Tutoriel Pygame

B. Lopez

Voici un petit tutoriel pour comprendre dans un premier temps les bases du module, de l'ouverture d'une fenêtre à la gestion des évènements. Dans un second temps, nous verrons les sprites, ces objets avec lesquels ont peut interagir dans un programme.

1 Pour commencer : importe Pygame

Comme vous l'avez vu en cours, il existe plusieurs façon d'importer un module en Python (ici le module Pygame), la plus simple consiste à écrire :

```
import pygame
```

mais on peut aussi écrire

```
import pygame as pg
```

ou encore

```
from pygame import *
```

Avec la première solution, vous importez le module et pour appeler une fonction du module (par exemple la fonction `init`), il faut alors écrire `pygame.init()`. Le deuxième choix importe Pygame de la même façon mais lui donne un alias (un pseudonyme si vous préférez). Cet alias est au choix, généralement plus court que le nom du module (c'est tout l'intérêt), et permet de simplifier l'écriture : `pygame.init()` s'écrira `pg.init()`.

La troisième solution importe toutes les fonctions et sous-module de Pygame, mais pas Pygame lui-même, ainsi on écrira directement `init()` au lieu de `pygame.init()`. Le problème de cette solution est qu'il devient impossible de différencier ce qui vient de Pygame et ce qui vient d'autres modules.

On utilisera donc toujours l'une des deux premières solutions, et généralement la deuxième.

Le module Pygame contient des constantes, comme `MOUSEBUTTONDOWN` par exemple qui correspond à l'évènement `on a cliqué avec la souris`, et il est nécessaire de les importer pour pouvoir les utiliser. En fait, ces constantes sont contenues dans le sous-module `pg.locals`, mais plutôt que d'écrire `pg.locals.MOUSEBUTTONDOWN`, on va vouloir écrire directement le nom des constantes (elles sont en majuscule donc facilement différentiables du reste dans le code). Finalement, pour les imports du module, on écrira :

```
import pygame as pg #importe le module avec comme alias pg
from pygame.locals import * #importe les constantes du module
```

2 Ouverture et personnalisation d'une fenêtre Pygame

La fonction `pg.init()` est la première fonction à utiliser car c'est elle qui initialise les fonctionnalités de Pygame.

Ensuite il faut créer la fenêtre, et pour ça il faut définir sa taille en utilisant un tuple. L'idéal est de définir une variable qui représentera ce tuple, afin de pouvoir le réutiliser par la suite. Ensuite, on crée la fenêtre en utilisant la fonction `pg.display.set_mode()` avec comme argument notre tuple (ou la variable). Cette fonction renvoie une `Surface` (un élément de Pygame représentant une zone graphique ou en d'autres termes une image).

```
SCREEN_SIZE = (800,600) #800 pixels en largeur, 600 pixels en hauteur
screen = pg.display.set_mode(SCREEN_SIZE)
```

La fenêtre est créée mais elle est vide (le fond est noir), il faut donc l'habiller un peu. Pour cela deux solutions : soit colorer la fenêtre, soit coller un fond de même taille (habituellement c'est cette deuxième solution qui est utilisée, pour une raison que l'on décrira par la suite). Ce fond peut être un fichier image ou une surface au départ vide que l'on colore totalement.

— Utiliser un fichier image en fond :

```
background = pg.image.load("background.png").convert()
background = pg.transform.scale(background, SCREEN_SIZE)
```

L'image de fond est chargée et la surface générée est stockée dans la variable `background`. La méthode `convert()` des `Surface` permet d'obtenir la surface la plus optimisée pour son utilisation ultérieure.
Ensuite, si besoin, la deuxième ligne redimensionne la surface à la taille de l'écran (il vaut mieux définir la taille de l'image en fonction de celle de la fenêtre que l'inverse).

— Utiliser une surface unie :

```
background = pg.Surface(SCREEN_SIZE)
background.fill((255,255,255))
```

On définit une surface de la taille de l'écran et on remplit (`fill`) avec une couleur au format RGB (rouge, vert, bleu) : chaque valeur est un entier entre 0 et 255 (ici le tuple (255,255,255) correspond au blanc).

