UFR SCIENCES & TECHNIQUES
UNIVERSITÉ DE FRANCHE-COMTÉ

Réseau Figure
CURSUS MASTER EN INGÉNIERIE

Projet d'initiation à l'ingénierie logicielle
L1 CMI

Curie Logan Gallone Nathan Perrin Jérémy

Tetris

Frogger

Année 2019-2020

Table des matières

Introduction

Ce projet d'ingénierie logicielle s'inscrit dans le cadre des modules du CMI informatique et a pour but de nous initier à la démarche traditionnelle d'ingénierie logicielle. Cette partie aborde l'étape d'analyse du système à réaliser, s'ensuivront les étapes de conception et de développement d'une application.

Dans cette première étape, nous identifierons les structures de données nécessaires, les comportements attendus de l'application ainsi que l'architecture logicielle à mettre en oeuvre.

Nous analyserons deux jeux vidéo phares des années 80, Frogger et un grand classique, Tetris. Le premier est un jeu vidéo d'arcade sorti en 1981. Le second est un jeu de puzzle très populaire, datant de 1984.

1 Tetris

1.1 Présentation du jeu

Tetris est un jeu conçu par Alekseï Pajitnov à partir de 1984 et édité par plusieurs sociétés au cours du temps (Nintendo par exemple). Les droits de Tetris appartiennent aujourd'hui à la société The Tetris Company. Selon le journaliste Bill Kunkel, "Tetris répond parfaitement à la définition du meilleur en matière de jeu : une minute pour l'apprendre, une vie entière pour le maîtriser."
Dans cette analyse, nous nous intéresserons à la version originale de Nintendo développée pour sa célèbre Game Boy.

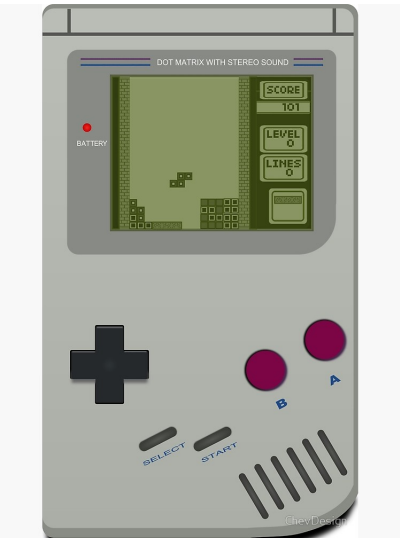

FIGURE 1 – Nintendo Game Boy avec le jeu Tetris

Le concept du jeu Pour gagner un maximum de points, le joueur doit compléter les lignes d'une grille à l'aide de pièces, appelées "tetrominos", qui apparaissent une à une en haut de la grille et tombent à travers celle-ci.
La complétion d'une ligne de cette grille entraîne sa suppression.
Si l'apparition d'un nouveau tetromino sur la grille est impossible, la partie est terminée. Une partie de Tetris ne se termine donc toujours par la défaite du joueur.

1.2 Éléments de l'état du jeu

Ce jeu, dans sa phase principale, présente quatre éléments à prendre en compte :

1. La grille de jeu ;
2. Le tetromino en cours de placement ;
3. La fenêtre des scores ;
4. La fenêtre de prévisualisation des pièces.

On définira donc une action `printGame()` qui effacera l'écran et affichera ensuite tous les éléments du jeu à l'écran.
Nous détaillerons donc chacun des éléments de ces fenêtres dans les parties suivantes.

1.2.1 La grille de jeu

Dans cette fenêtre du jeu, on retrouve les éléments suivants :
— la grille (de 20 par 10),
— les tetrominos fixés sur la grille,
— le tetromino en cours de placement par le joueur.

La grille pourrait être représentée par un tableau de booléens de taille 20x10 :

```
booleen [20][10] grid
```

Cependant, afin d'améliorer la jouabilité, on préférera associer des couleurs aux cases de la grille. Il vaudra donc mieux définir le tableau `grid` comme un tableau d'entiers.
Sachant qu'il existe 7 couleurs de pièces, une couleur différente pour chaque pièce, on représentera chacune de ces 7 couleurs par un entier compris entre 1 et 7.
Ainsi, si la case est vide on inscrit 0, sinon on inscrit la valeur correspondant à la couleur de la pièce présente.

De ce fait, on définit la grille par :

```
entier [20][10] grid
```

Où l'on initialise toutes les cases au début de la partie par la valeur 0.

Les tetrominos déjà placés Chaque tetromino correspond à un ensemble de 4 blocs ayant des coordonnées dans la grille.
Il s'avère donc nécessaire de définir une structure position :

```
Structure Position
   entier x
   entier y
Fin Structure
```

Si un tetromino occupe une case de la grille, la valeur de cette case devient différente de 0 et aura une valeur correspondant à celle assignée à la couleur du tetromino .
Cependant, le tetromino en cours de placement doit être géré différemment.

1.2.2 Le tetromino en cours de placement

Sachant que tous les tetrominos sont des combinaisons de 4 blocs, on peut les représenter par une structure composée de 4 positions, un entier correspondant au nombre de fois que le tetromino a été tourné et une couleur représentée par un entier :

```
Structure Tetromino
   Position block1
   Position block2
   Position block3
   Position block4
   entier nbRot <- 0
   entier color
Fin Structure
```

Désormais, il convient d'initialiser la position des différents tetrominos lorsqu'ils apparaissent sur la grille. Il existe un tetromino pour chaque combinaison de 4 blocs carré, donc 7.
Sur la figure suivante, les blocs sont numérotés, et nous nous référerons à ces numéros de blocs dans toute la suite de l'analyse.

