une limite latérale, donnée par l'attribut `largeur`. Ces deux variables sont sous la forme d'un tableau où les extremums sont indiqués. Enfin, le nain a 3 vies, la fin du jeu ne sera déclenchée qu'une fois les 3 vies perdues. L'attribut `nbVie` représente ses 3 vies.

```
1
2     Structure Nain
3         entier[2] hauteur
4         entier[2] largeur
5         Position pos
6         Dimension dim
7         entier[2] dir
8         entier nbVie
9         entier vitesse
10    finStructure
11
```

1.2.4 Champignon

Ce type agrégé représente les champignons du jeu. Ce sont des obstacles pour le déplacement des mille-pattes, du nain et parfois de l'araignée. Chaque champignon a sa position et sa dimension données respectivement par les attributs `pos` et `dim`.

Quand le nain tire sur les champignons, il lui enlève une vie. Chaque champignon a 4 vies que l'on défini par l'attribut de type entier `nbVie`. Lorsqu'un scorpion touche un champignon, il l'empoisonne. Quand l'attribut `empoisonne` passe à `True`, le `champignon` est empoisonné et change de couleur.

```
1
2     Structure Champignon
3         Position pos
4         Dimension dim
5         entier nbVie
6         booleen empoisonne
7     finStructure
8
```

1.2.5 Bille Centipede

La structure `BilleCentipede` permet de définir chaque petite partie du corps du mille-pattes géant. Si une bille du mille-pattes est une tête alors le booléen `estTete` est à `True`. Si la tête du mille-pattes touche un champignon empoisonné, alors celle-ci est également empoisonnée. (Le booléen `estEmpoisonne` passe à True). Enfin, les attributs `pos` et `dim` sont respectivement la position et la taille de chaque bille.

```
1
2     Structure BilleCentipede
3         booleen estEmpoisonne
4         booleen estTete
5         Position pos
6         Dimension dim
7     finStructure
8
```

1.2.6 Corps Centipede

Le mille-pattes est initialement composé d'un corps de 11 billes et d'une tête qui se réduira au fil du temps. En effet, lorsque le joueur lui tire dessus, la bille touchée disparaît. On a donc 3 attributs : une liste `liste_bille` où les billes du corps sont stockées, un entier `longueur` qui représente la longueur du corps sans compter la tête et un entier qui représente la vitesse du mille-pattes contenue dans l'attribut `vitesse`.

```
    Structure CorpsCentipede
        list liste_bille
        entier longueur = taille(liste_bille)
        entier vitesse
    finStructure
```

1.2.7 Araignée

Cette structure permet de définir les attributs de l'araignée. L'`araignee` est une entité qui apparaît régulièrement et se déplace en diagonale et en ligne droite. L'Araignée ne peut se déplacer que dans une zone de jeu définie par `hauteur` et `largeur` (zone de jeu plus grande que la fenêtre pour qu'elle puisse sortir de la fenêtre). Elle ne peut pas faire demi-tour. (En effet si elle est rentrée par la droite de l'aire de jeu, elle ne pourra pas sortir par la droite.). Sa vitesse est définie par l'attribut `vitesse`. Les attributs `pos` et `dim` sont respectivement la position et la dimension de l'araignée.

```
    Structure Araignee
        Position pos
        Dimension dim
        entier[2] dir
        entier [2] hauteur
        entier [2] largeur
    finStructure
```

1.2.8 Puce

Cette structure permet de définir les attributs de la Puce. La `puce` est une entité faisant des apparitions exceptionnelles avec une probabilité égale à `probabilite` qui est un entier. Elle chute verticalement du haut de la scène avec une vitesse définie par l'attribut `vitesse`. Elle dépose des champignons sur son chemin (voir `Arrivée des puces`). Les attributs `pos` et `dim` sont respectivement la position et la dimension de la puce.

```
    Structure Puce
        Position pos
        Dimension dim
        entier vitesse
        entier probabilite
    finStructure
```

1.2.9 Scorpion

Cette structure permet de définir les attributs du Scorpion. Le `Scorpion` est une entité faisant des apparitions exceptionnelles, passant horizontalement de la gauche vers la droite dans la scène de jeu avec une vitesse définie par l'attribut `vitesse`. Il empoisonne les champignons sur son chemin. Les attributs `pos` et `dim` sont respectivement la position et la taille du scorpion.

```

    Structure Scorpion
        Position pos
        Dimension dim
        entier vitesse
    finStructure

