Complément de programmation : 2ème partie

1 Objectif

Cette partie de l'UE Complément de Programmation a comme objectifs principaux de :

1. poursuivre l'approfondissement des notions étudiées en Algorithmique - Programmation 1,
2. poursuivre l'étude des tests commencée en première partie de l'UE,
3. pratiquer le travail en groupe, qui est un mode de travail courant dans les métiers de l'informatique (et beaucoup d'autres),
4. vous initier à la rédaction d'une documentation.

La mise en œuvre se fait au travers de la réalisation d'un logiciel. Pour cette année, il s'agit d'un jeu inspiré directement du *Tetris* : *cf.* `https://fr.wikipedia.org/wiki/Tetris`.

En pratique, le plus simple est de travailler en ouvrant dans `emacs` les trois fichiers suivants :

1. le fichier `CPinter.ml` ; celui-ci est exécuté avant toute autre chose ;
2. le fichier `jeuCP2.ml`, disponible sur `UPdago`, que vous compléterez en incluant vos fonctions de jeu (*cf.* section 5 et suivantes), ainsi que ce qui est nécessaire pour la production de la documentation : *cf.* section 2 ;
3. un fichier `tests_jeuCP2.ml` contenant vos fonctions de test : *cf.* section 3.

NB1. Attention : ce n'est pas le jeu en tant que tel qui est l'objectif de l'UE [1]. Pour mémoire, le travail que vous effectuez durant cette 2ème partie de l'UE est noté, et la notation évalue votre travail par rapport aux objectifs précisés ci-dessus. Pour cela, il est indispensable que vous participiez aux TDs et TPs. En cas d'absence, prévenez votre enseignant de TD / TP et fournissez votre justificatif au secrétariat.

NB2. Il vous est demandé de travailler en groupe de 4 personnes (éventuellement 3 selon l'effectif du TD). Pour chaque séance, répartissez-vous les rôles dans votre groupe, et faites « tourner » ces rôles, de manière à ce que le nombre de fois où une personne assure un certain rôle soit le même pour tous les membres du groupe (par exemple, faites en sorte que chacun puisse intervenir autant sur les fonctions de jeu que sur les tests).
Servez-vous des commentaires et de la documentation pour tenir un « historique » des séances, de manière à ce que vous puissiez expliquer à votre enseignant comment se sont déroulés les travaux de votre groupe. En particulier, notez, pour chaque fonction et chaque test, quelle est la personne qui a réalisé cette fonction ou ce test.

NB3. Tout ce qui figure dans ce document est susceptible d'être précisé, voire adapté, pour votre groupe par votre enseignant de TD/TP.

2 Production de la documentation

2.1 Ouverture d'un « buffer » `shell` sous `emacs`

La production de la documentation utilise des commandes. Si vous travaillez sous `emacs`, il est nécessaire d'ouvrir un buffer dans lequel vous pourrez écrire ces lignes de commande ; c'est un buffer `emacs`, de même que le buffer dans lequel vous saisissez votre code ou le buffer dans lequel vous interprétez votre code (le menu `Buffers` vous permet de passer d'un buffer à l'autre).

1. Indépendamment de cela, les droits du jeu *Tetris* sont protégés : *cf.* `https://fr.wikipedia.org/wiki/Tetris`.

L'ouverture du buffer se fait de la manière suivante : appuyez sur la touche `echap` (ou `escape`, ou `esc`, ou autre suivant les claviers), puis sur la touche `x` : dans la ligne du bas de la fenêtre `emacs` apparaissent les caractères `M - x`
Saisissez à la suite `shell`
puis allez à la ligne. Un nouveau buffer est créé, dans lequel vous pouvez « travailler en ligne de commande ».

Vous pouvez taper les comandes de compilation et de production de la documentation. Reportez vous aux exemples du cours disponibles sur UPdago.

2.2 Production de la documentation des tests

Pour la production de la documentation des tests, il est nécessaire que votre fichier `jeuCP2.ml` soit « propre » et que votre fichier `tests_jeuCP2.ml` soit « propre » aussi (c'est-à-dire que toutes les commandes pour l'interpréteur, comme `#use "CPinter.ml";;`, doivent être supprimées ou mises en commentaires).

Pour produire une documentation commune aux deux fichiers `jeuCP2.ml` et `tests_jeuCP2.ml`, il convient de produire l'ensemble de la documentation simultanément. Procédez comme suit :

1. En premier lieu, il est nécessaire de compiler votre fichier `jeuCP2.ml` (voir exemple du cours). Vérifiez que les fichiers `jeuCP2.cmi` et `jeuCP2.cmo` ont bien été (re-)créés.
2. Suivant les systèmes dont vous disposez, il est sans doute nécessaire d'ajouter, en début du fichier `tests_jeuCP2.ml`, les lignes
   ```
   open CPutil;;
   open JeuCP2;;
   ```
3. Dans le buffer `shell`, saisissez la commande
   `ocamldoc -html -d doc -open CPutil jeuCP2.ml tests_jeuCP2.ml`
   Le fichier `index.html` est créé et permet d'accéder aux documentations des différents fichiers.