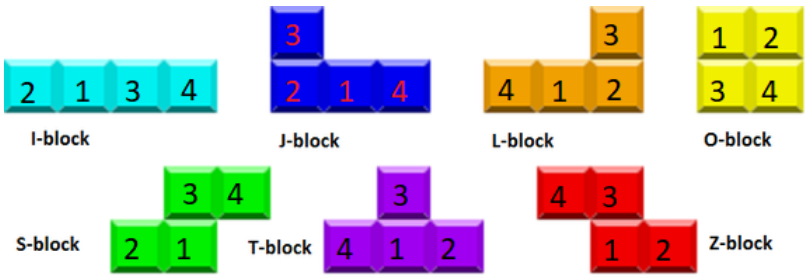

FIGURE 2 – Les Différents tetrominos existants

Le tableau ci-après représente les coordonnées de tous les blocs des tetrominos lorsqu'ils apparaissent sur la grille.

bloc correspondant		i-Block	j-Block	l-Block	o-Block	s-Block	t-Block	z-Block
color		1	2	3	4	5	6	7
block1	**block1.x**	4	4	4	4	4	4	4
	block1.y	0	1	1	0	1	1	1
block2	**block2.x**	3	3	5	5	3	5	5
	block2.y	0	1	1	0	1	1	1
block3	**block3.x**	5	3	5	4	4	4	4
	block3.y	0	0	0	1	0	0	0
block4	**block4.x**	6	5	3	5	5	3	3
	block4.y	0	1	1	1	0	1	0

TABLE 1 – Position des blocs des différents tetrominos lors de leur apparition sur la grille

On définira donc un tableau de type position de taille 7 par 4 qui contiendra les valeurs présentées dans la Table 1 : `position [4][7] init`. Il permettra d'initialiser rapidement chaque nouveau tetromino de la grille grâce à l'algorithme décrit ensuite.

```
fonction Tetromino spawnTetromino (->entier color):
    tetromino res
    res.block1 <- init[0][color-1]
    res.block2 <- init[1][color-1]
    res.block3 <- init[2][color-1]
    res.block4 <- init[3][color-1]
    res.color <- color
    retourner res
Fin fonction
```

1.2.3 La fenêtre des scores

Cette fenêtre permet d'afficher différentes données relatives à la partie en cours d'exécution, elle apparaît sur l'écran de jeu tout au long de la partie. Dans cette fenêtre, le joueur peut visualiser :

— son score : une variable de type entier,
— son level : une variable de type entier,
— le nombre de lignes qu'il a complété : variable de type entier.

Cette fenêtre des scores peut-être représentée par une structure composée des 3 éléments listés précédemment :

```
Structure ScoresWindows
   entier score
   entier level
   entier lines
Fin Structure
```

Nous détaillons les variables composants cette structure dans les paragraphes suivants.

entier score Cette variable représente le score du joueur. Ce dernier peut augmenter son score en gagnant des points après avoir complété une ligne ou en accélérant la descente des tetrominos. Les points lui sont attribués selon le mode de calcul présenté dans la partie **??** de cette analyse.
Cette variable entière `score` n'a pas de valeur limite, elle est initialisée à 0 au début de la partie, et augmente jusqu'à la fin de la partie selon les performances du joueur.

entier level Cette variable représente le niveau atteint par le joueur, le level est incrémenté à chaque fois que le joueur a complété 10 lignes.
Cette variable entière level n'a pas non plus de valeur limite, elle est initialisée au début de la partie est est incrémentée selon selon la méthode décrite ci-dessus jusqu'à ce que la partie soit terminée.

entier lines Cette variable, est initialisée à 0 au début de la partie. Elle indique le nombre de lignes que le joueur a complété. Une ligne est considérée comme complétée lorsqu'elle est composée de 10 blocs qui mis bout à bout parcourt la largeur de la grille.

1.2.4 La fenêtre de pré visualisation des pièces

Cette fenêtre, qui apparaît sur l'écran de jeu tout au long de la partie, permet au joueur de connaître quel seras le prochain tetromino qui sera mis en jeu, et lui permet ainsi de placer le tetromino qu'il est en train de jouer de manière stratégique.
Cette fenêtre est actualisée à chaque fois que le joueur place un tetromino et ainsi de suite jusqu'à la fin de la partie.

1.3 Les actions de l'utilisateur et leurs influences sur l'état du jeu

Le joueur interagit uniquement avec les 4 touches directionnelles de son clavier, ses actions ont une influence uniquement sur le tetromino en cours de déplacement.
Le joueur dispose de trois actions applicables sur le tetromino qu'il est en train de jouer.

1.3.1 Le déplacement latéral du tetromino

Le joueur peut déplacer latéralement le tetromino qui est en mouvement. Pour cela il doit appuyer une fois sur la flèche $\rightarrow$ du clavier pour déplacer son tetromino d'une case vers la droite.
De manière analogue, un appui simple sur la flèche $\leftarrow$ lui permet de décaler son tetromino d'une case vers la gauche sur la grille de jeu.

Le décalage latéral d'un tetromino doit respecter deux contraintes :

1. Aucun bloc composant le tetromino ne doit sortir de la grille de jeu ;
2. Aucun bloc composant le tetromino ne doit se superposer à un bloc déjà présent sur la grille.

Le déplacement latéral du tetromino imposé par le joueur, modifie les valeurs de `grid`, en effet, si on déplace le tetromino sur des cases vides, les valeurs de ces dernières vont passer de 0 à une valeur comprise entre 1 et 7 correspondant à la couleur du tetromino déplacé.

Le décalage du tetromino doit, selon la première contrainte, prendre en compte la largeur de la grille de jeu `'grid'`,donc les coordonnées en `x` de tous

les blocs le constituant, ces composantes doivent donc être comprises entre 0 et 10 inclus. La seconde contrainte impose que le décalage du tetromino prenne en compte la présence d'un tetromino déjà placé sur la grille. Ainsi dans le cas d'un déplacement du tetromino vers la droite, on doit vérifier que la valeur de chaque case à droite de chaque bloc composant le tetromino soit bien nulle, si tel est le cas, le déplacement du tetromino sera possible, sinon le tetromino sera empêché de ce décalage vers la droite. On définira donc une fonction booléenne `rightCheckTetromino` prenant en paramètre le tetromino en cours de déplacement et la grille de jeu. Cet algorithme vérifie si les blocs ne sont pas au bord de la grille et si la case à droite de chacun de ces blocs est bien vide. On définira également l'action `rightShift` qui incrémente de 1 l'abscisse de tous les blocs du tetromino. L'algorithme de gestion du décalage à gauche d'un tetromino en cours de placement se construit de manière analogue à celui pour un décalage à droite.

1.3.2 L'accélération de la chute du tetromino

Le joueur peut également accélérer la chute du tetromino qui est en jeu en appuyant sur la flèche ↓ . Un appui sur cette touche force la descente du tetromino d'une ligne, ce qui revient à augmenter d'une unité, pour chacun de ses blocs, la coordonnée en y.
Le joueur a tout intérêt à utiliser cette fonction car il augmente son score d'un point à chaque ligne descendu en accéléré. L'algorithme de traitement d'une descente accélérée peut-être réalisé grâce à une fonction `downCheckTetromino` et une action `fall` qui seraient réalisées de la même manière que pour le décalage latéral.