```

1.2.10 Tir

Cette structure va nous permettre de gérer les tirs de notre nain. Elle contient un attribut `vitesse` qui définit une vitesse de déplacement, un booléen `enCollision` permettant de savoir s'il a touché ou non une entité, une position `pos` et une dimension `dim`. Les tirs partent de la position du nain au moment où il tire et ne peuvent pas être déviés. ils vont en ligne droite (verticalement) jusqu'à toucher une entité où jusqu'à sortir de l'aire de jeu.

```

    Structure Tir
        Position pos
        Dimension dim
        entier vitesse
        booleen enCollision
    finStructure

```

1.3 Évolution des entités

1.3.1 Entités dirigées par le joueur

Le nain de jardin est dirigé, soit par la souris, soit par les flèches du clavier. L'évolution est différente selon le moyen de déplacement. Le joueur tire en effectuant un clic gauche ou en appuyant sur la barre d'espace.

Déplacement du nain avec la souris

Premièrement, si le joueur joue avec la souris, le nain va suivre le pointeur de la souris et les coordonnées de l'attribut `pos` dépendra des coordonnées de la souris et/ou des bordures de l'aire de déplacement du nain. Pour cela il faut donc récupérer les coordonnées de la souris.

On se retrouve avec différents cas de figures :

— Si le pointeur de la souris est à l'intérieur de l'aire de déplacement alors aucun problème, les coordonnées du nain sont celles du pointeur de la souris.

— Si le pointeur de la souris dépasse la limite de hauteur de déplacement, alors la position en `x` du nain sera égale à la limite maximale de l'aire de jeu sur la hauteur. La position en `y` du nain sera la même que celle de la souris.

— Si le pointeur de la souris dépasse les limites latérales de déplacement, alors la position en `x` sera la même que celle de la souris et la position en `y` du nain sera égale à la limite maximale de l'aire de jeu sur les largeurs.

— Si le pointeur de la souris dépasse à la fois la limite de hauteur et les limites latérales alors les positions en `x` et en `y` du nain seront égales aux limites de l'aire de jeu sur la hauteur et sur la largeur où se trouve le nain.

```
1
2     si typeEvenement = souris, alors
3         // Recuperer les coordonnees de la souris
4         souris.pos.x = obtenirPositionSourisX()
5         souris.pos.y = obtenirPositionSourisY()
6
7         // Mise a jour du x et du y
8         nain.pos.x = souris.pos.x
9         nain.pos.y = souris.pos.y
10
```

Déplacement du nain avec les flèches du clavier

Lorsque le joueur souhaite déplacer le nain en utilisant les touches fléchées, chaque pression de touches détermine une nouvelle direction de déplacement pour le nain.

Par exemple, si le joueur appuie sur la flèche droite, la coordonnée horizontale de la direction du nain `nain.dir[0]` est mise à 1, tandis que pour la flèche gauche, elle est mise à -1. De manière similaire, les flèches haut et bas affectent la direction verticale `nain.dir[1]`. Cette configuration permet de définir la direction dans laquelle le nain se déplacera lorsqu'il avancera.

Gestion des directions données par le joueur

Pour gérer les déplacements du joueur, nous avons développé un petit code qui permet au nain d'être déplacé soit avec les flèches du clavier, soit avec la souris. Dans une même partie, il est possible d'utiliser indifféremment les flèches ou la souris. Le code repose sur la comparaison des positions de la souris à deux instants différents : t et t-1. Si la position de la souris à l'instant t est différente de celle à l'instant t-1, cela signifie que la souris a été utilisée depuis le dernier tour de boucle. Dans ce cas, le nain prend la position de la souris. Sinon, c'est-à-dire si la souris n'a pas bougé, le nain continue à être déplacé avec les flèches du clavier. Le code associe ensuite les coordonnées du nain à celles de la souris, ajustées en fonction de la taille du nain pour un positionnement précis. Si la souris n'est pas utilisée, le nain est déplacé selon les instructions des flèches.

```

// curseur [0] = position a l'instant t
// curseur [1] = position a l'instant t-1
si (curseur[0].x != curseur[1].x) OU
  (curseur[0].y != curseur[1].y), alors
    nain.pos.x = curseur[0].x-nain.dim.x/2
    nain.pos.y = curseur[0].y-nain.dim.y/2

sinon    // Utilisation des fleches
    nain.mouv(dt*nain.dir.x*nain.vitesse,
              dt*nain.dir.y*nain.vitesse)

```

Pour faire avancer le nain dans la direction définie par le joueur, on utilise un mécanisme basé sur le temps. Lorsque le jeu exécute une boucle, on mesure l'intervalle de temps entre chaque itération de la boucle et on le note dans la variable `dt`. On combine ensuite la vitesse du nain (`nain.vitesse`), la direction de déplacement horizontale ou verticale (`nain.dir[0]` ou `nain.dir[1]`), et l'intervalle de temps (`dt`) pour calculer le déplacement du nain lors de cette itération.

