CXXVIII - TP 2

Prise en main Pyxel - Labyrinthe

1 Introduction

On se propose dans ce TP de réaliser un mini-jeu en utilisant la librairie Pyxel, qui sera utilisée lors de la nuit du code. Toutes les fonctions proposées par cette librairie se trouvent dans le paragraphe intitulé documentation de l'api.

Tous les codes seront écrits dans un fichier commençant par :

Code

```
import pyxel
```

Nous allons structurer le code à l'aide d'une approche par programmation objet. Plusieurs classes seront ainsi écrites :

- une classe `App`, responsable d'initialiser une fenêtre graphique, d'afficher et de mettre à jour les différents composants graphiques, de gérer les événements claviers...
- une classe `Labyrinthe`, responsable de générer un labyrinthe, de se repérer dans un labyrinthe...
- une classe `Personnage`, responsable de créer et manipuler les différents personnages (héros, adversaires éventuels) habitant le labyrinthe.

2 Classe `App`

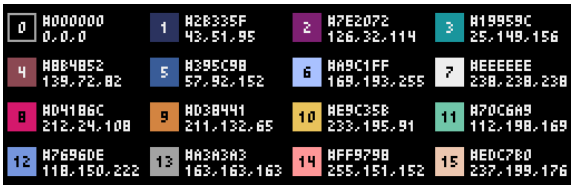
La fonction `init` du module `pyxel` initialise l'application Pyxel avec un écran de taille `(width, height)`. Il est possible de passer comme options : le titre de la fenêtre avec `title`, le nombre d'images par seconde avec `fps`, la touche pour quitter l'application avec `quit_key`, l'échelle de l'affichage avec `display_scale`. Lors de l'instanciation d'un objet de type `App`, la fonction `run` du module `pyxel` est exécutée, avec pour arguments la fonction `update` nécessaire pour mettre à jour chaque frame et la fonction `draw` pour dessiner sur l'écran quand c'est nécessaire (cf section suivante).
La nuit du code impose l'utilisation d'une fenêtre de taille `(128, 128)`, ayant pour titre `"La Nuit du Code"`.

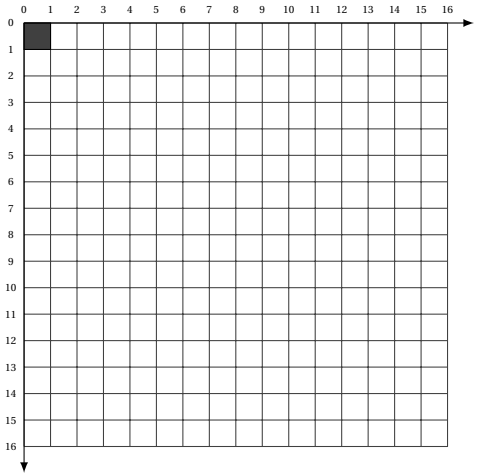
Code

```
class App:
    """ Application graphique - Pyxel """
    def __init__(self):
        """ App -> None """
        # initialisation de la fenêtre graphique
        pyxel.init(128, 128, title="La Nuit du Code")
        pyxel.run(self.update, self.draw)

    def update(self):
        """ App -> None
        Met à jour l'application """

    def draw(self):
        """ App -> None
        Affiche les éléments dans la fenêtre graphique """

App()
```

2.1 Méthodes `update` et `draw`

L'instruction `pyxel.rect(x, y, w, h, col)` dessine un rectangle de largeur `w`, de hauteur `h` et de couleur `col` à partir de `(x, y)` (coin supérieur gauche). Les couleurs par défaut (elles sont paramétrables) sont données dans la documentation de la librairie `pyxel`.