3 Tests

Le logiciel développé comporte des fonctionnalités d'affichage ; certains tests ne pourront donc être réalisés qu'en vérifiant visuellement le résultat produit. Pour cela, l'oracle de test ne sera pas réalisé automatiquement, mais par une question posée au testeur, ce dernier pourra y répondre en saisissant par exemple la chaîne de caractère "OK" (ou "KO"). Un `assert_equals` entre la réponse du testeur et la chaine de caractères `"OK"` permet d'enregistrer le succès (ou l'échec) du test et effectuer le rapport de test adéquate.

Le testeur doit pouvoir répondre à ces questions sans ambiguïtés. Pour cela les questions doivent être claires et précises. Il peut en outre être nécessaire de réaliser des affichages dédiés au test, pour avoir des repères visuels afin de vérifier les affichages réalisés par la fonction sous test. Autrement dit, le testeur doit pouvoir être une personne extérieur à l'équipe de développement et ne pas connaître le sujet du projet.

D'autres fonctions doivent être testées automatiquement, à l'instar de ce qui a été fait pour le jeu des *4 cartes*.

Concernant les tests, c'est à vous de les concevoir et de les réaliser, en fonction de ce que vous pensez être le plus judicieux pour la fonction de jeu que vous souhaitez tester. Il n'y a donc pas d'autres directives que :

1. toute fonction de jeu doit être testée ;
2. quel que soit le test effectué, faites un commentaire pour expliquez succinctement pourquoi vous avez choisi les tests réalisés.

4 Organisation

En début de chaque séance, le groupe s'organise et se répartit le travail.

Chaque membre du groupe doit être responsable d'une ou plusieurs fonctions à coder. Une fois codée la fonction doit être essayée sous l'interpréteur, puis documentée, sans omettre de préciser l'auteur. En cas de difficulté à coder une fonction un ou plusieurs autres membres du groupe pourront participer au code, l'ensemble des participants seront alors mentionnés comme co-auteurs de la fonction.

Une fois la fonction finie de codée et documentée, elle est transmise au responsable de son test. Ce dernier écrit le ou les tests, les exécute et documente les fonctions de tests, sans omettre de préciser l'auteur. Si l'un des tests est en échec (en KO), l'information est transmise à l'auteur de la fonction pour qu'il corrige son code.

Le responsable de l'intégration collecte les fonctions terminées et leurs tests. Il réalise les fichiers `jeuCP2.ml` et `tests_jeuCP2.ml` de la séance, vérifie que tous les tests sont en succès et que la documentation se génère correctement. En cas de soucis, il n'intègre pas les fonctions erronées dans les fichiers de la séance et demande à ses auteurs de les corriger. En début de séance suivante, il fournit à l'ensemble du groupe la nouvelle version des fichiers `jeuCP2.ml` et `tests_jeuCP2.ml`.

Chaque membre du groupe doit alternativement avoir les rôles de développeur, testeur et responsable de l'intégration. La bonne organisation du groupe, l'entraide, et la vérification mutuelle, fait partit intégrante de l'évaluation du travail de chacun. La note du projet est individuelle. La qualité du code produit compte pour 50% de la note, les tests, la documentation et l'organisation pour 50%.

5 Présentation générale

Le principe du jeu est connu (*cf.* figure 1) :

1. une forme apparaît en haut de l'espace d'affichage ;
2. si elle ne peut pas apparaître entièrement parce qu'elle est bloquée par d'autres formes apparues antérieurement, le jeu se termine ;
3. sinon, elle se déplace vers le bas, tant qu'elle n'est pas bloquée par d'autres formes qui se sont déposées antérieurement. Il est possible de contrôler des déplacements vers la droite ou la gauche, des rotations de 90 degrés vers la droite ou vers la gauche, de faire « tomber » la forme le plus bas possible. Une fois la forme bloquée, s'il y a des lignes entièrement remplies, elles disparaissent, ce qui entraîne le décalage vers le bas des lignes situées au-dessus des lignes disparues.

Une manière de faire consiste à distinguer l'*espace de travail* de l'*espace d'affichage* (appelé aussi *espace graphique* par la suite) : *cf.* figure 2. L'espace de travail est représenté par une matrice, l'espace d'affichage est une partie de la fenêtre graphique. Il y a bien sûr une correspondance entre les deux : chaque élément de l'espace de travail correspond à une zone carrée de l'espace d'affichage (sur la figure 2, les deux sont représentés par des carrés de même taille : cela n'a bien sûr aucun sens, il s'agit simplement de faciliter la visualisation de la correspondance).

Une forme n'existe dans l'espace de travail que quand elle est bloquée : tant qu'elle se déplace, elle n'apparaît pas dans l'espace de travail, elle « n'existe » que dans l'espace graphique. Cela facilite les tests effectués pour vérifier si une autre forme bloque la forme courante : comme la forme courante n'existe pas dans l'espace de travail, elle ne peut pas se bloquer elle-même.