Par exemple, si le nain se déplace vers la droite alors `nain.dir[0]` sera égale à 1 et son déplacement horizontal pendant un intervalle de temps `dt` serait :

```
nain.pos.x = nain.pos.x + (nain.vitesse * 1 * dt)
```

On obtient :

```

  nain.dir[0] = 0
  nain.dir[1] = 0

  si typeEvenement = clavier, alors
      // Direction x
      si evenement = flecheGauche, alors // Direction vers
 la gauche
          nain.dir[0] = GAUCHE
      finSi

```

```
11
12          si evenement = flecheDroite, alors // Direction vers
    la droite
13              nain.dir[0] = DROITE
14          finSi
15
16          // Direction y
17          si evenement = flecheHaut, alors // Direction vers le
     haut
18              nain.dir[1] = HAUT
19          finSi
20          si evenement = flecheBas, alors // Direction vers la
    droite
21              nain.dir[1] = BAS
22          finSi
23
24          // Mise a jour du x
25          si nain.dir[0] == DROITE, alors
26              nain.pos.x = nain.pos.x + (dt * nain.dir[0] *
    nain.vitesse)
27          si nain.dir[0] == GAUCHE, alors
28              nain.pos.x = nain.pos.x + (dt * nain.dir[0] *
    nain.vitesse)
29
30          // Mise a jour du y
31          si nain.dir[1] == BAS, alors
32              nain.pos.y = nain.pos.y + (dt * nain.dir[1] *
    nain.vitesse)
33          si nain.dir[1] == HAUT, alors
34              nain.pos.y = nain.pos.y + (dt * nain.dir[1] *
    nain.vitesse)
35
```

Interdiction de sortir de la zone de jeu du nain

Le code suivant restreint les déplacements du personnage (représenté par le `nain`) à l'intérieur des limites de la fenêtre de jeu. Il utilise les fonctions `max` et `min` pour s'assurer que les coordonnées `x` et `y` du personnage restent respectivement dans les limites horizontales et verticales de la fenêtre. Ainsi, le personnage ne peut pas sortir de la zone de jeu définie par la fenêtre. Cette action sera exécutée à chaque tour de boucle du jeu pour contrôler les déplacements du personnage.

```
1
2       nain.pos.x = min( max(nain.pos.x, 0),
3                         largeurFenetre - self.dim.x)
4
5       nain.pos.y = min( max(nain.pos.y, 0),
6                         hauteurFenetre - nain.dim.y)
7
```

1.3.2 Entités ennemies au joueur

Déplacement du mille-pattes

Le mille pattes a une manière de se déplacer : Il se déplace horizontalement vers la droite ou vers la gauche. Dès qu'il touche un obstacle ou un bord de la zone de jeu, il descend verticalement d'un cran (taille d'une bille du mille-pattes) et repart dans l'autre sens. Une fois arrivé en bas de la zone de jeu, il remonte de la même manière, ainsi de suite jusqu'à ce qu'il soit complètement détruit. Sa vitesse dépend du niveau dans lequel le joueur se trouve.

Déplacement des araignées

L'araignée se déplace soit en diagonale, soit verticalement, en direction du côté opposé à celui où elle est apparue. Elle ne peut pas faire demi-tour et repartir du côté par où elle est arrivée. Ses déplacements sont aléatoires mais doivent respecter ces conditions. Sa vitesse dépend du niveau dans lequel le joueur se trouve.

Arrivée des puces en fonction du nombre de champignons

Comme nous l'avons expliqué dans la structure `Puce`, les puces descendent du haut de l'aire de jeu en déposant des champignons. Les puces déposeront des champignons si et seulement si le nombre maximal de champignons n'est pas atteint. Ce nombre maximal est stocké dans la variable indépendante `nbMaxChampi`. Lorsque le nombre de puces n'est pas maximal, leur arrivée est aléatoire : elles n'apparaissent pas forcément immédiatement. De plus, plus le niveau est élevé, plus l'apparition de puces est probable.