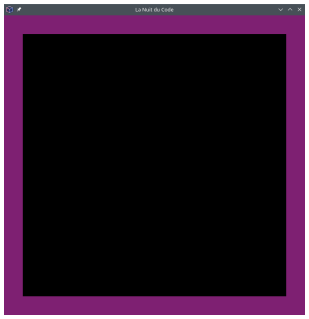




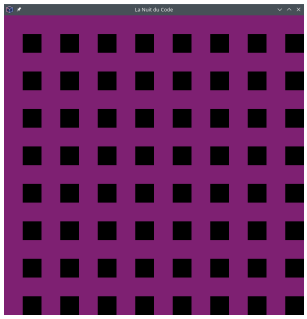
Attention. L'origine du repère est le coin supérieur gauche de la fenêtre. Les axes du repère de la fenêtre graphique sont orientés vers la droite et **vers le bas**. Ainsi, l'instruction permettant de dessiner le pixel noir de la figure ci-dessus est `pyxel.rect(0, 0, 1, 1, 0)`

Question 1. Compléter le code de la méthode `draw` de telle sorte que l'affichage réalisé lors de l'exécution du programme soit ("l'épaisseur" des traits est de 8 pixels) :

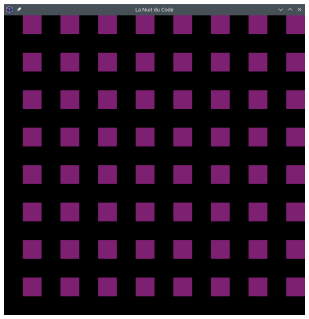
1. Un carré



2. Une grille



3. Des ilôts



3 Classe `Personnage`

Le personnage manipulé par le joueur possède une position `(x, y)` à l'écran. Il faut pouvoir le `deplacer` selon une `direction` et l'afficher à l'écran. La taille d'un personnage sera fixé à 8 pixels par 8 pixels. Initialement on représente le personnage à l'aide d'un carré de couleur, on verra par la suite comment utiliser une image pour l'affichage.

Code

```
class Personnage:
    """Un personnage dans le labyrinthe"""
    def __init__(self, x=0, y=0):
        """ Personnage -> None """
        self.x = x
        self.y = y

```

3.1 Méthode `deplacer`

Écrire une méthode `deplacer` de la classe `Personnage` qui met à jour les coordonnées du personnage `self` suivant le déplacement `direction`. `direction` est un couple d'entiers `(dx, dy)` : on incrémente `x` de `dx` et `y` de `dy`.

Code

```
def deplacer(self, direction):
    """ Personnage, (int, int) -> None
    Met à jour la position du personnage après un
    déplacement de direction = (dx, dy) """
    pass
```

3.2 Méthode `afficher`

Écrire une méthode `afficher` qui affiche un carré d'une certaine `couleur` de taille `(8, 8)` (en pixels) à la position `(x, y)` du personnage `self`. L'argument `couleur` sera un argument optionnel de la fonction et aura pour valeur par défaut `5`.

Code

```
def afficher(self):
    """ Personnage -> None
    Affiche le personnage à l'écran """
    pass
```

4 Notre premier `heros`

4.1 Ajout d'un personnage à l'application

Question 2. **1.** Ajouter un attribut `heros` de type `Personnage` à la classe `App`. Modifier les méthodes `update` et `draw` de telle sorte qu'à chaque frame : on déplace le personnage suivant la direction `(1, 1)` et on l'affiche à sa position courante.

Que constate-t-on ? Ajouter l'instruction `pyxel.cls(0)` pour corriger ce problème.

2. Modifier l'argument optionnel `fps` de la fonction `pyxel.init` : tester avec les valeurs `fps=1`, `fps=5`, et `fps=10`.

3. Dans la suite du TP, on divisera la fenêtre en `BLOC` de 8 pixels chacuns. Lors du déplacement d'un personnage, on ne pourra le déplacer que d'un `BLOC` suivant les directions `HAUT`, `BAS`, `GAUCHE`, `DROITE`. On écrira en tête de fichier les déclarations des constantes correspondant à cette situation.

Code

```
BLOC = 8
HAUT, BAS, GAUCHE, DROITE = (0, -BLOC), (0, BLOC), (-BLOC, 0), (BLOC,
↪  0)
```

a. Écrire les méthodes `haut`, `bas`, `gauche`, `droite` de la classe `Personnage` qui déplacent repectivement le personnage `self` d'un bloc vers le haut, le bas, la gauche, la droite. On fera appel pour cela à la méthode `deplacer`, ainsi qu'aux constantes correspondantes définies en tête de fichier.