Le fond a été créé, il faut maintenant le coller sur la fenêtre. En effet, en Pygame les surfaces peuvent uniquement se coller les unes sur les autres et ce de façon définitive (on ne peut pas bouger une image lorsqu'elle a été collée). Pour ce faire, on va coller, à l'aide de la fonction `blit`, notre fond sur notre fenêtre :

```
screen.blit(background, (0,0))
```

Le deuxième argument de la fonction `blit` est la position du coin supérieur gauche des deux images : on colle le coin de l'image background sur le coin de la fenêtre screen. Comme les deux images font la même taille, le fond remplira bien toute la fenêtre.

Accessoirement, on peut donner un nom à la fenêtre :

```
pg.display.set_caption("Super Jeu")
```

Pour que la fenêtre s'affiche avec le fond collé, il faut actualiser l'affichage avec la fonction suivante :

```
pg.display.flip()
```

Cette fonction doit être appelée à chaque fois que l'on met à jour un affichage (en théorie après chaque blit ou groupe de blit).

3 Gestion des évènements

À ce stade, la fenêtre est créée, un fond est collé dessus, mais elle ne reste pas ouverte, et c'est normal. Le langage Python étant un langage scripté, une fois que le traitement de tout votre script est effectué, le programme s'arrête, et le fenêtre se ferme. Il faut donc créer une **boucle infinie** pour empêcher le programme de se fermer et ainsi garder la fenêtre ouverte. Pour cela, on définit une variable qui vaut `True`, par exemple `play = True`, et on écrit une boucle infinie `while play :`. L'utilisation d'une variable va permettre de modifier sa valeur quand on souhaitera fermer la fenêtre, ce qui nous amène à gérer un premier évènement qui est la fermeture de la fenêtre si on clique sur la croix dans le coin supérieur (gauche ou droite selon le système d'exploitation) de la fenêtre.

Un **évènement** est une interaction de l'utilisateur : un clic sur la souris, une touche pressée sur le clavier, le relâchement d'une touche du clavier, le redimensionnement de la fenêtre, etc.

La première chose à faire est de récupérer la liste des évènements, ce qui se fait par l'appel à la fonction `pg.event.get()`. Cette fonction retourne la liste des évènements qui ont eu lieu depuis le dernier appel de cette même fonction. Il faut par conséquent utiliser une boucle `for` pour parcourir cette liste d'évènements. Ensuite, chacun de ces évènements a un type, qui donne la nature de l'évènement : un clic à la souris, un appui sur une touche du clavier, etc. L'évènement correspondant à la fermeture de la fenêtre par la croix dans le coin de la fenêtre est de type `QUIT` (qui est une constante Pygame correspondant à la fermeture de la fenêtre). On peut donc écrire le code suivant :

```
play=True
while play:
    for event in pg.event.get():
        if event.type == QUIT:
            play = False

pg.quit()
```

Si on clique sur la croix, `play` prend la valeur `False` et ainsi on sort de la boucle `while` et le programme s'arrête.

Une fois sortie de la boucle Pygame, il est nécessaire de libérer les modules Pygame qui ont été chargés lors de l'appel à la fonction `pg.init()`. Pour cela il faut appeler la fonction `pg.quit()`.

Il existe une quinzaine de types d'évènements, mais deux d'entres eux sont plus couramment utilisés : la pression d'une touche et le clic sur la souris, que nous allons d'écrire séparément.

3.1 Pression d'une touche du clavier

Le type correspondant à cet évènement est `KEYDOWN`, donc le test

```
if event.type == KEYDOWN:
```

vérifie si une touche est ou a été pressée. Si on veut simplement savoir si l'utilisateur a pressé une touche, n'importe laquelle, alors on peut s'arrêter là. A contrario, si on veut vérifier que l'utilisateur a pressé une touche particulière, il faut préciser la touche en question.

Pour cela, il faut noter que chaque type d'évènements à des attributs qui lui sont propres. L'évènement `KEYDOWN` à les trois attributs suivants :

— `key` : un identifiant entier représentant toutes les touches du clavier. Ces clés sont désignées dans le code par des constantes du module Pygame et commencent toutes par un K (`K_ESCAPE` pour la touche ECHAP, `K_a` pour la touche a, etc.).
— `unicode` : un caractère représentant la valeur de la touche pressée dans le langage du clavier (`\t` pour la touche TAB, `a` pour la touche a, etc.).
— `scancode` : un identifiant entier qui correspond à l'emplacement physique d'une touche sur le clavier, indépendamment du langage de celui-ci.