La boucle de notre jeu va s'exécuter environ 60 fois par seconde et on veut que la probabilité que les puces arrivent dans la zone de jeu soit proportionnelle au nombre de tours de boucle.

Par exemple au niveau 1, on veut que les puces arrivent environ toutes les 20 secondes. On va donc calculer de manière proportionnelle le nombres de chances qu'une puce arrive (ici, $20 \times 60 = 1200$).

Dans notre exemple, il y a donc 1 chance sur 1200 (soit toutes les 20 secondes) qu'une puce arrive lorsque le nombre `nbMaxChampi` n'est pas atteint.
Lorsque la condition `(entier)(hasard()*puce.probabilte) == 0)` est vraie, on va déclencher la fonction `arriverPuce()`.

```
1
2   si (taille(listeChampi) < nbMaxChampi), alors
3       si ( (entier)(hasard()*puce.probabilte) == 0), alors
4           arriverPuce()
5       finSi
6   finSi
7
```

1.3.3 Gestion des collisions

Gestion des collisions entre le nain, les entités ennemies et les obstacles

Pour gérer les collisions avec toutes les entités nous utilisons une fonction `detecteCollision()` qui détecte si le nain est en collision avec une entité. Si oui, elle permet de renvoyer un identifiant qui détermine l'entité avec laquelle le nain est en collision. Une fois cette information transmise, suivant le résultat, plusieurs actions sont possibles :

— Le nain est en collision avec un `Champignon`, on utilise la fonction `replacement()`, qui permet de replacer le nain au bon endroit en fonction des coordonnées du `Champignon` concerné.

— Le nain est en collision avec une autre entité, on utilise la méthode de la structure `Nain`, qui permet de retirer une vie au joueur :

```
1
2     action gestionVie(self, nb)
3         self.nbVie = self.nbVie + nb
4     finAction
5
6     nain.gestionVie(-1)
7
```

Gestion des collisions entres les balles et les autres entités

Le nain peut donc tirer et cela est géré par la structure `Tir` pour atteindre le(s) mille-pattes, les puces, les scorpions, les araignées et les champignons pour engendrer des points et/ou passer au niveau suivant. On va gérer les gains de point grâce au dictionnaire de points : `{} points`.
On va donc ajouter le nombre de points associé à chaque entités que le nain a touché avec ses tirs dans la variable `score`. Pour cela, il faut que le booléen `enCollision` de la structure `Tir` passe à `True` pour pouvoir engendrer des points (et faire disparaître les entités.).

```
1
2     entier fonction updateScore(entier identifiantCollision,
    entier score)
3         entier points
4         chaine objetCollision = idToChaine(
    identifiantCollision)
5         points = points[objetCollision]
6         score = score + points
7         retourner score
8     finFonction
9
```

1.3.4 Évolution des niveaux

A chaque changement de niveaux (indiqué par l'absence de mille-pattes dans la zone de jeu), des éléments vont subir des modifications afin d'accroître la difficulté à chaque niveau : modification du nombre maximal de champignons, des vitesses de déplacements, des couleurs, etc...
On obtient donc : (ici, les valeurs sont des valeurs arbitraires)

```
1
2     nain.vitesse = nain.vitesse + 30
3     corpsCentipede.vitesse = corpsCentipede.vitesse + 30
4     nbMaxChampi = nbMaxChampi + 10
5
```

De plus, le nombre de vie `nbVies` est réinitialisé à 3. Les champignons quant à eux, s'ils sont endommagés (s'ils leur restent des vies mais ne sont pas complètement détruits), récupéreront toutes leurs vies.

```
1
2     nain.gestionVie(3 - nain.nbVie)
3
```

1.4 Configuration initiale

Au lancement du jeu, le nain est au centre de son aire de jeu, face au mille-pattes. Ce dernier est centré sur la plus haute ligne de la zone de jeu. De nombreux champignons (on a le nombre maximal de champignon au niveau 1) sont déjà placés aléatoirement sur la zone. Une araignée est à droite du nain. Rien ne se lance tant que le nain n'a pas effectué une action (se déplacer ou tirer). Le nain a trois vies et le score est à 0. La vitesse de déplacement des ennemis est au plus faible.

Chapitre 2

Puyo Puyo

Introduction

Puyo Puyo est un jeu de puzzle crée en 1991 par Kazunari Yonemitsu de l'entreprise Compile et publié sur MSX2 et Famicom Disk System. Il est maintenant possédé par SEGA qui a sorti une édition du jeu sur les bornes d'arcade en 1992. Il est ainsi devenu le jeu le plus populaire du Japon. De nos jours, ce jeu a été revisité et associé à Tetris pour atteindre plus de public. On peut y jouer sur diverses consoles comme la NES, la GameBoy mais aussi sur la Nintendo Switch, la PS4 et la PS5.