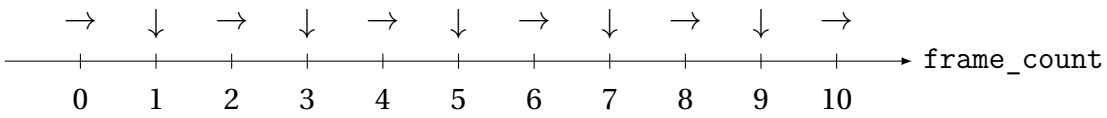
b. On cherche maintenant à modéliser une suite d'instructions à donner au personnage. Ajouter à la classe `App` un attribut `instruction` de type `list` définit de la manière suivante :

Code

```
self.instructions = [self.heros.droite, self.heros.bas]
```

Qu'y a-t-il de surprenant concernant les éléments de cette liste ?

c. L'attribut `frame_count` du module `pyxel` compte le nombre de frames passées depuis le début de l'application. Nous allons utiliser cet attribut pour répéter les `instructions` spécifiées.



Modifier la méthode `update` de telle sorte que lorsque le compteur de frames est pair, l'`instruction` d'indice `0` soit exécutée. Si le compteur de frames est impair, on exécute l'`instruction` d'indice `1`.

Pouvez-vous faire cela **sans utiliser de `if` ?**

4.2 Commandes clavier

On cherche maintenant à déplacer le personnage à l'aide des touches du clavier. L'instruction `pyxel.btn(key)` renvoie `True` si la touche `key` est appuyée, sinon renvoie `False`. On trouve (entre autres) dans la liste des touches les constantes :

- `pyxel.KEY_RIGHT`
- `pyxel.KEY_LEFT`
- `pyxel.KEY_DOWN`
- `pyxel.KEY_UP`

Question 3. **1.** Modifier la méthode `update` de la classe `App` : si la touche directionnelle `k` est pressée, alors on déplace le personnage dans cette direction en faisant appel à la méthode correspondante de la classe `Personnage`.

2. Lorsque l'utilisateur appuie sur une touche, cela correspond à une action à effectuer, ou encore à l'exécution d'une fonction sans arguments. Plutôt que de multiplier les instructions conditionnelles pour chacune des touches, on préfère ajouter à la classe `App` attribut `commandes` de type dictionnaire :

- les clés du dictionnaire correspondent aux touches pressées.
  Par exemple `pyxel.KEY_RIGHT`, `pyxel.KEY_LEFT`, etc.
- la valeur associée à la clé `k` est la méthode à exécuter lorsque la touche `k` est pressée.

Ainsi, le dictionnaire `commandes` est définit par :

Code

```
# commandes[touche] = action
self.commandes = {
    pyxel.KEY_RIGHT: self.heros.droite,
    pyxel.KEY_UP: self.heros.haut,
    pyxel.KEY_LEFT: self.heros.gauche,
    pyxel.KEY_DOWN: self.heros.bas
}
```

Écrire une méthode `exec_btn` de la classe `App` qui parcourt le dictionnaire `self.commandes` et exécute toutes les actions correspondant à des boutons pressés par l'utilisateur. Modifier la méthode `update` de la classe `App` : à chaque mise à jour on exécute l'entrée utilisateur en faisant appel à la méthode `exec_btn`.

Code

```
def exec_btn(self):
    """ App -> None
    Exécute toutes les commandes correspondant
    aux boutons actuellement pressés """
    pass
```

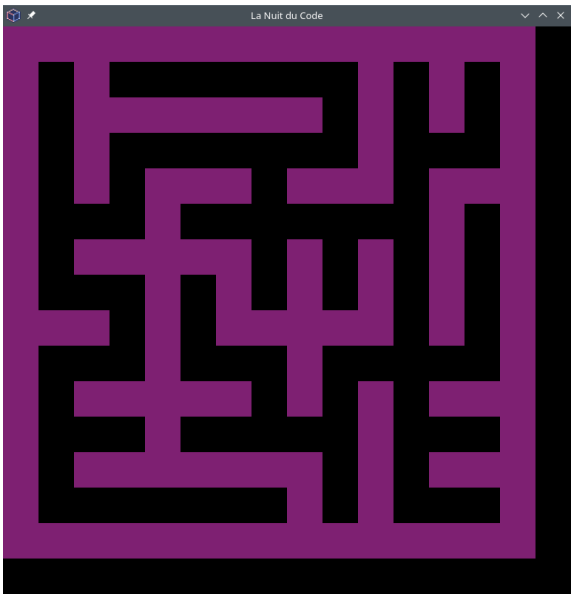
Exécuter l'application et tester les déplacements du personnage.