6 Graphique

Les fonctions graphiques suivantes sont utilisées :

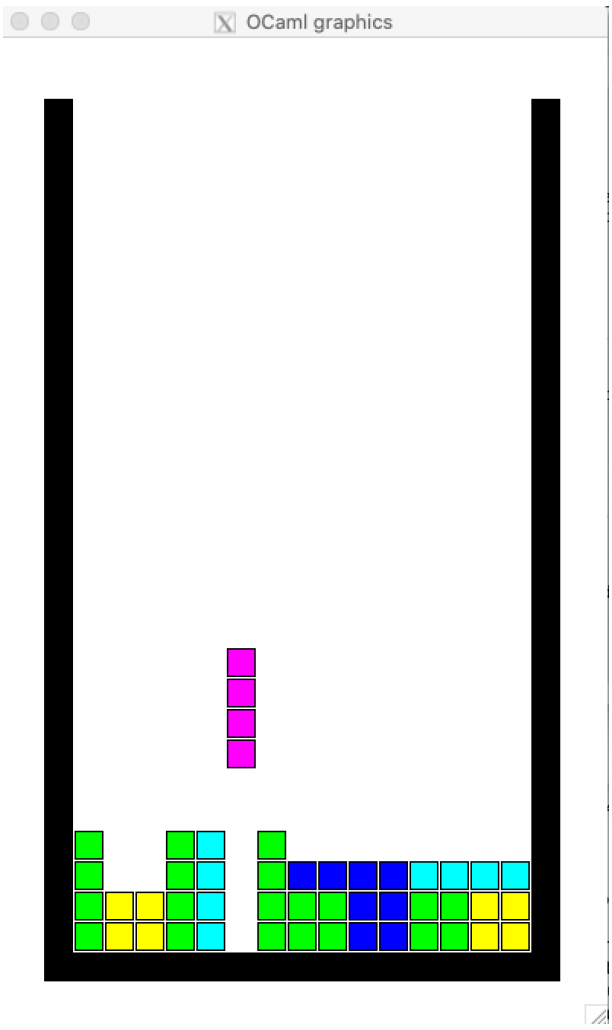

FIGURE 1 – Une image de la fenêtre graphique. Une forme est en cours de « descente ». Elle ne se déplace donc que dans l'espace graphique, elle est ignorée dans l'espace de travail. La forme n'est insérée dans l'espace de travail que quand elle ne peut plus se déplacer.

- `open_graph(dx, dy : int * int) : unit` ouvre une fenêtre graphique de taille `dx` par `dy` pixels. Le pixel en bas à gauche a pour coordonnées `(0, 0)`, le pixel en haut à droite a pour coordonnées `(dx - 1, dy - 1)` ;
- la fonction `clear_graph() : unit` « vide » la fenêtre graphique, c'est-à-dire qu'après exécution de la fonction, tous les pixels de la fenêtre graphique sont blancs ;
- la fonction `draw_rect(x, y, dx, dy : int * int * int * int) : unit` trace le contour d'un rectangle dont le pixel en bas à gauche a pour coordonnées `(x, y)`, et dont le pixel en haut à droite a pour coordonnées `(x + dx - 1, y + dy - 1)` ;
- la fonction `fill_rect(x, y, dx, dy : int * int * int * int) : unit` trace un rectangle, les paramètres sont similaires à ceux de la fonction `draw_rect` ;
- la fonction `set_color(col : t_color) : unit` définit `col` comme couleur courante. Initialement, la couleur courante est `black`. Les couleurs prédéfinies sont `black`, `white`, `blue`, `red`, `green`, `yellow`, `cyan`, `magenta`, `grey`. La fonction `color_of_rgb(r, g, b : int * int * int) : t_color` permet de calculer une couleur à partir de valeurs entières (comprises entre `0` et `256`) définissant les composantes rouge, verte et bleue de la couleur résultat.

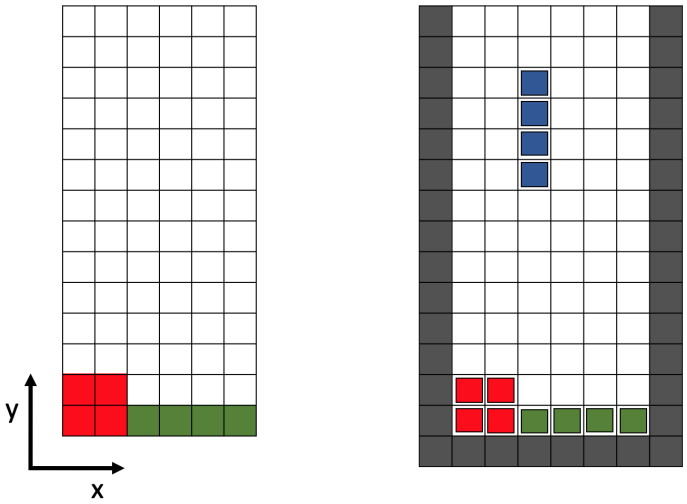

FIGURE 2 – À gauche, une représentation de l'espace de travail : c'est une matrice de taille `size_x` par `size_y` ; une abscisse est donc comprise entre `0` et `size_x - 1`, une ordonnée est comprise entre `0` et `size_y - 1`. À droite, une représentation de l'espace d'affichage correspondant à l'espace de travail (à gauche). Une forme est en cours de « descente ». Un élément de l'espace de travail de coordonnées `(x, y)` est associé à un carré de taille `dilat * dilat` pixels, où `dilat` est un coefficient de type `int`. Une translation est appliquée, afin que la base de l'espace d'affichage soit un peu décalée par rapport à la base de la fenêtre graphique (*cf.* figure 1). Un « cadre » incomplet d'épaisseur `d` entoure la partie de la fenêtre graphique correspondant à l'espace de travail.

