Python – Interface graphique avec Pygame

Utilisation de bibliothèques

Python, comme tous les langages de programmation, fourni un certain nombre de commandes et de fonctions de façon native. Les multitudes d'utilisations possibles des langages font que ces fonctionnalités sont rarement suffisantes pour couvrir tous les cas. C'est pour cela qu'il existe des extensions, appelées **bibliothèques**, qui permettent d'enrichir le langage sans obliger l'utilisateur à reprogrammer toutes ces nouvelles fonctionnalités. Par exemple, la bibliothèque `math` rajoute la plupart des fonctions et constantes nécessaires pour faire des calculs poussés et `random` rajoute les fonctions permettant de faire des tirages au sort ou la génération de nombres aléatoires.
Il existe plusieurs façon d'importer des bibliothèques dans Python. On peut par exemple importer toutes les fonctions et constantes définies dans une bibliothèque :

```
>>> from math import *
>>> sqrt(45)
6.708203932499369
```

Le problème de cette technique c'est que lorsqu'on importe plusieurs bibliothèques, il se peut que la définition d'une fonction soit écrasée par celle d'une autre. Par exemple, les bibliothèques `math` et `turtle` définissent toutes les deux la fonction `radians`, sauf que dans la première, cette fonction prend un argument, alors qu'elle n'en prend aucun dans la deuxième.

```
>>> from turtle import *
>>> from math import *
>>> radians(10)
0.17453292519943295
```

Alors que :

```
>>> from math import *
>>> from turtle import *
>>> radians(10)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: radians() takes 0 positional arguments but 1 was given
```

Lorsqu'on importe plusieurs bibliothèques, il vaut donc mieux utiliser la technique de base :

```
>>> import math
>>> math.sqrt(45)
6.708203932499369
```

Lorsqu'on fait cela, il faut explicitement mettre le nom de la bibliothèque devant le nom de la fonction. Cela rend l'écriture un peu plus lourde, mais cela évite tout problème de bibliothèques qui utiliseraient le même nom pour des objets différents. C'est la méthode qui doit être privilégiée, sauf si on n'utilise qu'une seule bibliothèque.
Si le nom de la bibliothèque est un peu long, on peut utiliser un nom plus court ou plus simple :

```
>>> import random as rd
>>> rd.random()
0.30741771725935685
```

Enfin, si on ne veut utiliser que quelques fonctions d'une bibliothèque, on peut choisir de n'importer qu'elles :

```
>>> from math import sin, cos
>>> sin(5) + cos(5)
-0.6752620891999122
```

La bibliothèque Pygame

Jusqu'à présent nous avons utilisé uniquement le mode textuel de Python. Mais il est également possible de faire des interfaces graphiques. Pour cela, il existe différentes bibliothèques telles que tkinter ou wxpython. Nous allons plutôt utiliser Pygame qui est spécialisée pour faire des jeux.
Toutes les interfaces graphiques fonctionnent de la même manière. Elles reposent sur une boucle infinie (enfin jusqu'à ce qu'on quitte l'application) :

- On observe les **évènements** produits par les **périphériques d'entrée** (clavier, souris, manette, micro, caméra...).
- On met à jour les variables (positions, vie, score...).
- On produit la nouvelle image à afficher et éventuellement le son associé.

Chaque tour de boucle correspond donc à une image affichée. Il y en a général 30 ou 60 par seconde. Pour que l'application soit réactive, il faut que le temps qui s'écoule entre l'action de l'utilisateur (cliquer sur un bouton) et le moment où la réaction est affichée à l'écran soit le plus court possible. Il faut également que le rythme auquel sont affichées les images soit stable. C'est pour cela qu'il faut souvent optimiser les calculs pour garantir un rythme constant. Pour ce que nous allons faire, ce ne sera pas un problème.

