Structure de base et initialisation

Structure de la boucle d'exécution

Le programme aura la structure de base suivante

```
import pygame
screen=pygame.display.set_mode((1024,768))
clock=pygame.time.Clock()
running=True
while running:
      dt=clock.tick(5)
      [event]
      [update]
      [display]
      pygame.display.flip()
```

La valeur (1024,768) désigne le mode écran dans lequel on veut travailler (ou la taille de la fenêtre) L'objet clock permet de s'assurer que la boucle s'exécute avec un délai constant, pour avoir le nombre de FPS correspondant au paramètre de tick(). La variable dt indique le temps qui se sera effectivement écoulé.

La commande flip() permute l'écran affiché et celui sur lequel on dessine et elle est quasi immédiate.

La section event permettra de gérer les événements (clic souris, appui bouton,…
La section update de calculer les nouvelles positions
la section display d'effectuer l'affichage à l'écran

Gestion d'événements

Pour gérer les événements, on fait une boucle qui récupérera les événements un à un. Une série de if permettra ensuite de tester les différents types d'événements : fin du programme (on a cliqué sur le [x] pour fermer la fenêtre), appui d'une touche, déplacement de la souris,...

```
 for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
```

L'événement aura des propriétés qui dépendent du type d'événement. Par exemple, KEYDOWN et KEYUP auront des propriétés key, mod (shift/ctrl/alt), unicode, scancode (position sur le clavier), MOUSEBUTTONUP et MOUSEBUTTONDOWN auront pos (la position (x,y) de la souris) et button (le numéro du bouton),…

Même si on utilise les fonctions de pygame.mouse et pygame.key pour gérer souris et clavier, il est nécessaire de traiter les événements avec le pygame.event.get().

On peut récupérer toutes les constantes (QUIT, KEYDOWN,… et bien d'autres) sans avoir besoin de les préfixer (pygame.QUIT,…) en utilisant l'import suivant :

```
from pygame.locals import *
```

Update

Dans la phase update, on effectuera les différents calculs : déplacements, détection de collision, …

La valeur dt permet de connaître le temps qui s'est effectivement écoulé depuis le tick() précédent. Pour des mouvements à vitesse constante, on multipliera la vitesse par dt pour calculer le déplacement. Ainsi, si par moment le nombre de FPS ne sait pas être atteint (programme ralenti), le déplacement sera fait sur une distance plus grande pour compenser.

Display

Dans la phase display, on mettra l'écran à jour, "couche par couche". On commence par l'arrière plan et on continue avec les éléments (joueur, monstres, éléments que l'on peut manipuler) en commençant par ceux qui apparaissent derrière et en terminant par ceux qui apparaissent au premier plan.

Il est courant de commencer par un screen.fill("#FFFFFF") qui permettra de mettre l'écran à une "page blanche" (ou plutôt, de la couleur précisée en paramètre). La couleur peut être transmise de plusieurs façons :

- (r,g,b) ou (r,g,b,a)
- [r,g,b,a]
- "#rrggbb" ou "#rrggbbaa"
- "green", "purple",… (attention, les noms sont parfois trompeurs, à éviter).

La mise à jour de l'écran se terminera par le pygame.display.flip() qui permute écran de travail (celui sur lequel on dessine) et écran affiché.

On peut tracer des formes à l'aide des fonctions de pygame.draw comme

- pygame.draw.rect(screen,color,(x,y,w,h))
- pygame.draw.circle(screen,color,(x,y),r)
- pygame.draw.line(screen,color,(sx,sy),(ex,ey),width=1)

On peut aussi coller d'autres surfaces (images, texte,…)

Gestion Clavier/Souris/Joystick

clavier/souris

La méthode la plus propre est d'utiliser les événements pour déterminer les actions et de les traiter dans la partie Update.

On pourra utiliser des variables indiquant ce qui se passe (click + position, direction de déplacement, saut, …) mises à 0 avant la gestion des événements et modifiée par les événements.

```
while running:
      pos=None
      b=0  # No button
```

```
for event in pygame.event.get():
      if event.type == pygame.QUIT:
            running = False

      if event.type == pygame.MOUSEBUTTONDOWN:
            pos = event.pos
            b = event.button
```

Il suffira de tester la variable b pour savoir si un bouton a été enfoncé (1 pour le bouton gauche) et si c'est le cas, on dispose de la position (x,y) dans pos.