1.3.3 La rotation du tetromino

L'utilisateur peut appuyer sur la flèche ↑ pour effectuer une rotation de son tetromino d'un quart de tour dans le sens anti-trigonométrique [1].
Selon les pièces, le centre de rotation n'est pas toujours le même. Nous décrirons donc, dans la table **??**, les différentes positions relatives des blocs du tetromino selon leur position précédente et la forme/couleur du tetromino.

1. Rotation de $\frac{-\pi}{2} \Leftrightarrow \arcsin(-1)$

Type de tetromino/couleur	**bloc du tetromino**	nbRot = 0	nbRot = 1	nbRot = 2	nbRot = 3
i-Block/1	block1	(x,y-1)	(x+1,y)	(x,y+1)	(x-1,y)
	block2	(x-1,y-2)	(x+2,y-1)	(x+1,y+2)	(x-2,y+1)
	block3	(x+1,y)	(x,y+1)	(x-1,y)	(x,y-1)
	block4	(x+2,y+1)	(x-1,y+2)	(x-2,y-1)	(x+1,y-2)
j-Block/2	block1	(x,y)	(x,y)	(x,y)	(x,y)
	block2	(x-1,y-1)	(x+1,y-1)	(x+1,y+1)	(x-1,y+1)
	block3	(x,y-2)	(x+2,y)	(x,y+2)	(x-2,y)
	block4	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)	(x+1,y-1)
l-Block/3	block1	(x,y)	(x,y)	(x,y)	(x,y)
	block2	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)	(x+1,y-1)
	block3	(x+2,y)	(x,y+2)	(x-2,y)	(x,y-2)
	block4	(x-1,y-1)	(x+1,y+1)	(x+1,y-1)	(x-1,y+1)
o-Block/4	block1	(x,y)	(x,y)	(x,y)	(x,y)
	block2	(x,y)	(x,y)	(x,y)	(x,y)
	block3	(x,y)	(x,y)	(x,y)	(x,y)
	block4	(x,y)	(x,y)	(x,y)	(x,y)
s-Block/5	block1	(x,y)	(x,y)	(x,y)	(x,y)
	block2	(x-1,y-1)	(x+1,y-1)	(x+1,y+1)	(x-1,y+1)
	block3	(x+1,y-1)	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)
	block4	(x+2,y)	(x,y+2)	(x-2,y)	(x,y-2)
t-Block/6	block1	(x,y)	(x,y)	(x,y)	(x,y)
	block2	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)	(x+1,y-1)
	block3	(x+1,y-1)	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)
	block4	(x-1,y-1)	(x+1,y-1)	(x+1,y+1)	(x-1,y+1)
z-Block/7	block1	(x,y)	(x,y)	(x,y)	(x,y)
	block2	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)	(x+1,y-1)
	block3	(x+1,y-1)	(x+1,y+1)	(x-1,y+1)	(x-1,y-1)
	block4	(x,y-2)	(x+2,y)	(x,y+2)	(x-2,y)

TABLE 2 – Les différentes positions relatives des blocs lors de la rotation d'un tetromino

Le jeu comprendra donc un tableau : `Tetromino [7][4] rotation` de taille 7 par 4 dont les éléments seront des objets de type Tetromino mis à jour à chaque fois que le tetromino joué change de position.
Le tetromino `rotation [1][3]` par exemple, correspond au tetromino à affecter au tetromino en cours de placement si celui-ci est un tetromino de couleur 2 (c.a.d un j-Block) ayant déjà été tourné 2 fois.
Notons également que si l'abscisse d'un bloc est supposée sortir de la grille ou se superposer à un autre tetromino, alors on effectue un décalage du tetromino sur le côté ou vers le haut afin de pouvoir effectuer la rotation. Si la rotation n'est toujours pas possible, alors il ne se passe rien.

Nous aurons donc besoin de définir si les coordonnées d'un tetromino sont valables à l'aide de deux fonctions booléennes supplémentaires : `checkTetromino` et `upCheckTetromino`.

En PDL, l'algorithme qui retourne un bloc est donc le suivant :

```
action rotation (->Tetromino tetromino,
                 -> entier [20][10] grid,
                 -> Tetromino [7][4] rotation)
    //Si le tetromino peut tourner sur place
    SI
    (checkTetromino(rotation[tetromino.color][(tetromino.nbRot+1)%4],grid))
    ALORS
     tetromino <- rotation[tetromino.color][(tetromino.nbRot+1)%4]
    SINON
     //si le tetromino doit se décaler vers la droite pour tourner
     SI
     (rightCheckTetromino(rotation[tetromino.color][(tetromino.nbRot+1)%4],
                                                                    grid))
     ALORS
         tetromino <- rotation[tetromino.color][(tetromino.nbRot+1)%4]
         rightShift(tetromino)
     SINON
      //si le tetromino doit se décaler vers la gauche pour tourner
      SI
      (leftCheckTetromino(rotation[tetromino.color][(tetromino.nbRot+1)%4],
                                                                    grid))
      ALORS
         tetromino <- rotation[tetromino.color][(tetromino.nbRot+1)%4]
         leftShift(tetromino)
      SINON
       //si le tetromino doit se décaler vers le haut pour tourner
       SI upCheckTetromino(rotation[tetromino.color][(tetromino.nbRot+1)%4],
                                                                     grid))
       ALORS
        tetromino <- rotation[tetromino.color][(tetromino.nbRot+1)%4]
        upShift(tetromino)
       FinSI
      FinSI
    FinSI
FinAction
```

1.4 Les règles du jeu et leurs influences sur l'état du jeu

Les règles que vérifie le jeu Tetris sont les suivantes :

— les tetrominos joués sont ceux précédemment affichés dans la fenêtre de prévisualisation,
— le level augmente à chaque fois que le joueur complète dix lignes et la vitesse du jeu augmente à chaque nouveau level,
— il existe deux moyens de marquer des points ;
— si une ligne de la grille est complétée, alors elle disparaît et toutes les lignes supérieures descendent d'un cran,
— il est possible de compléter plusieurs lignes en même temps,
— si l'apparition d'un nouveau tetromino est impossible alors la partie est terminée,
— les tetrominos chutent verticalement dans la grille et si un tetromino ne peut plus descendre, alors il est fixé, à sa position actuelle, sur la grille et le joueur joue le tetromino suivant.