Le type `t_point` permet de représenter des points de l'espace de travail :

```
type t_point = {x : int ; y : int} ;;
```

Question 1 : Définissez les fonctions suivantes :

- `draw_absolute_pt(p, base_draw, dilat, col : t_point * t_point * int * t_color) : unit` : les coordonnées du point `p` sont exprimées dans l'espace de travail, et il s'agit de tracer le contour du carré correspondant dans l'espace d'affichage (*cf.* section 5 et figure 2). Le point `base_draw` est l'origine de l'espace d'affichage ; la longueur du coté du carré [2] est égale à `dilat - 1`, et la couleur du tracé est `col` ;
- `fill_absolute_pt(p, base_draw, dilat, col : t_point * t_point * int * t_color) : unit`, qui trace un carré ; les paramètres ont la même signification qu'avant ;
- `drawfill_absolute_pt(p, base_draw, dilat, col : t_point * t_point * int * t_color) : unit`, qui trace un carré dont la couleur est donnée par le paramètre `col`, et dont le contour est noir ; les autres paramètres ont la même signification qu'avant.

Question 2 : Définissez les fonctions suivantes :

- `draw_relative_pt(p, base_point, base_draw, dilat, col : t_point * t_point * t_point * int * t_color) : unit` ;
- `fill_relative_pt(p, base_point, base_draw, dilat, col : t_point * t_point * t_point * int * t_color) : unit` ;
- `drawfill_relative_pt(p, base_point, base_draw, dilat, col : t_point * t_point * t_point * int * t_color) : unit`.

Ces fonctions effectuent les mêmes tracés que les fonctions précédentes, à la différence suivante près : le point `p` est défini relativement au point `base_point`. Dit autrement, chaque fonction est définie par un appel à la fonction correspondante précédente, avec en paramètre le point `p` translaté par le vecteur $\overrightarrow{TB}$, où T correspond à l'origine de l'espace de travail (de coordonnées

2. Dit autrement, le pixel en bas à gauche du carré est le point calculé à partir de `p` en appliquant une homothétie de rapport `dilat` et une translation de vecteur $\overrightarrow{GB}$, où G désigne le point origine de la fenêtre graphique, de coordonnées `(0, 0)`, et B correspond au point `base_draw`.

$(0, 0)$) et `B` correspond au point `base_point`.

Question 3 : Définissez les fonctions suivantes :
- `draw_pt_list(pt_list, base_pt, base_draw, dilat, col : t_point list * t_point * t_point * int * t_color) : unit`,
- `fill_pt_list(pt_list, base_pt, base_draw, dilat, col : t_point list * t_point * t_point * int * t_color) : unit`,
- `drawfill_pt_list(pt_list, base_pt, base_draw, dilat, col : t_point list * t_point * t_point * int * t_color) : unit`,

qui généralisent les trois fonctions précédentes pour le tracé de plusieurs carrés, dont les points (de l'espace de travail) sont indiqués dans la liste paramètre `pt_list`.

Question 4 : Définissez la fonction `draw_frame(base_draw, size_x, size_y, dilat : t_point * int * int * int) : unit`, qui trace en noir la partie de cadre entourant la zone d'affichage. Les entiers `size_x` et `size_y` indiquent la taille de la zone de travail. Les paramètres `base_draw` et `dilat` ont la même signification que dans les fonctions précédentes.

NB. Il y a correspondance entre les « coordonnées » de l'espace de travail et celles de l'espace graphique, c'est-à-dire que si `mat` est la matrice représentant l'espace de travail, `mat.(x).(y)` correspond au carré défini à partir du point de coordonnées `(x, y)`. Par rapport à la représentation de la figure 2, cela signifie qu'une ligne de la matrice `mat` correspond à une suite verticale de carrés, qu'une colonne de la matrice `mat` correspond à une suite horizontale de carrés, et que `mat.(0).(0)` correspond au carré en bas à gauche. Dit autrement, une colonne de la matrice correspond visuellement à une ligne pour le jeu.

7 Types, formes, paramétrage et initialisation du jeu

Comme il n'existe pas de fonction équivalent à `arr_len` dans tous les langages, nous définissons le type `t_array`, qui permet de représenter un tableau et sa longueur :

```
type 'a t_array = {len : int ; value : 'a array} ;;
```

Le type `t_shape` permet de représenter des formes « abstraites », et les 3 fonctions suivantes définissent les trois formes de la figure 1. La fonction `init_shapes`, définie ultérieurement, a pour résultat un tableau contenant ces trois formes :

```
type t_shape = {shape : t_point list ; x_len : int ; y_len : int ;
                rot_rgt_base : t_point ; rot_rgt_shape : int ;
                rot_lft_base : t_point ; rot_lft_shape : int} ;;
let init_sh011() : t_shape =
  {shape = [{x = 0 ; y = 0} ; {x = 1 ; y = 0} ; {x = 2 ; y = 0} ; {x = 3 ; y = 0}] ;
  x_len = 4 ; y_len = 1 ;
  rot_rgt_base = {x = 1 ;  y = 1} ; rot_rgt_shape = 1 ;
  rot_lft_base = {x = 2 ; y = 1} ; rot_lft_shape = 1}
;;
let init_sh112() : t_shape =
  {shape = [{x = 0 ; y = 0} ; {x = 0 ; y = -1} ; {x = 0 ; y = -2} ; {x = 0 ; y = -3}] ;
  x_len = 1 ; y_len = 4 ;
  rot_rgt_base = {x = -2 ;  y = -1} ; rot_rgt_shape = 0 ;
  rot_lft_base = {x = -1 ; y = -1} ; rot_lft_shape = 0}
;;
let init_sh211() : t_shape =
  {shape = [{x = 0 ; y = 0} ; {x = 0 ; y = -1} ; {x = 1 ; y = 0} ; {x = 1 ; y = -1}] ;
  x_len = 2 ; y_len = 2 ;
  rot_rgt_base = {x = 0 ;  y = 0} ; rot_rgt_shape = 2 ;
  rot_lft_base = {x = 0 ;  y = 0} ; rot_lft_shape = 2}
;;
```