Pour les événements claviers, les valeurs de la propriété "key" correspondent à des constantes K_… (K_TAB, K_LEFT, R_SPACE, K_a, K_0, K_F1,...)

joystick

Pour utiliser les joysticks, il est nécessaire d'utiliser un peu de code supplémentaire :

Au lancement, initialiser la bibliothèque et créer les objets Joystick pour les joysticks connectés (avant la boucle principale)

```
pygame.joystick.init()
joysticks = [pygame.joystick.Joystick(x) for x in range(pygame.joystick.get_count())]
```

Ajouter/retirer les joysticks connectés/déconnectés pendant que le programme tourne :

```
            if event.type == pygame.JOYDEVICEADDED:
                joy = pygame.joystick.Joystick(event.device_index)
                joysticks[joy.get_instance_id()] = joy

            if event.type == pygame.JOYDEVICEREMOVED:
                if event.instance_id in joysticks:
                    del joysticks[event.instance_id]
```

Il sera alors possible d'écouter les événements JOYAXISMOTION (un des "axes" a bougé) JOYHATMOTION (un des "chapeaux chinois" a bougé), JOYBUTTONUP/JOYBUTTONDOWN (un des boutons a été enfoncé ou relaché).

Les axes auront des valeurs de -1.0 à 1.0, les événements auront les valeurs "axis" et "value". Les boutons auront un numéro de bouton "button", les chapeaux chinois les valeurs "hat" et "value"

Lecture directe

Il est possible de lire directement l'état d'un bouton (ou une position).

La fonction pygame.key.get_pressed() retourne un pseudo-tableau représentant toutes les touches avec pour chaque touche une valeur True si la touche est enfoncée. On pourra utiliser un code comme

```
if pygame.key.get_pressed()[K_ENTER]:
```

pour déterminer si la touche Enter est enfoncée. De la même manière, pygame.key.get_mods() retournera la liste des modificateurs (KMOD_LSHIFT,…) enfoncés.

On pourra détecter quels sont les boutons de la souris qui sont enfoncés à l'aide de pygame.mouse.get_pressed(). Cette fonction détecte par défaut 3 boutons mais il est possible de lui passer un paramètre 5 pour en détecter 5 (deux "side-buttons" en plus).

Pour récupérer la position de la souris, on utilisera pygame.mouse.get_pos(). On obtiendra la position par rapport à la fenêtre pygame.

On peut aussi demander le déplacement de la souris à l'aide de pygame.mouse.get_rel(). On obtiendra le déplacement (dx,dy) depuis le dernier appel à get_rel(). Cela peut être utile pour détecter des mouvements de souris ("Gesture" comme un mouvement rapide vers la gauche).

Pour un jeu où la souris contrôle directement la position d'un élément du jeu (un "viseur" par exemple), on peut utiliser pygame.mouse.set_visible(True/False) pour afficher/cacher le pointeur de souris. Cela retournera l'état précédent (visible/caché).

Image et Blit

Souvent, on désirera afficher des images (personnages, éléments de décor,…). Pour cela il faudra effectuer deux opérations :

- charger l'image dans une surface. Cela serait fait une fois et on pourra utiliser l'image autant de fois qu'on le désire

- copier tout ou partie d'une surface dans une autre surface. Cette operation s'appelle le blit.

Pour charger l'image on utilisera la fonction

img = pygame.image.load("nom_image.png")

Cette fonction permettra de charger des images JPG, PNG, GIF, BMP, … La surface créée aura les mêmes caractéristiques que l'image sauvée (palette pour GIF/PCX, true color pour JPG ou PNG, transparence pour PNG,…)

Pour accélérer le tracé, on peut convertir les surfaces "images" dans un format correspondant à celui de l'affichage. Si on n'utilise que des images true-color (JPG ou PNG par exemple), cela ne sera généralement pas nécessaire.

Pour mettre les images dans un sous-répertoire, on utilisera os.path.join('img', 'bla.png') comme nom de fichier (qui se chargera de mettre des / ou des \ selon l'OS).