Nous détaillerons donc les algorithmes de chacune de ces règles.

Tetromino joué Sachant que le tetromino à jouer est celui étant précédemment affiché dans la fenêtre de prévisualisation. Pour changer le tetromino joué, on récupère la couleur du tetromino figurant dans la fenêtre de prévisualisation (entier entre 1 et 7) et on récupère un nouvel entier aléatoire pour générer un nouveau tetromino dans la fenêtre de prévisualisation.
Dans la suite on appellera `nextColor` l'entier définissant le prochain tetromino.

Mise à jour du level Étant donné que le niveau augmente toutes les dix lignes complétées, on décrit ici l'algorithme permettant d'actualiser le level du joueur :

```
action updateLevel (-> ScoresWindows ScoresWindows ->):
    SI (ScoresWindows.lines %10 = 0) ALORS
        ScoresWindows.level = ScoresWindows.level+1
    FinSI
FinAction
```

On définit également un algorithme qui détermine l'intervalle de temps nécessaire à la chute d'un tetromino.

Cet intervalle de temps dépend du level, il est d'autant plus court que le niveau est élevé :

```
//renvoie un entier : nombre de millisecondes
fonction entier maxTime(-> entier level)
    entier malus
    SI level < 15 ALORS
        malus <- level * 60
    SINON
        malus <- 900
    FinSI
    retourner (1000-malus)
FinFonction
```

Marquer des points Pour augmenter son score, le joueur a deux possibilités :

— compléter des lignes,

— accélérer la chute du tetromino.

En ce qui concerne la complétion des lignes, le jeu calcule les points de la manière suivante :

Level	Points pour 1 ligne	Points pour 2 lignes	Points pour 3 lignes	Points pour 4 lignes
0	40	100	300	1200
1	80	200	600	2400
2	120	300	900	3600
⋮	⋮	⋮	⋮	⋮
9	400	1000	3000	12000
n	40 * (n + 1)	100 * (n + 1)	300 * (n + 1)	1200 * (n + 1)

TABLE 3 – Calcul des points en fonction du level et du nombre de lignes complétées

Pour mettre à jour le score en fonction du nombre de lignes complétées en un coup, on utilisera l'action suivante :

```
action comboLines (-> ScoresWindows scoresWindows ->,
                   -> entier nbLines, -> entier level)
    dansLeCasDe (nbLines)
        1 :
            scoresWindows.score = scoresWindows.score + 40*(level+1)
        2 :
            scoresWindows.score = scoresWindows.score + 100*(level+1)
        3 :
            scoresWindows.score = scoresWindows.score + 300*(level+1)
        4 :
            scoresWindows.score = scoresWindows.score + 1200*(level+1)
    FinCasDe
FinAction
```

Lignes complétées Lorsqu'une ligne est complétée, elle doit être supprimée.

On considère qu'une ligne est complétée lorsque les valeurs de toutes les cases de la grille sur cette ligne, sont non nulles. Une fonction indiquant si une ligne de la grille est complétée est donc indispensable :

```
fonction booleen isCompleted(-> entier y, -> entier [20] [10] grid)
    booleen res = false
    SI (grid[y] [0] != 0 ET
        grid[y] [1] != 0 ET
        grid[y] [2] != 0 ET
            .    .    .
            .    .    .
            .    .    .
        grid[y] [9] != 0 ) ALORS
            res = true
    FinSI
    retourner res
FinFonction
```

On définit ensuite une action qui parcourt les lignes de la grille, supprime celles n'ayant pas de zéro et fait descendre toutes les lignes suivantes d'un cran.

```
action updateLines (-> entier [20][10] grid ->, -> scoresWindows scoresWindows
 entier nbLines = 0
    POUR i allant de 19 a 1, pas de -1 FAIRE
        SI (isCompleted(i,grid)) ALORS
            scoreWindows.lines++
            nbLines++
            updateLevel(scoresWindows)
            POUR j allant de 0 a 9, pas de 1 FAIRE
                grid[i][j] = 0
            FinPOUR
            POUR j allant de i à 1, pas de -1 FAIRE
                lineFall(j,grid)
            FinPOUR
        FinSI
    FinPOUR
    scoresWindows.score <-
    scoresWindows.score + comboLines(nbLines, scoresWindows.level)
FinAction
```

Voici l'implantation de l'action `lineFall` utilisé dans l'algorithme précédent :

```
action lineFall(-> entier y, ->entier [20][10] grid ->)
    POUR i allant de 0 a 9 pas de 1, FAIRE
        grid[y][i] <- grid[y-1][i]
        grid[y-1][i] <- 0
    FinPOUR
FinAction
```

Fin de la partie Selon les règles du jeu, le joueur a comme objectif d'obtenir le score le plus élevé possible, il n'y a aucune condition de victoire, la partie se termine lorsque l'apparition d'une nouvelle pièce est impossible.

L'algorithme vérifiant la possibilité de continuer la partie est celui-ci :

```
fonction booleen gameOver(-> Tetromino newTetromino, -> entier [20][10] grid)
    booleen res = true
    SI (checkTetromino(newTetromino, grid)) ALORS
        res = false
    FinSI
    retourner res
FinFonction
```

Descente d'un tetromino La descente automatique d'un tetromino est traitée plus ou moins de la même manière que la descente forcée du tetromino. La principale différence réside dans le fait que la descente automatique a lieu à intervalles réguliers dans le temps. Par conséquent, nous arrivons au point où nous décrivons la boucle de jeu.
Il s'agit d'une boucle tournant en permanence, mettant à jour l'affichage à l'écran environ tous les $60^{\text{ème}}$ de seconde pour donner une impression de continuité du déplacement des tetrominos.

1.5 La phase d'initialisation

Pour commencer une nouvelle partie, il est primordial d'initialiser le jeu, pour cela une action d'initialisation est nécessaire. La phase d'initialisation de Tetris s'enclenche par l'exécution d'une action `initialisation()`lors du lancement du jeu. Elle se déroule selon cet ordre :

1. La grille du jeu est créée ;
2. Le premier tetromino est initialisé ;
3. La fenêtre des scores est initialisée ;
4. Le tableau des rotations (7x4) est initialisé.