Pour les deux derniers attributs, la notion de langage est une notion de choix au niveau du système d'exploitation : on peut avoir un clavier physique de type AZERTY français pour PC Microsoft (les touches ne sont pas les mêmes que pour un Mac), et "dire" à son système de l'interpréter comme un clavier QWERTY ou encore comme un clavier en cyrillique. Cela changera l'attribut unicode, un A en AZERTY deviendra un Q sur le même clavier mais interprété en QWERTY, mais pas le scancode puisque c'est la même touche physique qui est pressée dans les deux cas. De plus, le scancode dépend entièrement du clavier utilisé, la valeur de la touche A sur un clavier Mac peut être différente de la valeur du A sur un clavier Microsoft par exemple.

Par conséquent, nous utiliserons exclusivement l'attribut `key` qui est généralisé à tout type de clavier.

La liste de toutes les clés possibles (les valeurs de event.key) se trouve à l'adresse suivante : `http://www.pygame.org/docs/ref/key.html` .

Ainsi, pour effectuer une opération spécifique lorsqu'une ou plusieurs touches sont pressées, par exemple les flèches du clavier, on écrira :

```
if event.type == KEYDOWN :
    if event.key == K_DOWN :
        # instructions
    elif event.key == K_UP :
        # instructions
    elif event.key == K_RIGHT :
        # instructions
    elif event.key == K_LEFT :
        # instructions
```

3.2 Clic sur la souris

Le type correspondant à cet évènement est `MOUSEBUTTONDOWN`, donc le test

```
if event.type == MOUSEBUTTONDOWN:
```

vérifie si l'utilisateur a cliqué sur la souris.

Comme dit précédemment, chaque type d'évènement a des attributs propres, ceux de `MOUSEBUTTONDOWN` sont :

— `pos` : un tuple (x,y) représentant la position en pixel en abscisse et en ordonnées du clic (l'origine du repère se situant dans le coin supérieur gauche de la fenêtre).
— `button` : un entier correspondant au clic effectué : 1 pour le clic gauche, 2 pour le clic droit, etc.

Pour l'attribut `button`, les entiers 4 et 5 ont des valeurs particulières : en effet ils ne correspondent pas à des clics mais aux mouvements vers le haut (pour le 4) et vers le bas (pour le 5) de la molette.

Généralement, dans les programmes simples implicant l'utilisation de la souris, on ne considère que le clic gauche de la souris, on peut donc se contenter de vérifier s'il y eu un clic à la souris, peu importe avec quel bouton.

L'attribut le plus important est `pos`, car c'est lui qui va donner la position au pixel près du clic. Or si on considère l'évènement de clic sur la souris c'est pour interagir avec l'écran, et cet interaction va dépendre de la zone sur laquelle on clique (un bouton, une image, etc.), il est donc important de connaître la position du clic.

Par exemple, pour vérifier si l'utilisateur clique sur un bouton rectangulaire de largeur 20 pixel et de hauteur 10, dessiné en position (500,300) (la position du coin supérieur gauche du bouton), on écrira :

```
if event.type == MOUSEBUTTONDOWN :
    pos = event.pos
    if (500 <= pos[0] < 520) and (300 <= pos[1] < 310) :
        # instructions
```

Autre exemple : on a une fenêtre de taille 150x150, découpée à la façon d'un jeu de Tic-Tac-Toe (ou Morpion), c'est-à-dire en 9 cases carrées (3x3) de 50 pixels de côté. On part du principe que la case en haut à gauche a pour indice (0,0) et que la case en bas à droite a pour indice (2,2). Si on veut, à partir d'un clic sur la fenêtre, retrouver l'indice d'une case, on écrira :

```
if event.type == MOUSEBUTTONDOWN :
    indice = [event.pos[0]/50, event.pos[1]/50]
```

En effet, la division étant la division entière, un clic par exemple en position (122, 83) donnera l'indice (2,1).

4 Évènements et mouvements d'images

Jusqu'ici nous avons uniquement parlé d'images (ou de `Surface`) qui faisaient la taille de la fenêtre, pour créer un fond, et ce fond a pour vocation d'être fixe. On souhaite maintenant créer des images plus petites, qui seront soit fixes soit mobiles (on considérera comme mobile une image qui peut être amenée à disparaître de sa position initiale).

Dans les deux cas, la première étape consiste à créer l'image à proprement parlé, la seconde à coller l'image à l'aide de la méthode `blit()`.