2.1 Description des modèles de données

2.1.1 Les structures générales

```

    Structure Position
        reel x
        reel y
    finStructure

```

Cette structure permet de donner des informations sur les positions `x` et `y` des entités. Cette structure est une structure générale car on l'utilise dans d'autres structures (celles des entités). Les déplacements se font verticalement et horizontalement, d'où la nécessité d'utiliser deux attributs (`x` et `y`).

```

    Structure Dimension
        reel x
        reel y
    finStructure

```

Cette structure permet de donner des informations sur les dimensions `x` et `y` des entités. Cette structure est une structure générale car on l'utilise dans d'autres structures (celles des entités). On définit grâce l'attribut `x` la largeur de l'entité et avec l'attribut `y` la longueur de l'entité.

```
Structure PositionMat
    entier i
    entier j
finStructure
```

Cette structure permet de donner des informations sur la position des entités dans une matrice. Les Puyos sont traités dans une matrice et cela facilite leurs déplacements. On se repère donc sur la matrice avec l'attribut `i` et l'attribut `j` respectivement les lignes et les colonnes de la matrice.

2.1.2 Les variables et constantes seules

Les autres variables seules utilisées sont :

```
constante entier WIDTH
constante entier HEIGHT
constante entier [3] COULEUR_FOND
constante entier [7][3] COULEUR_PUYOS
constante entier DROITE
constante entier GAUCHE
constante entier BAS
booleen enJeu
booleen enPause
Puyo[][] Matrice
CouplePuyos couplepuyos
CouplePuyos couplePuyosSuivant
entier score
entier puyoDetruit
entier vitesseChute
entier couleurDispo
```

constante entier WIDTH Cette constante représente la largeur en pixels de la fenêtre.

constante entier HEIGHT Cette constante représente la hauteur en pixels de la fenêtre.

constante entier [3] COULEUR_FOND Cette constante représente la couleur du fond de la fenêtre, elle est représentée sous la forme [R,V,B]. Ces valeurs des codes couleurs RVB font référence à l'intensité des couleurs primaires : rouge, vert et bleu. L'intensité varie de 0 à 255 où 255 est l'intensité maximale.

constante entier [7][3] COULEUR_PUYOS Cette constante représente une liste contenant toutes les couleurs que les puyos peuvent prendre, une couleur est représentée sous la forme [R,V,B]. Il y a 7 couleurs de puyos possibles dans le jeu. On les répertorie dans cette liste pour les utiliser plus tard dans différentes fonctions.

constante entier DROITE Cette constante est égale à 1. Elle permet d'indiquer que l'on décale sur la droite les valeurs `x`. Elle permet d'avoir un code plus lisible.

constante entier GAUCHE Cette constante est égale à -1. Elle permet d'indiquer que l'on décale sur la gauche les valeurs `x`. Elle permet d'avoir un code plus lisible.

constante entier BAS Cette constante est égale à 1. Elle permet d'indiquer que l'on décale vers le bas les valeurs `y`. Elle permet d'avoir un code plus lisible.

booleen enJeu : Cette variable est initialement définie sur `False`. Son état basculera à `True` dès le lancement de la partie, permettant ainsi de naviguer dans la section appropriée de la boucle de jeu. Une fois que le joueur aura dépassé la ligne du haut, elle reviendra à `False`, signalant ainsi la fin de la partie et le retour à l'écran d'accueil.

booleen enPause : Cette variable est initialement définie sur `False`. Elle basculera à `True` dès que l'utilisateur appuiera sur la touche dédiée. Cela mettra le jeu en pause, arrêtant tout mouvement des entités jusqu'à ce que la pause soit levée.

Puyos [ ][ ] Matrice : Cette variable permet de faciliter l'organisation des puyos en permettant de savoir plus facilement qui sont les puyos voisins. Par défaut le nombre de lignes est égal à 13 et le nombre de colonnes à 6.

CouplePuyos couplePuyos : Cette variable initialement définie à `Null` représente le couple de puyos actuellement en chute.