?? : Durant cette étape, la grille du jeu matérialisée par un tableau à deux dimensions 20x10 est initialisé avec une boucle "POUR". Toutes les valeurs du tableau sont mise à zéro.

?? : Le premier tetromino à jouer est initialisé en faisant appel à la fonction `spawnTetromino` implémentée dans la section ??

?? : La fenêtre des scores est initialisée dans cette partie les valeurs des attributs de la structure `ScoresWindows` sont initialisées de cette manière :

```
ScoresWindows.score = 0
ScoresWindows.level = 0
ScoresWindows.lines = 0
```

?? : Enfin un tableau à deux dimensions (7x4) de type Tetromino contenant les rotations des tetrominos est initialisé. Comme précédemment, le tableau est rempli au moyen d'une boucle "POUR" en faisant appel à un algorithme `initRot` prenant en paramètre un entier indiquant la couleur du tetromino ainsi que le nombre de rotations que l'on doit lui imposer. Cet algorithme renvoie un objet de type `Tetromino` remplissant de ce fait chaque case du tableau contenant les différentes rotations.

1.6 La boucle de jeu

```
//Boucle de Jeu
TANT QUE (!gameOver(tetromino, grid)) FAIRE
startTime = currentTime
printGame()

SI (time > maxTime(scoresWindows.level)) ALORS
time <- 0
SI (downCheckTetromino) ALORS
fall(tetromino)
SINON
grid[tetromino.block1.y][tetromino.block1.x] <- tetromino.color
grid[tetromino.block2.y][tetromino.block2.x] <- tetromino.color
grid[tetromino.block3.y][tetromino.block3.x] <- tetromino.color
grid[tetromino.block4.y][tetromino.block4.x] <- tetromino.color

updateLines(grid, scoresWindows)
tetromino <- spawnTetromino(nextColor)
nextColor <- (entier) (Math.random()*7+1)
FinSI
FinSI

SI (event = downKeyDown) ALORS
```

```
          SI (downCheckTetromino(tetromino, grid)) ALORS
              fall(tetromino)
              ScoresWindows.score = ScoresWindows.score+1
          FinSI
      FinSI

SI (event = upKeyDown) ALORS
          rotation(tetromino, grid, rotation)
      FinSI

       SI (event = rightKeyDown) ALORS
             SI (rightCheckTetromino(tetromino)) ALORS
                 rightShift(tetromino)
             FinSI
         FinSI

      SI (event = leftKeyDown) ALORS
             SI (leftCheckTetromino(tetromino)) ALORS
                 leftShift(tetromino)
             FinSI
         FinSI
     Fin TANT QUE
```

2 Frogger

2.1 Présentation du jeu

Développé par Konami, édité par SEGA et sorti en 1981, Frogger est un jeu tout d'abord sorti sur bornes d'arcade. Il a plus tard eu droit à plusieurs adaptations et suites sur différentes machines au fil des années.
Le jeu est aujourd'hui considéré comme un classique du jeu vidéo et apparaît dans le livre : "Les 1001 jeux vidéo auxquels il faut avoir joué dans sa vie".

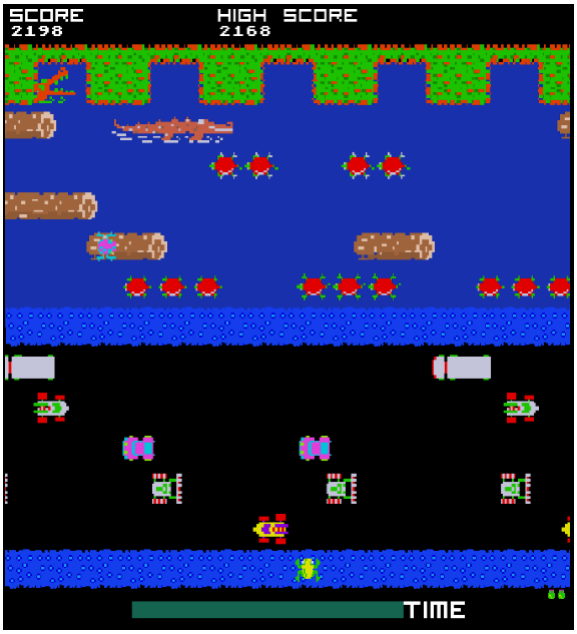

FIGURE 3 – Écran de jeu de Frogger

Le concept du jeu Le but est de diriger plusieurs grenouilles jusqu'à leur maison en traversant une route et une rivière. La difficulté du jeu réside dans la présence de voitures roulant toutes à des vitesses différentes et la possibilité de se noyer dans la rivière si l'on n'arrive pas à sauter de rondins en rondins.

2.2 Éléments de l'état du jeu

Dans sa phase principale, le jeu Frogger comporte un certain nombre d'éléments à prendre en compte :

1. La map où se déroule le jeu ;
2. Une interface graphique pour afficher le score ou d'autres informations.

Nous décrirons donc chacun de ces éléments.

2.2.1 La map

Le jeu se déroule sur une map carrée dont les côtés mesurent l'équivalent de 14 sauts de grenouille. On prendra par la suite le saut de grenouille comme unité de mesure de longueur. La map est donc un carré situé dans un repère dont l'origine est en haut à gauche, dont l'abscisse se lit de gauche à droite et l'ordonnée de haut en bas.

En parcourant la map de bas en haut, on remarque qu'elle est composée de plusieurs bandes :

— les bas côtés de la route de hauteur 1. Ces bas côtés sont horizontaux et sont placés en ordonnée 13 et 7,
— entre ces deux bas coté, la route de hauteur 5, elle est donc placée entre les ordonnées 8 et 12. On découpera cette route en 5 voies de circulations,
— De l'autre côté du second bas côté de la route, il y a la rivière de largeur 5, et que l'on découpera également en 5 "voies" dont l'ordonnée sera comprise entre 6 et 1,
— Sur la rive supérieur de la rivière se trouve les maisons des grenouilles, d'ordonnée 0.

Sur cette map, on retrouve de nombreux éléments :

1. La grenouille contrôlée par l'utilisateur ;
2. Les voitures circulant sur la route ;
3. Les rondins flottant sur la rivière ;
4. Les tortues nageant dans la rivière.