4.1 Charger ou dessiner une image

Pour créer l'image on a en effet deux solutions, comme pour le fond : on charge une image et on la met à l'échelle si on utilise un fichier image, ou on crée une surface, soit unie (une surface remplie d'une couleur) soit avec des formes dessinées à l'aide de `pg.draw` (voir `http://www.pygame.org/docs/ref/draw.html`).

Si on veut charger une image à partir d'un fichier, il y a une différence notable avec le fond : si on veut que le fond soit pleinement opaque et qu'il remplisse tout l'écran, une image pour représenter un objet peut être plus petite et surtout utiliser de la transparence pour ne représenter que l'objet lui-même avec sa forme particulière. Ainsi, le format d'image `png` (Portable Network Graphics) prenant en compte la transparence sera privilégié pour cette utilisation. En cas de d'utilisation d'un fichier image avec transparence, il faut quand même le préciser dans votre code, à l'aide de la méthode `convert_alpha()`. Une image avec transparence sera donc charger ainsi :

```
image = pg.image.load("mon_image.png").convert_alpha()
image = pg.transform.scale(image, (50,50))
```

avec une potentielle mise à l'échelle si nécessaire (ici on veut que l'image fasse 50 pixels sur 50 pixels). Sans `convert_alpha()` les zones transparentes seront remplacées par des zones blanches.

4.2 Coller une image fixe

Reprenons l'exemple du Tic-Tac-Toe. Dans ce jeu, lorsqu'une croix ou un cercle sont placés sur une case, c'est définitif : la croix ne va pas se déplacer sur une autre case !

On peut donc coller l'image sur le fond `background` puis on recolle `background` sur la fenêtre `screen`.

```
# on colle l'image a une certaine position
background.blit(image, position)
# on recolle le fond sur la fenetre en position (0,0)
screen.blit(background, (0,0))
```

L'image `background` représentera alors dans toute la suite du programme l'image de départ avec les autres images qui ont été collées dessus. La valeur `position` correspond aux coordonnées (x, y) auxquelles vous voulez coller le coin supérieur gauche de votre image `image`.

L'utilisation d'un fond ici peut paraître inutile, on pourrait en effet coller le fond au tout début et ensuite coller l'image directement sur la fenêtre `screen` sans passer par le fond. Cependant, dans un programme mêlant images fixes et mobiles, l'utilisation de ce fond est utile et même nécessaire.

4.3 Coller une image mobile

Comme dit précédemment, une image mobile est une image qui peut être amenée à disparaître de sa position initiale. Dans un jeu de Serpent par exemple, le serpent lui-même est évidemment mobile, mais la proie aussi : elle ne se déplace pas mais une fois mangée elle disparaît de sa position initiale et une nouvelle proie apparaît ailleurs.

Dans ce contexte, si on suit le même raisonnement que pour une image fixe, c'est-à-dire coller l'image sur le fond puis le fond sur la fenêtre, alors on aurait l'image à la nouvelle position mais aussi à l'ancienne position. En effet, comme dit précédemment, une image collée l'est de façon définitive, donc le fond contient de façon définitive l'ancienne image et la nouvelle. L'astuce consiste à coller l'image sur la fenêtre et non pas sur le fond pour garder le fond vierge. Ainsi, visuellement on voit le fond avec l'image par-dessus, mais les deux éléments sont indépendants.

on colle `background` sur `screen` puis on colle directement l'image sur `screen` pour que le fond reste vierge. L'intérêt est que quand on bougera l'image de place, on recollera le fond pour "effacer visuellement" l'ancienne image et on recollera l'image à sa nouvelle position.

```
# on recolle le fond sur la fenetre en position (0,0)
screen.blit(background, (0,0))
# on colle l'image a une certaine position
screen.blit(image, position)
```

4.4 Évènements et images

On veut maintenant lier la gestion d'évènements avec les images : faire apparaître une image lors d'un clic, déplacer une image avec les flèches, etc. Pour cela, des exemples valent

mieux que de longs discours.