Pour copier une image dans une autre, on utilisera la commande blit()

```
screen.blit(img, dest=(x,y), area=(xs,ys,ws,hs))
```

Le paramètre "area" peut être omis si on désire copier la totalité de l'image. Sinon, il décrit un rectangle dans l'image qui sera collé dans la surface destination.

Les images seront tracées de l'arrière plan à l'avant-plan.

On peut obtenir un rectangle des dimensions de l'image avec img.get_rect(). Il sera positionné en (0,0) mais on peut demander une autre position (avec les mêmes largeurs et hauteurs) en utilisant les paramètres top, bottom, left, right ; topleft, bottomleft, topright, bottomright, center (=(x,y)),…

Par exemple : rect = img.get_rect(topleft=(100,200))

Sprites

Un Sprite est dérivé de la classe Sprite et doit définir les propriétés image (une surface), rect (le rectangle où placer l'image) et une méthode update (mise à jour des caractéristiques : déplacement et autres)

Il est fait pour être placé dans des groupes (Group ou LayeredUpdates) qui permettront d'appeler les update() des sprites membres, de les tracer ou de détecter les collisions.

Sprite => Crée un Sprite. Si on passe un ou des groupe(s) en paramètre, l'ajoute dans ce(s) groupe(s)

update(*args,**kwargs) : les paramètres viennent de l'appel de update sur le groupe

add(group,group,…) : ajoute le sprite aux groupes cités

remove(group,group,…) : retire le sprite des groupes

kill() : retire de tous les groupes

alive() : True si dans au moins un groupe

groups() : liste des groupes

Group =>

sprites() : liste des sprites

copy() : nouveau groupe contenant les mêmes sprites

add(sprite,sprite,…) : ajoute des sprites

remove(sprite,sprite,…) : retire des sprites

has(sprite,sprite,…) : True si tous les sprites sont dans le groupe

update(*args, **kwargs) : appelle les update des sprites du groupe, transmet les paramètres

draw(surface) : trace les sprites sur la surface (utilise rect et image)

clear(surface, bg) : efface les sprites du dernier draw, bg est une surface ou une fonction qui prend une surface et un rectangle et "efface" la zone)

empty() : retire tous les sprites

LayeredUpdates => hérite de Group et permet des layers

add(sprite,layer) : ajoute sprite à un layer

get_sprite_at(pos) : liste des sprites à une position, bottom à top

get_sprite(idx) : récupère un sprite

remove_sprites_from_layer(layer) : retire tous les sprites du layer indiqué

layers() : retourne la liste des layers

change_layer(sprite,layer) : déplace un sprite vers un layer

get_layer_of_sprite(sprite) : retourne le layer du sprite

get_top_layer(), get_bottom_layer() : retourne le numéro du layer au top/bottom

move_to_back(sprite), move_to_front(sprite) : déplace un sprite au bottom/top

get_top_sprite() : retourne le sprite au top

get_sprites_from_layer(layer) : retourne la liste des sprites à un layer donné

switch_layer(layer1,layer2) : échange les sprites des layers 1 et 2

GroupSingle => groupe contenant un seul sprite (pour utilisation de update et draw)

sprite : propriété contenant le sprite unique concerné. On peut changer le sprite en changeant directement cette propriété.

On peut détecter les collisions. Par défaut pygame teste les rect des sprites mais on peut fournir un paramètre collided avec une fonction changeant ce comportement. collide_mask utilisera un masque basé sur la transparence du sprite pour une détection plus fine (mais plus lente)

spritecollideany(sprite,groupe[,collide_mask]) : retourne un sprite si collision, None s'il n'y en a pas. S'arrête à la première collision détectée

spritecollide(sprite, group, dokill [,collide_mask]) : retourne la liste des sprites en collision avec le sprite fourni. Si dokill est True, retire les sprites en collision du groupe.

groupcollide(group1,group2,dokill1,dokill2 [,collide_mask]) : retourne un dictionnaire où, pour chaque sprite de group1 est associée une liste des sprites du group2 en collision.

Tiled et pyTMX

Le programme Tiled (mapeditor.org) permet d'éditer une carte et de la sauver au format TMX. Le format tmx pourra être chargé par la bibliothèque pyTMX.