5 Représentation et génération d'un `Labyrinthe`

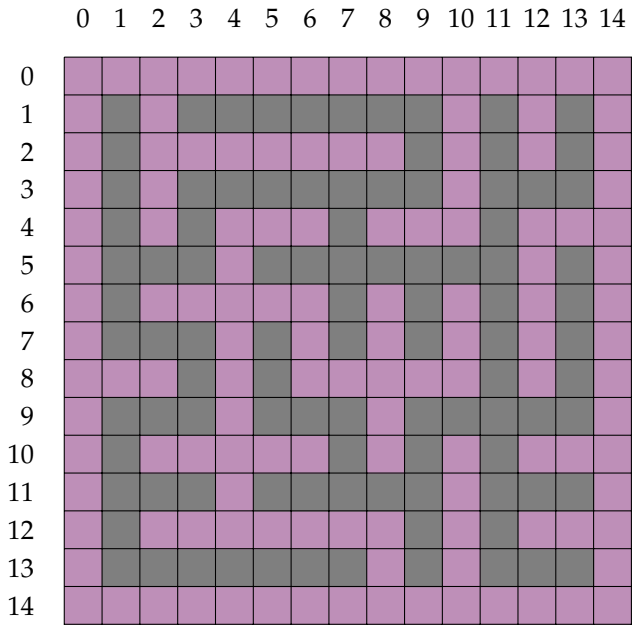
L'objectif de cette partie est d'aboutir à la génération automatique d'un labyrinthe, dans une fenêtre de taille `(128, 128)`. Chacun des blocs du labyrinthe aura pour taille 8 pixel par 8 pixel.

Pour des raisons de dimensions, une bande de 8 pixel sera laissée vacante sur le bord droit et sur le bord inférieur de la fenêtre.

On pourra trouver une description des algorithmes de génération aléatoire de labyrinthe sur la page modélisation mathématique d'un labyrinthe (wikipedia).



On représente un labyrinthe de taille `(n, m)` par une matrice de `murs` constituée de `n` lignes et `m` colonnes (`n` et `m` seront tous les deux des nombres impairs). Chaque élément de la matrice `murs` correspond à un bloc, qui peut être soit un mur, soit le sol.



On repère la position d'un bloc dans la matrice à l'aide de deux indices `i` et `j` vérifiant $0 \leq i < n$ et $0 \leq j < m$. Si `murs[i][j]` vaut `1` alors le bloc d'indice `(i, j)` est un mur ; si il vaut `0` il s'agit du sol. Initialement tous les blocs du labyrinthe sont des sols.

Attention. On utilisera exclusivement les indices `i` et `j` pour désigner l'indice d'un bloc du labyrinthe dans la matrice `murs`. On utilisera les noms `x` et `y` pour désigner une position à l'écran.

Code

```
class Labyrinthe:
    """Représente un labyrinthe"""
    def __init__(self, n, m):
        """Labyrinthe, int, int -> None
        Initialise un labyrinthe vide """
        assert n%2 == 1 and m%2 == 1
        self.dim = n, m
        self.murs = [[0 for j in range(m)]
                     for i in range(n)]

```

5.1 Méthode `est_dans`

Écrire les méthodes `est_dans` et `est_sol` de la classe `Labyrinthe`.