Un petit jeu

Afin de comprendre les bases de Pygame, nous allons faire un jeu très simple. Pour pouvoir réaliser l'activité, il faut placer le fichier `smiley.png` dans le même dossier que le fichier Python que vous allez créer.

```
import random # Pour les tirages aleatoires
import pygame # Le module Pygame
import pygame.freetype # Pour afficher du texte
pygame.init() # initialisation de Pygame

# L'image de fond
largeur, hauteur = 800, 600

# On définit la taille de la fenêtre
screen = pygame.display.set_mode((largeur, hauteur))

# On met le nom de la fenêtre
pygame.display.set_caption("Clic moi si tu peux")
```

```python
# On récupère l'image du smiley et on obtient sa taille
smiley = pygame.image.load("smiley.png").convert_alpha()
l_smiley, h_smiley = smiley.get_size()

# Pour le texte.
pygame.freetype.init()
taille_texte = 20
myfont = pygame.freetype.SysFont(None, taille_texte)

# Pour controler le nombre d'images par seconde
clock = pygame.time.Clock()

# Position de smiley
x_smiley = random.randint(0, largeur-l_smiley)
y_smiley = random.randint(0, hauteur-h_smiley)

score = 0
continuer = True

# Boucle principale
while continuer:
    # On remplit l'image de noir
    screen.fill((0, 0, 0))

    # Gestion des évènements
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            continuer = False  # Pour quitter
        elif event.type == pygame.MOUSEBUTTONDOWN: # Clic de souris
            x, y = pygame.mouse.get_pos() # coordonnees du pointeur de la souris
            if (x_smiley <= x < x_smiley+l_smiley
                and y_smiley <= y < y_smiley+h_smiley):

                # On bouge le smiley
                x_smiley = random.randint(0, largeur-l_smiley)
                y_smiley = random.randint(0, hauteur-h_smiley)
                score += 1

    texte, rect = myfont.render(f"score : {score}", (255, 255, 255))
    rect.topleft = (20, 20)
    screen.blit(texte, rect)

    screen.blit(smiley, (x_smiley, y_smiley))
    # On affiche le resultat final a l'ecran
    pygame.display.flip()
    # On attend le temps necessaire pour avoir un FPS constant
    clock.tick(30)

# On ferme la fenêtre
pygame.quit()
```

Exercice 1 : Recopier le programme ci-dessous et le tester.

Explications

La première partie du programme consiste à définir les valeurs et objets qui seront nécessaires pour le jeu. Par exemple `screen` correspond à la surface qui contient l'image finale, `smiley` est celle qui contient le dessin et `myfont` correspond à la police utilisée pour écrire le texte.
Pour connaître les fonctions disponibles, vous pouvez aller voir la documentation officielle https://www.pygame.org/docs/.
Dans la documentation sur les surfaces, on trouve :

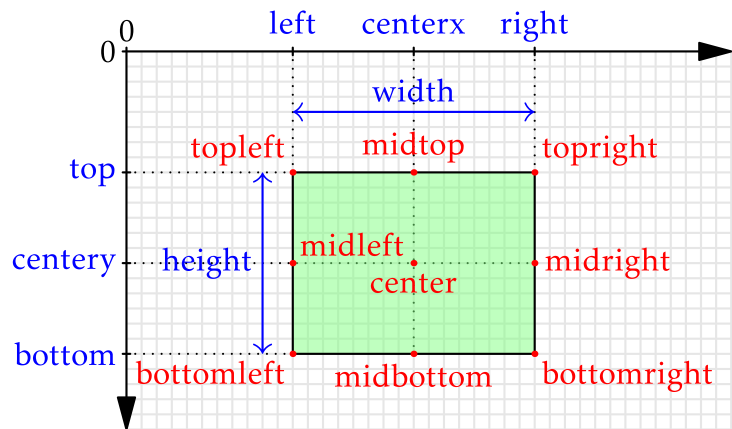
> `pygame.Surface` :
>
> *pygame object for representing images*
>
> `Surface((width, height), flags=0, depth=0, masks=None) -> Surface`
> `Surface((width, height), flags=0, Surface) -> Surface`
>
> `pygame.Surface.blit`
> draw one image onto another
>
> `pygame.Surface.convert_alpha`
> change the pixel format of an image including per pixel alphas
>
> `pygame.Surface.fill`
> fill Surface with a solid color

On voit donc qu'une `Surface` est un objet représentant une image et qu'elle est créée en définissant sa largeur et sa hauteur. Parmi les méthodes qui peuvent être utilisées sur la surface, `blit` permet de dessiner une autre image sur la surface, `convert_alpha` permet de prendre en compte le canal alpha (la transparence) de chaque pixel et `fill` permet de remplir la surface avec une couleur. Pour avoir plus d'informations sur ces méthodes, il faut aller voir directement sur la description détaillée.