Une valeur de type `t_shape` est définie par une liste de points `shape` (les coordonnées de ces points sont exprimées dans l'espace de travail). Ces points sont « relatifs », c'est-à-dire qu'ils sont définis à partir de l'origine de la forme, fixée à `(0, 0)`. On peut ainsi voir que toutes les

formes définies contiennent `4` points, que la première est horizontale, la deuxième est verticale, et la troisième est un carré. Plus généralement, une forme doit contenir un point de cordonnées `(0, 0)` ; les autres points ont des abscisses positives et des ordonnées négatives, puisqu'une forme apparaît en haut de l'espace d'affichage, ce qui correspond au haut de l'espace de travail.

Les champs `x_len` et `y_len` indiquent l'amplitude en `x` et en `y` de la forme.

Les champs `rot_rgt_base` et `rot_rgt_shape` définissent la forme résultant d'une rotation droite, les champs `rot_lft_base` et `rot_lft_shape` définissent la forme résultat d'une rotation gauche : *cf.* figure 3.

NB. Les rotations gauche et droite ont été choisies de manière à être inverses l'une de l'autre. Ce choix peut être remis en question dans les extensions que vous définirez (*cf.* section 12).

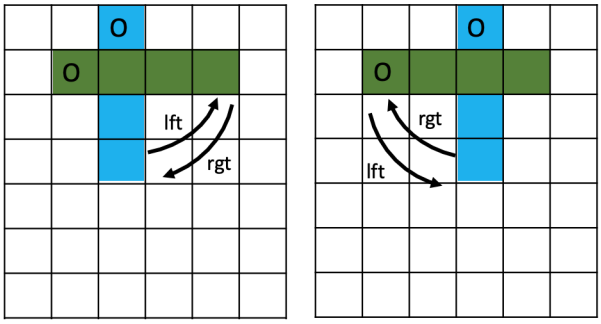

FIGURE 3 – Rotations. La forme définie par la fonction `init_sh011` est en vert, la forme définie par la fonction `init_sh112` est en bleu. Chaque point marqué `O` est l'origine de la forme correspondante. À gauche, rotation à droite de la forme en vert : l'origine de la forme résultante est au point de coordonnées `(1, 1)` (champ `rot_rgt_base`) par rapport à l'origine de la forme initiale ; la forme résultante a pour indice `1` (champ `rot_rgt_shape`) dans le tableau résultat de la fonction `init_shapes`. La figure à gauche peut aussi être interprétée comme la rotation à gauche de la forme en bleu, produisant la forme en vert. C'est similaire à droite pour la rotation à gauche (resp. à droite) de la forme en vert (resp. en bleu).

Le type `t_cur_shape` permet la définition de formes « concrètes », c'est-à-dire de formes évoluant dans l'espace d'affichage :

```
type t_cur_shape = {base : t_point ref ;
      shape : int ref ; color : t_color ref} ;;
```

Une forme concrète contient :

1. un « point de base » (dont les coordonnées sont exprimées dans l'espace de travail) qui indique à quoi correspond réellement le point origine de la forme ; ce point de base est contenu dans un champ mutable, puisque la forme concrète se déplace ;
2. un champ de type `int ref`, indiquant quelle est la forme associée à un instant donné à la « vraie » forme ; l'entier contenu dans ce champ est l'indice de la forme dans le tableau contenant le descriptif des formes, résultat de la fonction `init_shapes`. En effet, la forme peut varier si une rotation est appliquée ;
3. un champ stockant la couleur de la forme concrète. Pour la version de base du jeu, la couleur d'une forme concrète ne change pas au cours du temps, bien qu'elle soit stockée dans un champ mutable.