CouplePuyos couplePuyosSuivant : Il s'agit du prochain couple de puyos qui va chuter.

entier score : Cette variable contient le nombre total de points que le joueur a gagné depuis le début de la partie.

entier puyoDetruit : Cette variable recense le nombre de puyos détruits depuis le début de la partie.

entier vitesseChute : Cette variable, d'abord initialisée à 1, définie la vitesse de chute des puyos. La valeur peut varier que de 1 à 4.

entier couleurDispo : Cette variable, d'abord initialisée à 4, définie le nombre de couleurs disponibles pour la chute. Plus on progresse dans le jeu, plus la valeur augmente. 7 est le maximum.

2.1.3 Puyo

Cette structure permet de définir les puyos. Tout d'abord, les attributs `pos` et `posMat` permettent respectivement de définir sa position sur le plateau de jeu et dans la matrice de puyos. Ensuite, l'attribut `dim` permet de définir la taille du puyo sur le plateau de jeu. Pour finir, l'attribut `couleur` permet de définir la couleur du puyo.

```
1
2   Structure Puyo
3       Position pos
4       PositionMat posMat
5       Dimension dim
6       entier [3] couleur
7   finStructure
8
```

2.1.4 Couples Puyos

Les puyos forment des couples. l'un des deux membres du couple est considéré comme le principal par l'attribut `principal` : c'est autour de lui que gravite l'autre membre du couple, appelé `secondaire`. L'entier `placeSecondaire`, pouvant varier de 0 à 3, permet de définir la place du puyo secondaire, 0 étant en haut, 1 à droite, 2 en bas, 3 à gauche. Enfin, le booléen `estControlable` permet de définir si le joueur a le contrôle sur ses déplacements.

```
1
2   Structure CouplePuyos
3       Puyo principal
4       Puyo secondaire
5       entier placeSecondaire
6       booleen estControlable
7   finStructure
8
```

2.2 Évolution des entités

2.2.1 Entités dirigées par le joueur

Le joueur peut contrôler le couple de puyos uniquement lorsqu'il est en chute. Il peut aller à droite (en appuyant sur la flèche droite), à gauche (en appuyant sur la flèche gauche), descendre plus vite (en appuyant sur la flèche du bas) et faire graviter le puyo secondaire dans le sens anti-horaire (en appuyant sur la touche Q) ou dans le sens horaire (en appuyant sur la touche D). Le déplacement et la gravitation ne sont possible seulement si les positions visées sont comprises dans le tableau et qu'aucun puyo est présent. Si l'un des deux puyos du couple ne peut plus se déplacer vers le bas alors le joueur n'a plus le contrôle du couple.

On doit dans un premier temps récupérer les positions des deux puyos du couple s'il est contrôlable. Pour cela on a :

```
1
2     entier[2] dir = [0,0]
3     si (typeEvenement = clavier ET couplePuyos.estControlable
      ) alors
4         // On recupere la position actuelle du couple
5         entier coupleMatPosX = couplePuyos.principal.posMat.i
6         entier coupleMatPosY = couplePuyos.principal.posMat.j
7         // On recupere la position de gravitation du puyo
      secondaire
8         entier posGravSecond = couplePuyos.placeSecondaire
9
```

On doit, pour pouvoir déplacer les couples, savoir si les emplacements qu'ils visent sont libres. Pour cela on a besoin de :

```
1
2     // Permet de savoir si un couple peut etre place a la
      position [x][y] de la matrice
3     booleen fonction couplePeutEtrePlace(-> CouplePuyos cp,->
      entier x, -> entier y)
4
```

On doit, pour pouvoir faire graviter le puyo secondaire et savoir si l'emplacement qu'il vise est libre. Pour cela on a besoin de :

```
1
2     // Permet de savoir si le puyo secondaire peut graviter
      vers la position x, comprit entre 0 et 3
3     booleen fonction couplePeutGraviter(-> CouplePuyos cp,->
      entier x)
4
```

Ensuite, on gère les déplacements si le joueur presse une touche. On obtient :

```
1
2     // Direction
3     si (evenement = flecheDroite
4         ET couplePeutEtrePlace(couplePuyos,
5             coupleMatPosX + DROITE,coupleMatPosY) alors
6         dir[0] = DROITE
7
8     sinon si (evenement = flecheGauche
9         ET couplePeutEtrePlace(couplePuyos,
10        coupleMatPosX + GAUCHE,coupleMatPosY) alors
11        // Direction vers la gauche
12        dir[0] = GAUCHE
13    finSi
```

```
14
15      si (evenement = flecheBas
16          ET couplePeutEtrePlace(couplePuyos, coupleMatPosX,BAS
    )), alors
17          // Direction vers le bas
18          dir[1] = BAS
19      finSi
20
```

```
1
2       // Deplacement du couple de puyo
3       placeCouplePuyos(couplePuyos,coupleMatPosX +dir[0],
4       coupleMatPosY + dir[1])
5
```