Reprenons l'exemple du Tic-Tac-Toe. On a une fenêtre de taille 150×150 et on veut placer une croix sur une case en utilisant le clic de la souris. On suppose qu'on a un fichier `croix.png` (avec d la transparence, seule la croix elle-même est opaque) pour représenter notre croix.

```
# code d'initialisation de la fenetre et du fond
# ...

# on charge l'image en prenant en compte la transparence
croix = pg.image.load("croix.png").convert_alpha()
croix = pg.transform.scale(croix, (50,50))

play=True
while play:
    for event in pg.event.get():
        if event.type == MOUSEBUTTONDOWN:
            position = [event.pos[0]/50*50, event.pos[1]/50*50]
            background.blit(croix, position)
            screen.blit(background, (0,0))
            pg.display.flip()
pg.quit()
```

La ligne

```
position = [event.pos[0]/50*50, event.pos[1]/50*50]
```

calcule bien la position du coin supérieur gauche de la croix : la division entière par 50 d'un nombre entre 0 et 149 donne 0, 1 ou 2, et la multiplication par 50 donne donc bien 0, 50 ou 100. Si on clique en position (122, 83), le coin de la croix se collera en coordonnées (100,50).

À noter : la mise à jour de l'affichage (`pg.display.flip()`) ne se fait que lorsqu'il y a un clic. En effet, ça ne sert à rien de mettre à jour l'affichage si aucune modification n'a été apporté, pire, le programme va faire beaucoup d'opérations en continu pour rien ce qui va le ralentir.

Pour cet exemple, il faudrait évidemment changer de joueur après chaque clic (alterner croix et rond) et surtout vérifier que la case cliquée n'est pas déjà occupée.

Autre exemple, mouvement de la barre dans un jeu de casse-briques (le but étant de casser des briques avec une balle qui rebondit sur les murs et de la renvoyer vers les briques à l'aide d'une barre qui bouge uniquement vers la gauche ou vers la droite). On suppose qu'on a une fenêtre de taille 500×700 et que la barre est une surface rectangulaire. On veut donner une idée de comment déplacer la barre en utilisant les flèches sans donner le code complet : au lieu de gérer "tant qu'on presse la flèche gauche on va à gauche", on montre juste comment gérer "si j'appuie sur la flèche gauche, je bouge d'un cran à gauche".

```
# code d'initialisation de la fenetre et du fond
# ...

# on cree une image pour representer la barre
barre = pg.Surface( (100, 25) )
barre( (255, 255, 255) )

# on stocke la position de la barre dans une variable
pos_barre = [200, 650]

play=True
```

```
while play:
    for event in pg.event.get():
        if event.type == KEYDOWN:
            if event.key == K_RIGHT:
                pos_barre[0] += 1
            elif event.key == K_UP:
                pos_barre[0] -= 1
            screen.blit(background, (0,0))
            screen.blit(barre, pos_barre)
            pg.display.flip()
pg.quit()
```

D'après ce code, si on presse la flèche gauche (resp. droite), la coordonnée en abscisse de la barre est diminuée (resp. augmentée) de 1. Si une touche a été pressée, n'importe laquelle, on met à jour l'affichage. Logiquement on devrait mettre à jour l'affichage si on presse une des deux flèches et non pas n'importe quelle touche, mais ici ce n'est pas grave si on suppose qu'on presse uniquement l'une des deux flèches.

Pour compléter l'exemple, il faudrait prendre en compte le fait de rester appuyé, en utilisant par exemple l'évènement `KEYUP` qui vérifie si une touche est relâchée, ou encore `pg.key.get_pressed()` qui retourne une liste de booléens pour chaque touche (si la flèche gauche est pressée, `pg.key.get_pressed()[K_LEFT]` vaudra `True`, et `False` sinon).

5 Les sprites

Les sprites sont des objets ayant des interactions avec le joueur mais aussi entre eux, et c'est pour ce dernier point qu'ils sont le plus utiles. En effet, tester si deux objets rentrent en contact est beaucoup plus aisé avec des sprites qu'avec des images. De plus, les sprites sont des objets, c'est-à-dire ils sont définis par des classes, et peuvent avoir plusieurs attributs : une image, une position, des points de vies si c'est un personnage, etc.

La documentation sur le sujet se trouve ici :
`http://www.pygame.org/docs/ref/sprite.html`
mais n'est pas des plus clair au premier abord, on se propose donc dans cette section de s'intéresser à la création et à l'utilisation des sprites.