Les types suivants permettent de représenter les paramètres du jeu :

```
type t_param_time = {init : float ; extent : float ; ratio : float} ;;
type t_param_graphics =
    {base : t_point ; dilat : int ; color_arr : t_color t_array} ;;
```

```
type t_param =
  {time : t_param_time ;
   mat_szx : int ; mat_szy : int ;
   graphics : t_param_graphics ;
   shapes : t_shape t_array
  }
;;
let init_shapes() : t_shape t_array =
  {len = 3 ; value = [| init_sh011() ; init_sh112() ; init_sh211() |]}
;;
let init_color() : t_color t_array =
  {len = 7 ; value = [|blue ; red ; green ; yellow ; cyan ; magenta ; grey|]} ;;

let init_param() : t_param =
    {
    time = {init = 1.0 ; extent = 10.0 ; ratio = 0.8} ;
    mat_szx = 15 ; mat_szy = 28 ;
    graphics = {base = {x = 50 ; y = 50} ; dilat = 20 ; color_arr = init_color()} ;
    shapes = init_shapes()
    }
;;
```

Une valeur de type `t_param_time` contient les paramètres de temps : le champ `init` contient la durée initiale (en secondes) d'un déplacement ; le champ `extent` contient la durée entre deux accélérations, et le champ `ratio` contient le coefficient d'accélération. Avec les valeurs résultat de la fonction `init_param`, la durée d'un déplacement est de `1` seconde pendant les `10` premières secondes, puis de `0.8` seconde durant les `10` secondes suivantes, puis de `0.64` secondes durant les `10` secondes suivantes, *etc.*

Une valeur de type `t_param_graphics` contient les paramètres graphiques : le champ `base` contient l'origine de l'espace d'affichage ; le champ `dilat` contient le « coefficient d'homothétie » entre l'espace de travail et l'espace d'affichage ; dit autrement, si on considère que le coté d'une case de l'espace de travail a une longueur égale à `1`, la longueur correspondante dans l'espace d'affichage (exprimée en pixels) est égale à la valeur du champ `dilat`. Afin de bien voir dans l'espace d'affichage ce qui correspond à une case de l'espace de travail, la longueur d'un côté du carré effectivement tracé est égale à `dilat - 1`, comme indiqué précédemment (*cf.* figure 1). Le champ `color_arr` contient un tableau contenant les couleurs utilisables pour les formes concrètes. Avec les valeurs résultat de la fonction `init_param`, le point origine de l'espace de travail a pour coordonnées `(50, 50)` dans la fenêtre graphique, le coefficient d'homothétie est égal à `20`, et le tableau de couleurs est défini par la fonction `init_color` : les formes concrètes peuvent donc prendre chacune l'une des `7` couleurs disponibles.

Une valeur de type `t_param` contient les paramètres de temps et les paramètres graphiques, deux champs `mat_szx` et `mat_szy` contenant les dimensions de la matrice représentant l'espace de travail, ainsi qu'un tableau contenant les formes abstraites disponibles. Avec les valeurs résultat de la fonction `init_param`, la taille de l'espace de travail est de `15 * 28`, et les `3` formes disponibles sont contenues dans le tableau résultat de la fonction `init_shapes`.

Le type `t_play` permet de représenter les informations de jeu :

```
type t_play = {par : t_param ;
       cur_shape : t_cur_shape ; mat : t_color matrix} ;;
```

Le champ `par` contient les paramètres du jeu. Le champ `cur_shape` contient le descriptif de la forme actuelle, qui se déplace jusqu'à être bloquée ; dans ce cas, elle est intégrée dans la matrice de jeu, et une nouvelle forme est choisie comme forme actuelle. Le champ `mat` est la matrice décrivant l'espace de travail.

Question 5 : Documentez l'ensemble des types précédents.

Question 6 : Définissez des fonctions permettant d'extraire d'un paramètre chacune des valeurs qui y sont contenues. Comme pour le jeu des *4 cartes*, cela permettra de changer les types définissant les paramètres sans que cela ait d'autre incidence que la modification de ces fonctions d'extraction.

Question 7 : Écrivez les fonctions suivantes :

1. la fonction `color_choice(t : t_color t_array) : t_color` a pour résultat une couleur choisie aléatoirement dans le tableau `t` ;
2. la fonction `cur_shape_choice(shapes, mat_szx, mat_szy, color_arr : t_shape t_array * int * int * t_color t_array) : t_cur_shape` a pour résultat une valeur de type `t_cur_shape` : le numéro de la forme doit être choisi aléatoirement, ainsi que sa couleur et sa position. La position est choisie de manière à ce que la forme apparaisse :
   - le plus haut possible dans l'espace d'affichage ; cela correspond au fait que l'ordonnée du point base de la forme est la plus grande possible dans l'espace de travail ;
   - à un endroit quelconque, compte-tenu de la contrainte précédente ; cela signifie que l'abscisse du point base de la forme est choisie aléatoirement, mais de telle manière que l'ensemble de la forme tienne dans l'espace d'affichage. L'abscisse doit donc être supérieure ou égale à `0`, et inférieure ou égale à une valeur que vous déterminerez. Bien sûr, `shapes` est le tableau contenant les formes abstraites disponibles, `mat_szx` et `mat_szy` sont les dimensions de l'espace de travail, et `color_arr` est le tableau des couleurs disponibles ;
3. la fonction *récursive* `insert(cur, shape, param, my_mat : t_cur_shape * t_point list * t_param * t_color matrix) : bool` réalise l'insertion de la forme décrite par les paramètres `cur` et `shape` dans l'espace d'affichage ; les paramètres `param` et `my_mat` décrivent les paramètres de jeu et l'espace de travail. Le résultat de la fonction est un booléen dont la valeur est `true` si la forme peut être insérée sans qu'il n'y ait de « collision » avec une forme antérieure (ce qui peut arriver quand l'espace de travail contient des formes bloquées dont des points ont des ordonnées élevées), et `false` sinon. Le descriptif de forme `cur` permet en particulier de connaître le point base de la forme ainsi que sa couleur ; la liste `shape` contient les coordonnées des points de la forme abstraite qu'il reste à tester et à afficher.
4. la fonction `init_play() : t_play` réalise l'initialisation du jeu, c'est-à-dire :
   - initialise les paramètres de jeu et l'espace de travail ;
   - choisit et insère la première forme concrète ;
   - trace le cadre de l'espace d'affichage ;
   - a pour résultat le descriptif de jeu ainsi défini.