Code

```
def est_dans(self, bloc):
    """ Labyrinthe, (int, int) -> bool
    Détermine si bloc est un indice de bloc valide pour le labyrinthe self
    """
    pass

def est_sol(self, bloc):
    """ Labyrinthe, (int, int) -> bool
    Détermine si bloc est un indice de bloc valide correspondant à un sol
    dans le labyrinthe self """
    pass
```

5.2 Génération aléatoire d'un labyrinthe : exploration aléatoire

L'idée de l'algorithme est la suivante :

- On part d'un labyrinthe où tous les murs sont fermés. On stocke dans un ensemble `visitees` les blocs visités par l'algorithme. Initialement aucun bloc n'est visité.
- On choisit arbitrairement le bloc `(1, 1)` comme valeur initiale pour la `position_courante`. On marque ce bloc comme étant visité et on l'ajoute au `chemin`.
- Puis on regarde quelles sont les blocs voisins possibles et non visités.
    - S'il y a au moins une possibilité, on en choisit une `nouvelle_position` au hasard et on ouvre le mur entre `position_courante` et `nouvelle_position`. La `position_courante` devient la `nouvelle_position` et on ajoute la `nouvelle_position` au `chemin`.
    - S'il n'y en pas, on revient à la position précédente et on recommence.
- Lorsque l'on est revenu à la case de départ et qu'il n'y a plus de possibilités, le labyrinthe est terminé.

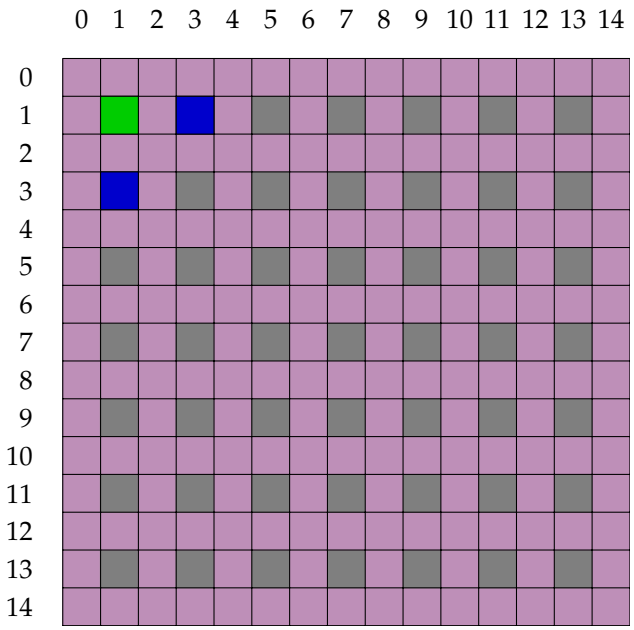
Question 4. **1.** Quelle structure de donnée utiliser lorsque l'on doit stocker les indices des blotcs constituant un `chemin` et que l'on doit revenir en arrière en cas d'impasse ?

5.2.1 Méthode `blocs_possibles`

Écrire une méthode `blocs_possibles` qui étant donné une position `(i, j)` dans le labyrinthe `self`, et un ensemble de blocs `visites` renvoie la liste des blocs voisins et non visités.

Attention. Les blocs renvoyés par cette méthode **ne sont pas** les blocs adjacents au bloc d'indice `(i, j)`.

Par exemple, dans la situation ou `position` vaut `(1, 1)` (en vert) et où l'ensemble des blocs `visites` est initialement vide, la méthode renverra les indices des blocs en bleu sur la figure.



Code

```
def blocs_possibles(self, position, visites):
    """ Labyrinthe, (int, int), {(int, int)} -> [(int, int)]
    Renvoie la liste des blocs voisins et non visités """
    pass
```

5.2.2 Méthode `generer`

Écrire le code de la méthode `generer` de la classe `Labyrinthe` qui génère un labyrinthe de manière aléatoire à l'aide de l'algorithme décrit dans la section **??**. Quelques indications :

- initialement, tous les blocs du labyrinthe sont des murs **sauf** les blocs d'indice `(i, j)` avec `i` **et** `j` impairs.
- on initialisera une variable `visites` de type `set`, et une variable `chemin` de type `list`, tous les deux initialement vides.
- la `position_courante` est initialement le bloc d'indice `(1, 1)`, que l'on ajoute à `visites` et `chemin`.
- On regarde quels sont les blocs voisins possibles et non visités tant que `chemin` n'est pas la liste vide (car alors cela veut dire que l'on a testé toutes les possibilités et qu'il n'y a plus rien à faire).
- on pourra utiliser la fonction `choice` du module `random` qui renvoie un élément choisit uniformément au hasard dans la liste passée en argument.
- l'indice du mur à ouvrir est la moyenne entre les indices des blocs `position_courante` et `nouvelle_position`.
- ajouter à la fin de la méthode `__init__` une instruction permettant de générer aléatoirement le labyrinthe.