5.1 Création de sprites

Comme nous l'avons dit les sprites sont définis par des classes, il faut donc, comme pour tout objet, définir le constructeur `__init__`. La documentation Pygame sur les sprites donne le bout de code suivant, où le sprite considéré est un bloc :

```
class Block(pygame.sprite.Sprite):

    # Constructor. Pass in the color of the block,
    # and its x and y position
    def __init__(self, color, width, height):
        # Call the parent class (Sprite) constructor
        pygame.sprite.Sprite.__init__(self)

        # Create an image of the block, and fill it with a color.
        # This could also be an image loaded from the disk.
        self.image = pygame.Surface([width, height])
        self.image.fill(color)
```

```
        # Fetch the rectangle object that has the dimensions of the image
        # Update the position of this object by setting the values
        # of rect.x and rect.y
        self.rect = self.image.get_rect()
```

Ce code contient les trois points nécessaires à tout constructeur de sprites :

— l'héritage de la classe Sprite de Pygame : en effet, Pygame contient une classe Sprite, à partir de laquelle on peut créer nos propres sprites. On utilise pour cela la notion d'héritage, qui se fait en modifiant une classe à deux endroits :
  — à la déclaration de la classe : on écrit `class Notre_classe(Classe_parente)` au lieu de `class Notre_classe()` (ou au lieu de `class Notre_classe(object)`), pour préciser de quelle classe la nouvelle est héritée ;
  — au début du constructeur : on appelle le constructeur de la classe parente :

```
def __init__(self, arguments):
    Classe_parente.__init__(self)
    # ...
```

— l'attribut `image` : il est obligatoire et doit s'écrire impérativement `image`, c'est un attribut existant dans la classe Sprite de Pygame. Il permet de gérer l'image de votre sprite.

— l'attribut `rect` : idem que l'attribut `image`, il faut respecter le nom de cet attribut hérité de la classe Sprite. Il représente le rectangle dans lequel se trouve le sprite. On peut ainsi s'en servir pour stocker à la fois la position du sprite (`mon_sprite.rect.x` donne la position en abscisse, `mon_sprite.rect.y` donne la position en ordonnée) et la taille du rectangle (`mon_sprite.rect.w` donne la largeur du rectangle et `mon_sprite.rect.h` la hauteur).

L'appel `self.rect = self.image.get_rect()` est nécessaire et permet de déterminer le rectangle (le plus petit en largeur et en hauteur pouvant contenir l'image) à partir de l'image définie juste avant. La position par défaut du rectangle est (0,0) et peut être modifiée en affectant des valeurs à l'attribut `self.rect.topleft` pour la position du coin supérieur gauche du sprite ou à l'attribut `self.rect.center` pour la position du centre de l'image du sprite.

D'autres attributs peuvent être ajoutés, propres à l'objet que l'on veut représenter, comme pour n'importe quelle classe.

Un peu plus loin dans l'héritage

Les classes héritées d'autres classes possèdent par défaut les mêmes méthodes que les classes dont elles héritent : si une classe A possède une méthode `ma_methode`, alors une classe B héritée de A possèdera, sans avoir besoin de la définir, la méthode `ma_methode`. À noter qu'il est possible de définir `ma_methode` pour la classe B (on peut vouloir un comportement légèrement différent pour les objets de la classe B par rapport aux objets plus généraux de la classe A), celle-ci sera appelée en priorité avant la méthode héritée de A (on parle de surcharge de méthode).

Ainsi, la classe Sprite de Pygame possède plusieurs méthodes qui seront héritées par toutes les classes que l'on créera à partir de celle-ci. L'une d'elle est même spécifiquement prévu pour être surchargée : dans la classe Sprite, la méthode `update` ne fait rien, mais est prévue pour mettre à jour les différents sprites hérités, chacun d'une manière qui lui est propre. Les méthodes existantes dans la classe Sprites de Pygame et héritées par tout

sprite personnalisé peuvent se trouver dans la documentation à l'adresse suivante : `http://www.pygame.org/docs/ref/sprite.html`

5.2 Groupes de sprites

La notion de groupe est très importante dans l'utilisation des sprites, elle permet en effet de regrouper différents objets de type sprites et de manipuler tous les sprites d'un même groupe en une seule fois plutôt qu'en faisant la manipulation souhaitée pour chaque sprites.

Nous allons décrire ici comment utiliser un groupe pour chaque classe héritée de la classe Sprite. En effet, chacune de ces classes peut avoir son propre groupe. Il est possible de créer des groupes mêlant n'importe quels sprites mais nous nous focaliserons dans ce tutoriel uniquement aux groupes associés à une classe.