8 Déplacements

Le contrôle des déplacements se fait au clavier. La fonction `move` ci-dessous essaie d'appliquer le déplacement indiqué par le caractère `dir`. La signification d'un caractère entraînant l'application d'une opération est évidente. Le caractère `'v'` est utilisé pour indiquer un déplacement de la forme le plus bas possible : dans ce cas, l'opération n'est pas effectuée par la fonction `move`, mais par sa fonction appelante `new_step`, décrite ultérieurement :

```
let move(pl, dir : t_play * char) : bool =
  (
  if dir = 't'
    then rotate_right(pl)
    else
      if dir = 'c'
      then rotate_left(pl)
      else
        if dir = 'd'
        then move_left(pl)
        else
          if dir = 'h'
          then move_right(pl)
          else () ;
  (dir = 'v')
  )
;;
```

Question 8 :

1. écrivez la fonction `valid_matrix_point(p, param : t_point * t_param) : bool` qui teste si un point `p` est valide par rapport aux dimensions de l'espace de travail ;
2. écrivez la fonction `is_free_move(p, shape, my_mat, param : t_point * t_point list * t_color matrix * t_param) : bool`, qui teste si la forme décrite par le point base `p` et la liste de points `shape` d'une forme abstraite n'est pas en collision avec une forme insérée antérieurement. `my_mat` décrit l'espace de travail, `param` contient les paramètres du jeu ;
3. écrivez la fonction `move_left(pl : t_play) : unit` qui effectue, si c'est possible, un déplacement de la forme courante d'une case à gauche ;
4. écrivez la fonction `move_right(pl : t_play) : unit` qui effectue, si c'est possible, un déplacement de la forme courante d'une case à droite ;
5. écrivez la fonction `move_down(pl : t_play) : bool` qui effectue, si c'est possible, un déplacement de la forme courante d'une case vers le bas (le booléen résultat indique si le déplacement a été effectué) ;
6. écrivez la fonction `rotate_right(pl : t_play) : unit` qui effectue, si c'est possible, une rotation de la forme courante vers la droite ; des informations concernant les opérations de rotation ont été présentées précédemment : *cf.* section 7 et figure 3 ;
7. écrivez la fonction `rotate_left(pl : t_play) : unit` qui effectue, si c'est possible, une rotation de la forme courante vers la gauche ;
8. écrivez la fonction `move_at_bottom(pl : t_play) : unit` qui déplace verticalement la forme courante le plus bas possible, jusqu'à atteindre le bas de l'espace de travail, ou être bloquée par une forme figée antérieurement.

Comme expliqué précédemment, aucune de ces fonctions ne modifie l'espace de travail.

9 Suppression des « lignes » pleines

Quand la forme courante ne peut plus se déplacer, elle « se fige » : en pratique, elle est insérée dans l'espace de travail. S'il y a des « lignes[3] » pleines, elles sont alors supprimées ; chaque suppression de "ligne" entraîne le déclage des « lignes » du dessus.

Question 9 : Écrivez les fonctions suivantes :

1. `is_column_full(my_mat, y, zise_x : t_color matrix * int * int) : bool` ; cette fonction teste si une colonne de la matrice `my_mat` est « pleine » (`size_x` est la longueur d'une ligne de la matrice, et `y` est l'ordonnée des points de la colonne) ;

3. Attention : comme indiqué précédemment en section 6, ce qui correspond visuellement à une ligne est en fait représenté par une colonne de la matrice représentant l'espace de travail.

2. `decal(my_mat, y, size_x, size_y, par : t_color matrix * int * int * int * t_param) : unit` ; cette fonction décale vers le bas les colonnes de la matrice `my_mat` d'ordonnées strictement supérieures à `y`. `size_x` et `size_y` sont les dimensions de la matrice `my_mat` ;
3. `clear_play(pl : t_play) : unit`, qui « nettoie » l'espace de travail ; cette fonction supprime toutes les « lignes » pleines (le mot « ligne » étant entendu au sens visuel) ;
4. `final_insert(cur, shape, my_mat : t_cur_shape * t_point list * t_color matrix) : unit` ; cette fonction « fige » la forme courante, décrite par le descriptif de forme `cur` et la liste de points `shape` ; dit autrement, la forme est insérée dans la matrice ;
5. `final_newstep(pl : t_play) : bool` ; cette fonction réalise la dernière phase d'une étape de jeu (*cf.* section 10 suivante). En particulier, elle teste si la forme courante peut encore se déplacer. Si c'est le cas, la fonction ne fait rien ; sinon, cela signifie que la forme ne peut plus se déplacer (elle est bloquée par des formes apparues antérieurement). La forme courante est insérée, les « lignes » pleines sont supprimées, une nouvelle forme est choisie. Dans tous les cas, le booléen résultat de la fonction indique si le jeu doit s'arrêter.