Code

```
def generer(self):
    """ Labyrinthe
    Génère un labyrinthe de manière aléatoire """
    pass
```

5.3 Méthode `afficher`

Écrire une méthode `afficher` qui affiche le labyrinthe `self` à l'écran. Le coin supérieur gauche du bloc d'indice `(i, j)` aura pour coordonnées à l'écran `(i*BLOC, j*BLOC)`. Ce carré sera de dimensions `(BLOC, BLOC)`. On utilisera la couleur 2 pour les murs et la couleur 3 pour le sol.

Code

```
def afficher(self):
    """ Labyrinthe -> None
    Affiche le labyrinthe self à l'écran """
    pass
```

Question 5. **1.** Comment écrire la méthode `afficher` sans avoir recours à l'utilisation du mot-clé `if` ?

2. **a.** Modifier la classe `App` :

- lui ajouter un attribut `lab` de type `Labyrinthe` ;
- modifier la méthode `draw` afin de redessiner le labyrinthe à chaque frame. Attention à faire afficher le personnage **par dessus** le labyrinthe !

b. Exécuter l'application et tester l'affichage du labyrinthe.

6 Un peu d'interaction : soyons créatifs !

6.1 Un héros n'est pas un fantôme

Pour le moment les déplacements du personnage ne sont pas contraints par les murs du labyrinthe. C'est de la triche !

Question 6. **1.** Écrire une méthode `deplacer` de la classe `App` qui déplace le `perso` dans la `direction` indiquée si le bloc de destination correspondant du labyrinthe `self.lab` est de type sol.

Code

```
def deplacer(self, perso, direction):
    """ App, Personnage, (int, int) -> None
    Déplace le personnage dans la direction indiquée si cela est
    ↪ possible """
    pass
```

Indication. Si le personnage `perso` a pour coordonnées `(perso.x, perso.y)` alors le bloc correspondant a pour indice `(perso.x//BLOC, perso.y//BLOC)`. On admettra que la méthode `deplacer` est toujours appelée avec un argument `direction` parmi `HAUT`, `BAS`, `GAUCHE`, `DROITE`.

2. Modifier le dictionnaire des `commandes` afin d'utiliser la méthode `deplacer` de la classe `App`.

Indication. Si `self` est un objet de type `App` muni d'une méthode `deplacer`, alors l'instruction `lambda: self.deplacer(self.heros, HAUT)` renvoie une fonction (anonyme) sans argument qui exécute `self.deplacer(self.heros, HAUT)` lorsqu'elle est appelée.

6.2 À vous de jouer

Vous avez maintenant tout ce qu'il faut pour laisser libre cours à votre imagination ! Vous pouvez par exemple penser à faire ramasser à votre personnage des pièces avant d'atteindre la sortie, introduire des pièges, permettre au personnage de détruire des murs…

6.2.1 Gestion des collisions

Classe `Labyrinthe`

Code

```
def collision(self, position):
    """ (int, int) -> bool
    Renvoie True ssi la position = (x, y) à l'écran (en pixels)
    entre en collision avec un des murs du labyrinthe. """
    i, j = position[0]//BLOC, position[1]//BLOC
    ri, rj = int(position[0]%BLOC > 0), int(position[1]%BLOC > 0)
    return not all(self.est_sol((i + di, j + dj)) for di, dj in
                           [(0, 0), (ri, 0), (0, rj), (ri, rj)])

```

Classe `App`

Code

```
def deplacer(self, perso, direction):
    """ App, Personnage, (int, int) -> None
    Déplace le personnage perso dans la direction indiquée si cela est
    possible """
    dx, dy = direction
    position = (perso.x + dx, perso.y + dy)
    if not self.lab.collision(position):
        perso.deplacer(direction)
        position = (perso.x + dx, perso.y + dy)
```

On peut alors utiliser les directions en pixels (le déplacement est plus lisse).

Code

```
HAUT, BAS, GAUCHE, DROITE = (0, -1), (0, 1), (-1, 0), (1, 0)
```