Pour définir les différentes structures qui composent le jeu, on aura recours à l'utilisation d'une structure générale `position` pour chacun des objets du jeu.

```
Structure Position
   entier x
   entier y
Fin Structure
```

La structure **Position** permet de stocker les coordonnées d'un objet (qu'il s'agisse d'une grenouille, d'une tortue,...) sur la zone de jeu.

Structure Frog La structure `Frog` permet l'instanciation des objets représentant les grenouilles que le joueur dirige dans le jeu.

```
Structure Frog
   Position pos
   constante entier SIZE
   caractère rotation <- u
   booléen life
   constante entier JUMP_LENGTH
Fin Structure
```

Un objet de la structure `Frog` agrège plusieurs informations :

— Un premier objet de la structure `Position` renseigne les coordonnées de la grenouille selon des axes x et y de la map.
— Une constante entière `SIZE` qui décrit la taille de la grenouille (utile pour savoir si elle est en contact avec un objet).
— Une variable de type caractère enregistre la direction dans laquelle regarde la grenouille par rapport à la map. Initialement attribuée à `u` (up), la valeur du caractère peut enregistrer les situation `u`, `d` (down), `l` (left), et `r` (right). Cette variable sert de suivi afin de modifier les sprites de la grenouilles après chaque nouveau déplacement.
— Une variable booléenne `life` pour indiquer que la grenouille est toujours en vie. La variable passe à `false` dans le cas où l'objet de type `Frog` percute un objet de type `Vehicle`, sort de la map, (en étant emporté par les obstacles flottants), ou simplement tombe dans l'eau. Si le joueur se trouve dans un de ces cas, il perd une vie.
— La constante `JUMP$\_$LENGTH`, qui nous sert d'unité pour décrire les positions de tous les objets dans le jeu.

Structure Vehicle Les objets de type véhicule sont construits par la structure `Vehicle`. Chaque véhicule est animé d'un mouvement horizontal uniforme.
Tous les véhicules circulant sur la même voie, roulent à la même vitesse, cependant la vitesse de circulation diffère en fonction des voies. De plus, plusieurs skins sont implémentés dans le jeu, ceci crée une certaine variété (chaque ligne de véhicules sur la route ayant un skin différent.

```
Structure Vehicle
   Position pos
   entier length
   entier speed
   entier skin
Fin Structure
```

La structure agrège quatre variables :
— une variable de type `Position` qui enregistre la position d'un véhicule sur la map,
— une variable de type entier `length` stockant la valeur de la longueur d'un objet (comprise entre 1 et 2 sauts de grenouille),
— une variable de type entier `speed` qui stocke la vitesse d'un objet. Cette vitesse est positive si l'objet se déplace vers la droite et négative s'il se déplace vers la gauche,
— une variable `skin` de type entier qui fait référence à des skins enregistrés dans le jeu selon un identifiant et les associes de ce fait à un objet.

Chaque fois qu'un nouveau véhicule apparaît à l'écran, on l'ajoute à un tableau de dimension 1 : `vehicle [] vehicleTab`. Quand celui-ci disparaît de l'écran, on l'efface. Ce tableau nous servira à vérifier si la grenouille est en train de percuter une voiture.

Structure Logs Tout comme les véhicules, les rondins de bois se déplacent de manière rectiligne à la différence des voitures qu'ils doivent être chevauchés par la grenouille afin de rejoindre les différentes maisons. Une structure `Logs` est donc nécessaire afin de pouvoir les représenter.

```
Structure Logs
   Position pos
   entier length
   entier speed
Fin Structure
```

La structure est composée de :
— une variable de type `Position` stockant la position d'un rondin de bois sur la zone de jeu,
— une variable de type entier `length` spécifiant la longueur d'une plateforme,
— une variable de type entier `speed` qui stocke la vitesse de l'objet. Cette vitesse est positive si l'objet se déplace vers la droite, réciproquement négative s'il se déplace vers la gauche.

Chaque fois qu'un nouveau rondin apparaît à l'écran, on l'ajoute à un tableau de dimension 1 : `Logs [] logsTab`. Lorsque celui-ci disparaît de l'écran, on l'efface.
Ce tableau nous servira à vérifier si la grenouille est posée sur un rondin.

Structure Turtle L'état des tortues est un élément important à prendre en compte par le joueur. Tout comme les rondins de bois, celles-ci traversent la rivière avec un mouvement de translation rectiligne uniforme.
La principale différence avec les rondins de bois réside dans le fait que les tortues peuvent plonger dans l'eau.
De ce fait, si la grenouille du joueur se trouve sur une de ces tortues lorsque celle-ci plonge dans l'eau, il perd alors une vie.

```
Structure Turtle
   Position pos
   entier length
   booleen dives
   entier divingTime
   entier underWater
   entier speed
Fin Structure
```

Cette structure caractérise chaque objet de type `Turtle` initialisé sur la

zone de jeu.
La structure agrège plusieurs données notamment :

- une position selon des coordonnées x et y dans un objet de type **Position**,
- la longueur d'une série de tortues (comprise entre 1 et 3 sauts de grenouille) est enregistrée dans la variable `length`, le fait que la série de tortues plonge dans l'eau à certains moments est défini par le booléen `dives`,
- la vitesse à laquelle se déplacent les tortues est enregistrée dans la variable `speed`,
- un entier `underWater` allant de 0 à 3 correspondant à l'état actuel des tortues (4 états différents possibles : de complètement émergées à plongées dans l'eau).

Chaque fois qu'un nouveau groupe de tortues apparaît à l'écran, on l'ajoute à un tableau de dimension 1 : `turtles [] turtleTab`. Quand celui-ci sort de l'écran de jeu, on l'efface.
Ce tableau nous servira à vérifier si la grenouille est posée sur une tortue.

Structure House Les maisons sont des espaces délimitées dans la map, lesquelles le joueur doit atteindre avec les grenouilles. Elles sont au nombre de 5.

```
Structure House
   Position pos
   constante entier SIZE
   booléen filled <- false
Fin Structure
```

La structure `House`agrège plusieurs données notamment :

- un objet de type `Position` enregistre l'emplacement de chaque maison sur la map,
- un entier `SIZE` permet de délimiter la "hit-box" de la maison,
- un booléen filled, initialisé à `false` indique si une maison à été rejointe par le joueur.