> `blit()`
>
> draw one image onto another
>
> `blit(source, dest, area=None, special_flags=0) -> Rect`
>
> Draws a source Surface onto this Surface. The draw can be positioned with the dest argument. Dest can either be pair of coordinates representing the upper left corner of the source. A Rect can also be passed as the destination and the topleft corner of the rectangle will be used as the position for the blit. The size of the destination rectangle does not effect the blit.

On apprend ainsi que pour utiliser `blit` il faut lui donner 2 paramètres : la `source` (l'image qu'on veut appliquer sur la surface) et la destination `dest` qui peut être soit les coordonnées de l'angle en haut à gauche de l'image à appliquer par rapport à celui de la surface, ou alors un rectangle, qui est aussi un objet de pygame, et dont on utilisera le coin en haut à gauche pour placer l'image, peu importe ses dimensions.
Ainsi `screen.blit(smiley, (x_smiley, y_smiley))` va placer le smiley sur `screen` aux coordonnées indiquée. Pour afficher le texte, on positionne le rectangle `rect` qui est généré en même temps que l'image du texte en mettant son coin haut gauche aux coordonnées $(20;20)$ et on place ensuite l'image du texte à l'emplacement de ce rectangle.

En allant chercher dans la documentation sur les rectangles, on voit qu'on peut aussi utiliser différents attributs pour placer le rectangle ou pour connaître ses dimensions. Mais dans tous les cas, ce sera l'angle en haut à gauche qui sera utilisé pour placer l'image.
Il faut aussi savoir que l'origine du repère est en haut à gauche, que les abscisses vont vers la droite et les ordonnées vers le bas, comme sur le schéma ci-contre.



Pour comprendre comment comment fonctionne une bibliothèque, il faut donc aller voir les documentations, regarder des exemples et surtout ne pas hésiter à modifier les valeurs pour voir ce qui se passe.

Gestion des évènements

Lorsqu'un utilisateur effectue une action sur un périphérique d'entrée, il y a un évènment qui est généré par le système. Cet évènement peut être utilisé par l'application pour réagir à l'action. Les évènements peuvent être l'appui sur une touche, un mouvement de la souris, le fait de cliquer sur le bouton pour fermer l'application... Dans notre exemple, nous utilisons deux évènements, identifiés par leur type `QUIT` et `MOUSEBUTTONDOWN`. Le premier est généré lorsqu'on essaie de fermer la fenêtre et le second lorsqu'on clique sur un bouton de la souris. Lorsqu'on relâche ce bouton, il y a un évènement `MOUSEBUTTONUP` qui est généré.
Si on veut distinguer les clics fait avec le bouton gauche, droit ou milieu de la souris, il faut regarder les attributs supplémentaires de l'évènement. Ainsi, dans le cas d'un évènement `MOUSEBUTTONDOWN` ou `MOUSEBUTTONUP`, `event.button` vaut 1, 2 ou 3 selon le bouton utilisé.

Exercice 2 : En rajoutant un `print(event.button)` au bon endroit, déterminer à quel valeur correspond chaque bouton de la souris. Le fait de tourner la molette du milieu vers le haut ou le bas est également considéré comme des boutons différents.

Vous pourrez remarquer que l'évènement `MOUSEBUTTONDOWN` n'est envoyé qu'une fois par clic et que même si le bouton est gardé appuyé, l'évènement n'est pas renvoyé. Si vous voulez utiliser le fait que le bouton est gardé appuyé, il faut utiliser une variable qui vaudra `True` tant que vous n'aurez pas vu l'évènement `MOUSEBUTTONUP` correspondant au bon bouton.
De façon analogue, les évènements `KEYDOWN` et `KEYUP` servent à savoir lorsqu'on appuie sur une touche du clavier. La touche correspondant peut être obtenue avec `event.key` et vaut `K_a`, `K_b`... pour les touches du clavier ou encore `K_UP` ou `K_LEFT` pour les flêches. Chaque touche du clavier a son propre code.
Pour revenir au jeu, savoir si un bouton de la souris est pressé ne suffit pas à savoir si le joueur a cliqué sur l'image. Il faut aussi savoir si la souris se trouve au dessus de l'image. Pour cela, on obtient les coordonnées de la souris et on regarde si ces coordonnées se trouvent entre les coordonnées du coin haut gauche et celui du coin bas droite de l'image. Si c'est le cas on met à jour les coordonnées de l'image. Cela veut dire que la zone testée est un rectangle et pas un cercle. Les zones de collisions dans les jeux sont généralement des rectangles car cela simplifie grandement les calculs. Il est possible de faire des collisions plus fines, mais cela demande plus de travail.