On doit également gérer la gravitation du puyo secondaire, on gère donc les pressions des touches associées à la gravitation pour déterminer où le puyo va se placer :

```
1
2       // Gravitation sens horaire
3       si (evenement = D
4           ET couplePeutGraviter(couplePuyos,
5           (posGravSecond + 1) % 4)) alors
6           posGravSecond = (posGravSecond + 1) % 4
7       // Gravitation sens anti-horaire
8       sinon si (evenement = Q
9           ET couplePeutGraviter(couplePuyos,
10              (posGravSecond - 1) % 4)) alors
11          posGravSecond = (posGravSecond - 1) % 4
12      finSi
13
14
15      // Positionnement du puyo secondaire
16      gravite(couplepuyos,posGravSecond)
17
18      si (NON couplePeutEtrePlace(couplePuyos,
19          coupleMatPosX,coupleMatPosY + BAS)) alors
20          couplePuyos.estControlable = Faux
21      finSi
22  finSi
23
```

Une fois que que l'on a déterminé où le puyo secondaire va se placer, on le place :

```
1
2       // Permet de faire graviter le puyo secondaire a la
    position, comprit entre 0 et 3
3       action gravite(-> CouplePuyos cp, -> entier x)
4
```

Pour placer dans la matrice qui représente le plateau les couples de puyos une fois qu'ils sont placés on a besoin de :

```
1
2     // Permet de placer le couple dans la position [x][y] de
    la matrice
3     action placeCouple(-> CouplePuyos cp, -> entier x, ->
    entier y)
4
```

2.2.2 Gestion des scores

Le score du joueur évolue selon le nombre de puyos détruits. Le joueur peut bénéficier d'une multiplication du score s'il fait des combos. Il existe deux types de combos :

— Multiplication du score en fonction du nombre de chaînes détruites après la première destruction de chaînes de puyos.
— Multiplication du score en fonction du nombre de couleurs présentes lors des destruction de chaînes de puyos.

On incrémente le score comme ceci :

— s'il n'y a pas de combos alors le score est incrémenté de
$$\texttt{(10 fois le nombre de puyos détruits)}$$
— s'il y a des combos alors le score est incrémenté de
$$\begin{aligned}&\texttt{(10 fois le nombre de puyos détruits)} \times \\ &\texttt{((le nombre de couleurs différentes)} \times \\ &\texttt{(le nombre de couleurs différentes} - 1) + \\ &\quad \texttt{le nombre de chaînes détruites} - 3))\end{aligned}$$

On peut le traduire par cette action :

```
1
2     action updateScore (-> entier score, -> Puyo [][] combo)
3         si taille(combo) == 1, alors
4               score += taille(combo[0])*10
5
6         sinon
7             entier [][3] coul_utilisee
8
9             pour entier i allant de 0 a taille(combo)-1, pas
    de 1 faire,
10
11                si combo[i].couleur not in coul_utilisee,
    alors
12                      coul_utilisee += [combo[i].coul]
13                finSi
14            finPour
15
```

```
16
17          pour entier i allant de 0 a taille(combo)-1, pas
    de 1 faire,
18              score += taille(combo[i]) * 10 * (taille(
    coul_utilisee) * (taille(coul_utilisee)-1) + taille(combo)
    -3)
19          finPour
20      finSi
21   finAction
22
```

2.3 Configuration initiale

Au lancement du jeu, la grille est vide. Le score et le nombre de puyos détruits est initialisé à 0, la vitesse de chute est égal à 1 et le nombre de couleurs disponibles est égal à 4. Le premier couple de puyos va descendre sur la grille à la verticale au plus à gauche de l'aire de jeu. Le puyo secondaire est au dessus du puyo principal.

2.4 Règles du jeu

Le principe du jeu est de former des combinaisons de puyos pour les détruire et ainsi gagner des points. Le jeu Puyo Puyo est un jeu de Puzzle infini, il n'y a donc pas de moyen de gagner. Le but est simplement d'avoir le score le plus élevé possible. Le principe est de maintenir le plus longtemps possible les piles de puyos en dessous de la ligne de fin (haut de la zone de jeu). Lorsque la ligne de fin est dépassée, le joueur a perdu et le score est enregistré.