10 Une étape de jeu

La fonction suivante gère une « étape » de jeu :

```
let newstep(pl, new_t, t, dt : t_play * float ref * float * float) : bool =
  let the_end : bool ref = ref (!new_t -. t > dt) and dec : bool ref = ref false in
  let dir : char ref = ref 'x' and notmove : bool ref = ref false in
    (
    while not(!the_end)
    do
      if key_pressed()
      then dir := read_key()
      else () ;
      dec := move(pl, !dir) ;
      dir := 'x' ;
      new_t := Sys.time() ;
      the_end := !dec || (!new_t -. t > dt) ;
    done ;
    if !dec
    then (move_at_bottom(pl) ; notmove := true)
    else notmove := not(move_down(pl)) ;
    if !notmove
    then the_end := final_newstep(pl)
    else the_end := false;
    !the_end ;
    )
;;
```

Une étape de jeu s'exécute dans un certain temps `dt` ; les paramètres `new_t` et `t` correspondent respectivement au « temps actuel » et au temps auquel une nouvelle étape de jeu a commencé. Dans une première phase, la fonction teste si le joueur a saisi une commande [4] : si c'est le cas, la commande est prise en compte, par appel de la fonction `move` dont le résultat est stocké dans la variable mutable `dec`. La variable `dir` prend alors une autre valeur, qui ne correspond à aucune commande de déplacement (ici, la valeur `'x'`). La valeur de `dec` indique si la commande éventuellement saisie correspond à un déplacement de la forme au plus bas de l'espace de travail. Si c'est le cas, la commande est exécutée, sinon la forme se déplace d'une « ligne » vers le bas. Si la forme ne peut plus se déplacer, la fonction `final_newstep` est appliquée (*cf.* section précédente).

4. Lors de l'expérimentation, faites bien attention à ne saisir qu'un caractère pour indiquer un déplacement : dit autrement, n'appuyez pas sur une touche en continu, car tous les caractères saisis sont pris en compte pour piloter les déplacements.

11 Fonction principale

La fonction principale du jeu est la suivante :

```
let jeuCP2() : unit =
  let pl : t_play = init_play() in
  let t : float ref = ref (Sys.time()) and new_t : float ref = ref (Sys.time()) in
  let dt : float ref = ref (time_init(pl.par)) and t_acc : float ref = ref (Sys.time()) in
  let the_end : bool ref = ref false in
    while not(!the_end)
    do
      the_end := newstep(pl, new_t, !t, !dt) ;
      if ((!new_t -. !t_acc) > time_extent(pl.par))
      then
        (
        dt := !dt *. time_ratio(pl.par) ;
        t_acc := !new_t
        )
      else () ;
      t := !new_t
    done
;;
```

Il s'agit essentiellement d'une boucle permettant d'enchaîner les étapes de jeu, et de la gestion de l'accélération. Assurez-vous de bien comprendre le fonctionnement de cette fonction.

NB. La fonction `mywait` est une fonction utilitaire pouvant être utilisée pour ralentir certaines phases du jeu :

```
(* fonction d'attente *)
let mywait(x : float) : unit =
  let y : float ref = ref (Sys.time()) in
  while (Sys.time() -. !y) < x
  do ()
  done
;;
```

12 Extensions

12.1 Questions et extensions obligatoires

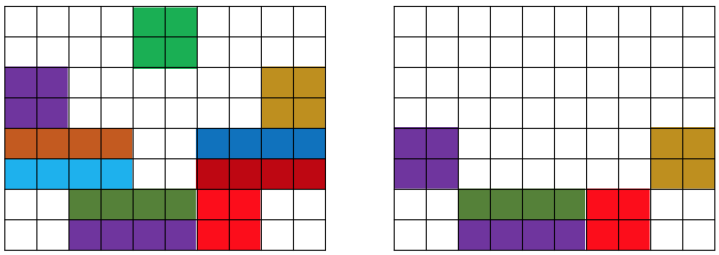

FIGURE 4 – À gauche, le carré vert n'est pas encore figé. S'il continue à se déplacer vers le bas, deux « lignes » pleines vont être supprimées, produisant la configuration de droite.

1. La couleur `white` est utilisée pour indiquer qu'une case est vide. Qu'aurait-il fallu faire pour permettre de modifier facilement le choix de cette « couleur vide » ? Faites-le.
2. Aucune vérification des paramètres de jeu n'est effectuée ; or, il existe des valeurs des paramètres qui ne sont pas cohérentes. Exprimez les contraintes sur les valeurs des paramètres de jeu, et écrivez une fonction vérifiant si les paramètres de jeu sont valides.
3. La suppression de « lignes » pleines peut aboutir à des configurations insolites, comme celle de la figure 4, où l'on pourrait s'attendre à ce que les deux carrés supérieurs « tombent en

bas ». Que faudrait-il faire pour éviter que ces configurations ne se produisent ? (il n'est pas demandé de le faire !)

12.2 Extensions libres

Vous pouvez donner libre cours à votre imagination, et adapter ce que vous avez réalisé jusqu'à présent pour offrir de nouvelles fonctionnalités (par exemple ajouter de nouvelles formes, ajouter la possibilité d'avoir des cases permettant de ralentir la vitesse du jeu si la forme courante y « passe », *etc*), voire pour modifier les règles du jeu.