Exercice 3 : Vous avez dû remarquer que le score reste à 0 quoi qu'il arrive. Rajoutez un point au score à chaque fois que l'image est cliquée.

Gestion de l'affichage

Comme nous l'avons déjà vu, les images, ou parties d'images, sont gérées par des surfaces. Ces surfaces peuvent ensuite être "appliquées" les unes sur les autres, comme des autocollants. Si une surface est plus grande ou dépasse de la surface sur laquelle elle est appliquée, les pixels en trop ne sont pas ajoutés à l'autre surface. Par contre, si on applique plusieurs surfaces à la suite, il faut faire attention à l'ordre. Dans le jeu, le texte est appliqué avant l'image pour qu'elle soit visible même si elle se trouve sur le texte.
Au moment de produire l'image finale, il faut que tout doit être visible ait été appliqué sur la surface principale, `screen` et il faut utiliser la commande `pygame.display.flip()` pour envoyer cette image à la carte graphique. La commande `clock.tick(30)` fait une pause pour garantir que la boucle ne durera pas moins qu'un trentième de seconde, afin d'assurer 30 images par seconde.

Exercice 4 : Commentez la commande `screen.fill((0, 0, 0))` et testez le jeu.

La surface `screen` n'est pas réinitialisée après chaque tour de boucle. C'est pourquoi il faut explicitement réinitialiser cette surface, ici en la remplissant de noir, pour assurer que l'image à cliquer ne sera pas affichée à plusieurs endroit si elle bouge. Il serait possible de ne mettre cette réinitialisation qu'après un clic réussi, mais c'est une bonne habitude à prendre de le faire à chaque nouvelle image.

Exercice 5 : Modifiez le jeu pour qu'il soit sur fond blanc et que le texte soit écrit en noir.

Pour aller plus loin

Voici quelques suggestions pour améliorer le jeu.

Exercice 6 : Vous pouvez mettre une pénalité en cas de clic en dehors de l'image.

Exercice 7 : Vous pouvez déterminer un score à atteindre à partir duquel la partie se termine. À ce moment là, le jeu peut se terminer ou alors un message est affiché et un nouveau clic réintialise la partie. Pour afficher un texte plus grand, vous pouvez rajouter le paramètre `size=k` à la commande `myfont.render`, en mettant une valeur à la place de `k` (par exemple 30 ou 40).

Exercice 8 : Vous pouvez ajouter un chronomètre afin d'essayer d'attendre le score fixé le plus vite possible ou au contraire un compte-à-rebours indiquant le temps restant pour faire le plus grand score possible. Vous pouvez utiliser `debut = pygame.time.get_ticks()` pour obtenir le temps au début et `pygame.time.get_ticks() - debut` pour obtenir le nombre de millisecondes s'étant écoulées depuis ce début.

Exercice 9 : Vous pouvez modifier le jeu pour que les smileys cliqués restent à l'écran et rendent le jeu plus difficile.

Exercice 10 : Vous pouvez diminuer progressivement la taille du smiley tant qu'on ne clique pas dessus, auquel cas, il reprend sa taille initiale. S'il devient trop petit, c'est la fin de la partie. Vous pouvez utiliser les lignes suivantes pour changer la taille du smiley d'un certain coefficient (`COEFF`), à mettre à la place de la ligne avec `blit` :

```
    smiley_zoom = pygame.transform.smoothscale(smiley,
                                           (l_smiley*COEFF, h_smiley*COEFF))
    screen.blit(smiley_zoom, (x_smiley, y_smiley))
```

Exercice 11 : On peut transformer le jeu en *snake* en utilisant une liste pour stocker les positions précédente du smiley. Pour avancer, il faut rajouter la nouvelle position au début de la liste et enlever le dernière élément.

```
>>> liste = [4, 6]
>>> liste.insert(0, 8)   # pour insérer au début
>>> liste
[8, 4, 6]
>>> liste.pop()   # pour supprimer le dernier élément.
6
```