Pour cela, il est nécessaire, à la création d'une classe de sprites, de préciser que toute instance (tout sprite) de cette classe sera automatiquement inclue dans le groupe propre à cette classe. Ceci se fait en modifiant légèrement le constructeur de la classe :

```
class Block(pygame.sprite.Sprite):

    # Constructor. Pass in the color of the block,
    # and its x and y position
    def __init__(self, color, width, height):
        # Call the parent class (Sprite) constructor
        pygame.sprite.Sprite.__init__(self, self.groups)
        # ...
```

L'ajout de `self.groups` suffit pour dire "chaque objet de type `Block` sera inclu dans le groupe de la classe `Block`". Ou presque... En effet, il faut définir au préalable ce qu'est `self.groups`, cela peut être une liste de groupe, ou un seul groupe.

Pour associer tous les objets de type `Block` (la classe de l'exemple ci-dessus) à un groupe propre à cette classe, on va écrire, dans le corps du code principal, la ligne suivante :

```
Block.groups = pg.sprite.Group()
```

pour préciser que le groupe associé à la classe est un nouveau `Group`. À partir de maintenant, chaque fois qu'une instance de la classe `Block` sera créée, elle sera automatiquement ajoutée au groupe `Block.groups`.

Il existe de nombreuses méthodes associées aux groupes, mais deux d'entres elles sont particulièrement utiles : `update` et `draw`. La méthode `update` va appeler, pour chaque sprite contenu dans le groupe, la méthode `update` du sprite. En effet, comme nous l'avons vu précédemment, la classe `Sprite` de Pygame prévoit une méthode `update` qui ne fait rien mais qu'il est généralement nécessaire de re-définir selon les besoins.

Par exemple, si on veut bouger un sprite verticalement à chaque instant d'un pixel vers le bas, on va définir la méthode `update` de la classe de ce sprite de façon à augmenter sa position en ordonnée de 1, et on appellera dans la boucle de jeu la méthode `update` du sprite. Maintenant, si on veut faire ce traitement pour tout un ensemble de sprites issus de la même classe, on va appeler la méthode `update` pour le groupe en faisant `Ma_classe.groups.update` et cette méthode ira appeler elle-même, pour chaque sprite du groupe, la méthode `update` qui a été redéfinie précédemment.

En fait, la ligne de code `Ma_classe.groups.update()` est équivalente au code suivant :

```
for sprite in Ma_classe.groups.sprites():
    sprite.update()
```

où la méthode `sprites()` d'un groupe renvoie la liste des sprites contenus dans ce groupe.

La méthode `draw` quant à elle prend en argument un objet de type `Surface` (par exemple l'écran de la fenêtre Pygame ou encore le fond si on en a créé un) et dessine (`blit`) tous les sprites du groupe sur cette surface. Si `background` est un objet de type `Surface`, alors la ligne de code `Ma_classe.groups.draw(background)` est équivalente au code suivant :

```
for sprite in Ma_classe.groups.sprites():
    background.blit(sprite.image, sprite.rect)
```

On peut alors constater que la méthode `draw` a besoin des attributs `image` et `rect` des sprites (respectivement l'image et le rectangle incluant la position des sprites) du groupe pour les afficher. C'est pourquoi il est important, dans la définition d'une classe issue de classe `Sprite` de Pygame, de définir les attributs `image` et `rect` en respectant leurs noms.

Groupe à un élément : Il existe en effet un groupe particulier qui ne peut contenir qu'un seul sprite d'une classe : le `GroupSingle`. Il est utile lorsqu'on veut, pour un sprite n'existant qu'en un seul exemplaire dans un jeu, le gérer en utilisant les propriétés et méthodes des groupes (par exemple `update` et `draw`). Pour une classe associée, le premier sprite de cette classe sera inclu dans ce groupe, et chaque nouvel objet remplacera le précédent dans le groupe.

Par exemple, dans un jeu de casse-brique, il y a de nombreuses briques, une ou plusieurs balles, mais généralement une seule barre. Cette barre peut donc être gérer par un `GroupSingle` plutôt que par un `Group` classique. Quand une vie est perdue, plutôt que de repositionner la barre en lui retirant les éventuelles transformations subies par des bonus (agrandissement ou rétrécissement de la barre par exemple), on crée une nouvelle barre qui supprimera automatiquement la première et se positionnera directement à l'emplacement par défaut.