La condition de défaite peut être traduite comme cela :

```
1
2    int matMilieu = (longueur(2,Matrice)-1) / 2
3    si (Matrice[0][matMilieu] != Null) alors
4        gameOver()
5    finSi
6
```

Au début du jeu, la vitesse de chute est assez lente et qu'il n'y a que 4 couleurs disponibles. Plus on progresse dans le jeu, plus la vitesse varie et des puyos de couleurs différentes font leur apparition. On a le puyo vert clair qui apparaît après avoir détruit 220 puyos, le puyo bleu après en avoir détruit 360, et le puyo noir après en avoir détruit 600.

On a :

```
1
2       // Definition des couleurs disponibles
3       si (puyoDetruit < 220) alors
4           couleurDispo = 4
5
6
7       sinon si (puyoDetruit < 360) alors
8           couleurDispo = 5
9       sinon si (puyoDetruit < 600) alors
10          couleurDispo = 6
11      sinon
12          couleurDispo = 7
13      finSi
14
15      // Definition de la vitesse de chute
16      si (puyoDetruit < 135) alors
17          si (puyoDetruit < 25) alors
18              vitesseChute = 1
19          sinon si (puyoDetruit < 60) alors
20              vitesseChute = 2
21          sinon si (puyoDetruit < 115) alors
22              vitesseChute = 3
23          sinon
24              vitesseChute = 4
25          finSi
26
27      sinon
28          entier a = 0
29          si (puyoDetruit < 220) alors
30              a = puyoDetruit - 135
31          sinon si (puyoDetruit < 280) alors
32              a = puyoDetruit - 220
33          sinon
34              a = (puyoDetruit - 40) % 80
35          finSi
36
37          si (a < 5) alors
38              vitesseChute = 1
39          sinon si (a < 20) alors
40              vitesseChute = 2
41          sinon si (a < 40) alors
42              vitesseChute = 3
43          sinon
44              vitesseChute = 4
45          finSi
46      finSi
47
```

Conclusion

Centipède et Puyo Puyo sont des jeux en temps réel. C'est pourquoi on utilise une boucle qui se répète plusieurs fois par seconde. Centipède utilise un système de vies (dont le nombre de vies est de 3) et une seule collision suffit pour perdre une vie, ce qui fait réapparaître le joueur au centre de son aire de jeu.

En revanche, dans Puyo Puyo, il n'y a pas de vie. La partie s'arrête lorsque les puyos dépassent la ligne du haut. De plus, le joueur contrôle l'orientation et le déplacement du couple de puyos alors que dans Centipède le joueur peut déplacer sans contrainte le nain dans son aire de déplacement (sauf s'il entre en collision avec des champignons).

Les déplacements ne sont pas gérés de la même manière :

- Dans Centipède, les déplacements se font pixels par pixels et dépendent de la vitesse du joueur.
- Dans Puyo Puyo, les déplacements se font selon les positions de la grille de la matrice.

Pour conclure on peut retenir que les éléments clés à prendre en compte pour coder Centipède sont :

- La gestion des mouvements du nain et du mille-pattes.
- La détection des collisions entre le nain, les balles, les segments du mille-pattes et les autres ennemis.
- La mise à jour de l'état du jeu en fonction des actions du joueur.

Et les éléments clés à prendre en compte pour coder Puyo Puyo sont :

- La gestion de la grille de jeu et des couples de puyos.
- La détection des combinaisons de couleurs et leur suppression.
- La détection de la ligne de fin.

En résumé, la programmation de ces deux jeux nécessite une compréhension approfondie de leurs mécanismes de jeu, une bonne gestion des états et des interactions.

Résumé

Ce rapport est une analyse structurelle de deux jeux vidéo, Centipède et Puyo Puyo. Chaque chapitre de notre document est composé de deux parties principales : l'une est l'étude de l'état du jeu et de ses modèles (constantes et variables) et l'autre est l'étude de l'évolution du jeu. Chaque chapitre contient également une section **Configuration initiale** et **Règles du jeu**.

Ce rapport a été rédigé dans le cadre d'un projet d'ingénierie en Informatique par des étudiants en première année de licence.

This report is a structural analysis of two video games, Centipede and Puyo Puyo. Each chapter of our document is made of two main parts: one is the study of the game's state and its models (constants and variables) and the other one is the study of the game's evolution. Each chapter also contains an **Initial Configuration** and **Game Rules** section.

This report was written as part of an engineering project in Computer Science for a Bachelor's degree.