Il faudra fournir le fichier TMX, les fichiers TSX et les fichiers images associés, avec des chemins relatifs correspondant à ceux des répertoires utilisés avec Tiled (généralement on fait des répertoires tmx, tsx et img et on copie ces trois répertoires dans le projet Python)

Pour travailler avec PyGame, on importera

```
from pytmx.util_pygame import load_pygame
```

On utilisera ensuite

```
tmx_data = load_pygame("xyz.tmx")
```

Pour charger la carte dans tmx_data. load_pygame() se chargera de charger et convertir les images au format de surface PyGame automatiquement. Il faut que pygame ait déjà été initialisé et que l'on ait déjà fait le pygame.display.set_mode()

On peut récupérer un layer à l'aide de tmx_data.get_layer_by_name(nom_layer). Il faudra traiter le layer en fonction de son type.

visible_layers est un itérateur permettant de récupérer uniquement les layers visibles et objects un itérateur retournant tous les objets de la carte.

Les attributs width et height donnent la taille de la carte en tiles et tilewidth et tileheight les dimensions des tiles en pixels.

Les layers "Tiles" ont une propriété "data". On peut utiliser if hasattr(layer,"data") pour les identifier. On pourra alors utiliser les propriétés name, width, height, opacity, visible, offsetx, offsety et itérer sur layer.tiles() pour obtenir x,y,surface_image. Les positions sont en tiles (à multiplier par les dimensions des tiles pour une dimension en pixels)

Pour les objets, on disposera d'un type qui sera soit le type d'élément fixé dans Tiled, soit "Shape" pour les éléments de tracé de base.

Pour les images, outre le type, on disposera également de x,y et image qui fourniront la position en pixels et la surface de l'image.

Pour les formes, on aura le nom de la forme dans name, x et y pour la position, width et height pour la taille (bounding box pour polygon et polyline) et la liste des points (polygon/polyline).

polygon : points = liste des objets points par rapport à (0,0) => (p.x,p.y) , closed=True

polyline : points = liste des points, closed = False

Rectangle, Ellipse : x,y,width et height

Marker : x,y (pas de width ou height)

Les objets texte ne sont pas supportés.

Tiles et objets images pourront être affichés à l'aide du mécanisme de sprites. Pour les formes, on devra utiliser les fonctions draw.rect, draw.circle, draw.ellipse, draw.polygon,…

Sons

Musique de fond

Pygame.mixer.music permet de jouer un morceau en musique de fond sans le charger totalement en mémoire. Le format préféré est OGG mais MP3 est accepté. On peut aussi utiliser des fichiers MOD, XM. IT, S3M (formats tracker).

```
MUSIC_EVENT=pygame.USEREVENT + 1
pygame.mixer.music.set_endevent(MUSIC_EVENT)
pygame.mixer.music.load("fichier.ogg")
pygame.mixer.music.play()
```

Quand le morceau se termine, l'événement MUSIC_EVENT se déclenche (dans la boucle d'event). On peut alors charger le morceau suivant.

Bruitages

Pour jouer des sons, il suffit de les charger dans des objets Sound et d'utiliser la fonction play() de cet objet. Par défaut, 8 sons différents peuvent être joués simultanément. Ce nombre peut être changé par pygame.mixer.set_num_channels()

```
son1 = pygame.mixer.Sound("son1.ogg")
son1.play()
```

On peut utiliser la fonction set_volume() sur un son pour en changer son volume, avec un volume allant de 0.0 à 1.0

Texte

Pour afficher du texte, on génère une surface avec le texte que l'on blit sur l'écran.

```
myfont = pygame.font.Font("myfont.ttf",24)
textsurf=myfont.render("Hello World",True, "#0000FF")
```

La valeur True indique le mode anti-alias et la couleur fournie est la couleur de tracé.

Pour une fonte système, on utilisera pygame.font.Sysfont("Times New Roman",24,bold=False,italic=False). A noter que les fontes système dépendent de l'OS et sont généralement à éviter si on désire un programme cross-platform.

La plupart des fontes sont fournies avec des fichiers séparés pour les gras et italiques.