Afin de vérifier si une grenouille entre dans une maison, nous aurons besoin de stocker l'ensemble des maisons dans un tableau de dimension 1 : `House [5] houseTab`.

2.2.2 L'interface graphique

L'interface graphique du jeu indique au joueur :
- le temps restant pour traverser la route et la rivière afin d'atteindre sa maison, pour chaque essais,
- le nombre de vies qu'il lui reste (représenté par autant de grenouilles que de vies restantes),
- son score au moment de la partie en cours,
- le meilleur score (High-Score) réalisé sur la machine sur laquelle le joueur est en train de jouer.

Les différentes variables utilisées dans le programme, pour la gestion des données de l'utilisateur, sont regroupées dans une structure `ScoresWindows` :

```
Structure ScoresWindows
   entier timer
   entier liveNbr
   entier score
   entier highestScore
   entier level
   constante entier MAX_TIME <-60
   booleen [5] houseFill
Fin Structure
```

Les 4 premières variables présentées dans la structure seront affichées sur l'écran du joueur, et leurs valeurs dépendent en partie des autres variables et constantes citées.

entier timer Cette variable sert à stocker le temps restant au joueur pour chaque essai. Au début de chaque essai, on lui affecte la valeur de MAX_TIME, l'entier représentant des secondes.
`timer` est donc décrémenté de 1 chaque seconde. S'il atteint la valeur 0, le joueur perd une vie et obtient un nouvel essai s'il lui restait une vie. Sinon la partie est terminée.

entier liveNbr Cette variable stocke le nombre de vies du joueur. Elle est initialisée à 7 au lancement de la partie[2]. Chaque fois que le joueur crash sa grenouille, `liveNbr` est décrémenté de 1. De même si le temps lui étant accordé pour rentrer sa grenouille est écoulé (`timer` $== 0$). La partie continue tant que le nombre de vies du joueur n'a pas atteint 0.
Notons qu'il n'y a pas la possibilité de récupérer des vies.

entier score La variable enregistre le score du joueur en temps réel lors de sa partie actuelle. Ce score est de type entier et compris entre 0 et 9999.
Au début de la partie, le score est initialisé à 0, ensuite le joueur accumule des points selon l'algorithme décrit dans la partie **??**.

entier highestScore Permet de stocker le meilleur score ayant été réalisé par un joueur sur une machine. Il prend la valeur `ScoresWindows.score` si le score est supérieur au highestScore.

constante entier MAX_TIME Constante de type entier stockant le temps que le joueur a pour réaliser une traversée de la zone de jeu. Cette constante prend pour valeur 60.

booléen [5] houseFill Correspond à l'état de remplissage de chacune des maisons. Le tableau est donc initialisé à `false` pour toutes ses valeurs. Quand la n-ième maison est remplie, on affecte à `houseFill [n]` la valeur `true`.

On définira donc une action `printGame()` qui effacera l'écran et affichera ensuite tous les éléments du jeu à l'écran dans leur nouvel état, qu'il s'agisse de la fenêtre des scores, ou de la map et tout ce qui la compose.

2.3 Configuration initiale du jeu

Afin de mettre en lien toutes ces structures et faire fonctionner le jeu, il conviendra de les initialiser. Pour ce faire, on fera appel à une action `initialisation()`. Cette action servira à :

- placer la grenouille en bas au milieu de la fenêtre,
- remplir les tableaux `houseTab`, `logsTab`, `turtleTab` et `vehicleTab`,
- mettre à `false` toutes les valeurs de `ScoreWindows.houseFill`,
- initialiser la valeur de `previousPos` à : 13*frog.JUMP_LENGTH.

2. 7 n'a pas été choisi au hasard, c'est le nombre de lettres composant le nom du jeu

2.4 Les actions de l'utilisateur et leurs influences sur l'état du jeu

Le joueur n'a dans le jeu que la possibilité de faire avancer la grenouille qu'il contrôle selon quatre directions (avancer/reculer/aller à gauche/aller à droite) avec les quatre touches directionnelles du clavier. Nous allons détailler l'algorithme de déplacement d'une grenouille pour une direction, en sachant que les autres directions sont traitées de manière similaire.

```
action upShift(-> Frog frog ->)
    SI (event = upKey) ALORS
        frog.pos.y <- frog.pos.y - frog.jumpLength
        frog.rotation <- 'u'
    FinSI
FinAction
```

On regroupera toutes ces actions de déplacement dans une unique action `frogShift(->Frog frog->)`.

2.5 Les règles du jeu et leurs influences sur l'état du jeu

Les règles que vérifient le jeu Frogger sont les suivantes :
- les voitures, tortues et rondins se déplacent indépendamment,
- la grenouille se déplace dans le même sens que le rondin ou la tortue où elle se trouve,
- le joueur peut gagner des points en effectuant différentes actions,
- le meilleur score est enregistré dans highestScore,
- la grenouille meurt quand :
  - elle percute un véhicule,
  - elle rate une maison,
  - elle chute dans l'eau,
  - son temps de parcours s'est écoulé.

Mouvement des objets Dans Frogger, tous les objets ont une vitesse associée. Il est donc possible de les déplacer à l'aide des algorithmes suivants :

```
action moveVehicle(-> Vehicle [] vehicleTab ->)
    POUR entier i allant de 0 a longueur(vehicleTab)-1 FAIRE
        //insérer ici algorithme spécifique aux objets flottants sur l'eau
        vehicleTab[i].pos.x <- vehicleTab[i].pos.x + vehicleTab[i].speed
        //insérer ici algorithme spécifique aux tortues
    FinPOUR
FinAction

action moveThings(-> Vehicle [] vehicleTab ->, -> Turtle [] turtleTab ->, -> Logs
-> Frog frog ->)
    moveVehicle(vehicleTab)
    moveTurtle(turtleTab)
    moveLogs(logsTab)
    moveFrog(frog, turtleTab,logsTab)
FinAction
```

Les actions moveVehicle, moveTurtle et moveLogs utilisées dans le dernier algorithme fonctionnent sur le même principe. En revanche, `moveTurtle` possède une particularité représentée par le code ci-dessous :

```
SI (turtleTab[i].dives) ALORS
    SI (scoresWindows.timer % turtleTab[i].divingTime == 0 OU turtleTab[i].underWa
        turtleTab[i].underWater <- turtleTab[i].underWater - 1
        SI (turtleTab[i].underWater == -1) ALORS
            turtleTab[i].underWater <- 3
        FinSI
    FinSI
FinSI
```

Déplacement de la grenouille sur un rondin ou une tortue Afin de déplacer la grenouille lorsqu'elle se trouve sur un objet flottant sur l'eau, on intégrera aux algorithmes `moveTurtle` et `moveLogs` le sous algorithme suivant (on prend l'exemple du rondin) :

```
SI (frog.pos.x + frog.SIZE > logsTab[i].pos.x
    ET frog.pos.x < logsTab[i].pos.x + logsTab[i].length
    ET frog.pos.y == logsTab[i].pos.y) ALORS
        frog.pos.x <- frog.pos.x + logsTab[i].speed
FinSI
```

Marquer des points Le joueur peut marquer des points de quatre manières différentes :

1. En faisant avancer sa grenouille d'un pas il accrédite son score de 10. Par contre il en perd 10 lorsqu'il recule d'une case. Par contre un déplacement latéral de la grenouille ne modifie pas le score ;
2. Lorsqu'il ramène une grenouille saine et sauve dans sa maison, il remporte 50 points supplémentaires ;
3. Un bonus de temps lui est attribué en fonction du temps qu'il lui restait pour chaque essai lorsque celui-ci parvient à amener une grenouille à sa maison ;
4. Enfin la complétion d'un level, en remplissant les 5 maisons, lui rapporte 1000 points.

Voici l'algorithme qui est implanté dans la boucle de jeu servant à gérer la mise à jour du score du joueur selon tous les points cités ci-dessus :

```
action updateScore()
    // ajout de points quand une grenouille entre dans une maison
    POUR i allant de 0 à 4 FAIRE
        SI (scoresWindows.houseFill[i] != houseTab[i].filled) ALORS
            scoresWindows.score <- scoresWindows.score +50
            scoresWindows.houseFill[i] <- houseTab[i].filled
        FinSI
    FinPOUR
    // ajout de points quand toutes les maisons sont remplies
    //(1000 + bonus de temps restant)
    SI (isFull(houseTab)) ALORS
        scoresWindows.score <- scoresWindows.score + 1000 +
                               20*(scoresWindows.MAX_TIME -
                               scoresWindows.timer)
    FinSI

    // ajout de points selon l'avancement de la grenouille
    scoresWindows.score <- scoresWindows.score +
                           10*(frog.pos.y - previousPos)/frog.JUMP$\_$LENGTH
    previousPos <- frog.pos.y
FinAction
```

Mise à jour du meilleur score Algorithme de mise à jour du High Score. Cet algorithme n'est appelé que lorsque la partie est terminée.

```
    action updateHighScore (-> entier score, ->entier highScore ->)
        SI (score>=highScore) ALORS
            highScore <- score
        FinSI
    FinAction
```

Mort d'une grenouille Aucune contrainte n'empêche au joueur de faire avancer la grenouille qu'il contrôle là où il le désire sur la map. Cependant, il se peut que le joueur perde des vies pour différentes raisons :

- elle percute un véhicule ou un mur,
- elle rate une maison,
- elle chute dans l'eau,
- son temps de parcours s'est écoulé.

Si l'on prend en compte l'existence des variables générales `vehicleTab turtleTab logsTab houseTab`, l'algorithme qui met à jour `Frog.life` est le suivant :

```
action frogLifeUpdate(-> entier timer, -> Frog frog ->)
    SI (timer >= MAX_TIME) ALORS
        frog.life <- false
    SINON
        hitFrog(frog, vehicleTab)
        drownFrog(frog, turtleTab,logsTab)
        getInHouse(frog, houseTab)
    FinSI
FinAction
```

Où les actions hitFrog, drownFrog et getInHouse sont des algorithmes qui parcourent les différents tableaux pour vérifier :

- lorsque la grenouille est sur la route, si elle percute un véhicule,
- lorsque la grenouille est sur la rivière, s'il existe une tortue ou un rondin sur lequel elle se tient,
- lorsque la grenouille est au niveau des maisons, si elle est bien rentrée dans une maison et n'a pas sauté contre un mur.

2.6 Boucle de jeu

Pour finir de décrire le fonctionnement du jeu, nous pouvons désormais nous intéresser à la rédaction de la boucle de jeu. On supposera que toutes les variables utilisées ici auront été initialisées au préalable comme indiqué dans la section **??**.

```
TANT QUE (scoresWindows.liveNbr != 0) FAIRE
    printGame()
    moveThings(vehicleTab, turtleTab, logsTab, frog)
    frogShift(frog)
    frogLifeUpdate(scoresWindows.timer, frog)
    SI (!frog.life) ALORS
        frog.Life <- true
        frog.pos.x <- 6*frog.JUMP_LENGTH
        frog.pos.y <- 13*frog.JUMP_LENGTH
        scoresWindows.liveNbr <- scoresWindows.liveNbr - 1
    FinSI
    SI (scoresWindows.filled) ALORS
        scoresWindows.filled <- false
        scoresWindows.level  <- scoresWindows.level +1
    FinSI
    updateScore()
Fin TANT QUE
```

3 Conclusion

Les deux jeux analysés ici, malgré leurs gameplays très éloignés, présentent des similitudes intéressantes : si le joueur ne commet jamais d'erreur, alors la partie ne s'arrête jamais. C'est probablement l'un des points qui a contribué à leur succès. De même, afin que le joueur puisse déplacer à loisir l'objet principal du jeu, le développeur est conduit à réaliser des algorithmes vérifiant la validité de la position de l'objet.
Cependant, Tetris et Frogger restent deux jeux très différents. Tetris s'apparente à un casse-tête, tandis que Frogger se rapproche plus du style "Parcours du combattant".
Le tetromino en cours de placement se déplace de manière "discrète" (de case en case) tandis que la grenouille de Frogger semble suivre une certaine continuité dans ses mouvements. Dans notre analyse, cette continuité se constate dans le déplacement des voitures, des rondins et des tortues, qui ne se contentent pas de se téléporter d'une position vers la suivante puisque l'on retrouve la notion de vitesse.
Dans tous les cas, ces jeux sont des défis à relever et tous les joueurs, qu'ils soient novices ou invétérés, pourront trouver leur compte et prendre du plaisir à améliorer leur score ou battre ceux de leurs amis.