Construction d'un jeu vidéo tile-based

On se propose de programmer un jeu tile-based avec les particularités suivantes :

- un personnage avec 4 sprites (haut, bas, gauche, droite)
- l'appui sur un touche fleche fait bouger le personnage d'une case dans la direction de la fleche
- la carte est stockée dans un fichier et décrite avec du texte (e.g. `#` pour les murs, `.` pour du vide)
- pour le rendu de la carte, on utilisera des images (tiles) de mur et de vide de 32x32 pixels
- le heros ne peut pas avancer si il y a un mur

Nous pouvons construire progressivement ce jeu avec les différentes étapes, toutes permettant de valider au fur et à mesure le comportement des fonctionnalités.

0. Preparation de l'ecran

Préparer les dimensions de l'écran. Nous pouvons par exemple partir du principe que notre map aura une taille de 10x10 tiles, et que chaque tile fait 32 pixel de largeur. On calcule donc la taille en pixel de l'ecran :

```
largeurMap_enTile = 10
hauteurMap_enTile = 10

tailleTile = 32

largeurMap_enPixels = largeurMap_enTile * tailleTile
hauteurMap_enPixels = hauteurMap_enTile * tailleTile
```

On utilise ensuite ces dimensions lors de la création de notre fenêtre / écran.

1. La classe personnage

1.1 Creation de la classe et chargement d'un premier sprite

Dans un premier temps, faisons en sorte de créer une classe Personnage qui va contenir plusieurs attributs et méthodes qui permettent de gérer notre personnage.

- Créer une classe Personnage avec une methode `__init__` .
- Dans la méthode `__init__` , créer un attribut `sprite` qui contiendra le sprite (image) "haut" du héro.
- Toujours dans `__init__` , faire un `print(self.sprite)` pour vérifier que cette variable contient bien un objet de type image. Si `self.sprite` contient bien une image, cela

devrait afficher quelque chose comme `<Surface(384x32x32 SW)>` .
- Dans le programme principal, initialiser un personnage pour vérifier le bon fonctionnement de la classe jusqu'ici.

1.2 Afficher le personnage sur l'écran

Nous voudrions maintenant donner à notre personnage une position, et afficher le sprite du personnage à cette position sur l'écran.

- Dans `__init__` , initialiser des attributs `x` et `y` correspondants à la position du personnage. Pour le moment, le personnage sera en (0,0).
- Ajouter une méthode `render` (ou `afficher`) qui affiche le sprite du personnage à sa position `(x,y)` sur l' `ecran` .
- Faire appel à cette méthode dans la boucle principale, avant le rafraichissement de l' `ecran` .
- Tester le programme pour vérifier que le personnage s'affiche bien.

1.3 Ajouter une méthode `move` (ou `deplacer`)

Maintenant, nous souhaiterions avoir un moyen de modifier les coordonnées (x,y) du personnage grâce à une fonction `move` .

- Ajouter une méthode `move` (ou `deplacer`) qui prends un argument le nombre de case dont doit bouger le héros en `x` et `y` (par exemple nommée `dx` et `dy`). Par exemple, `move(1,0)` ajoutera 1 à l'attribut `x` du personnage, et 0 à l'attribut `y` .
- Tester cette méthode en apellant `move(1,0)` directement après l'initialisation du personnage.

1.4 Faire le lien entre les touches du clavier et le deplacement du personnage

Maintenans, nous pouvons faire en sorte que l'utilisation des fléches du clavier provoque un appel à la méthode `move` !

- Dans le programme principal, s'assurer d'avoir une fonction `eventHandler()` (ou `gestionEvenements()`) qui :
  - gère l'événement `pygame.QUIT` (et execute `pygame.quit()` et `sys.exit()` si il est déclenché)
  - gère l'événement `pygame.KEYDOWN` et apelle une nouvelle fonction `keysHandler(event.key)` (ou `gestionTouches(event.key)`)
- Ecrire la fonction `keysHandler(key)` : suivant si `key` vaut `pygame.K_UP` , `pygame.K_DOWN` , `pygame.K_LEFT` ou `pygame.K_RIGHT` , cette fonction apelle `perso.move()` avec les arguments adéquats.

1.5 Ajouter une méthode `look` (ou `regarder`)

Notre personnage se déplace maintenant sur l'écran, mais tout cela serait plus joli si le personnage tournait la tête lorsqu'il se déplace ! Nous allons donc maintenant gérer les différentes sprites du personnage, où celui-ci regarde en haut, en bas, à gauche, à droite (*ces soirées là, tintin, tintin~*). Pour ce faire, nous allons introduire une méthode `look` qui permet de changer la sprite actuelle du héros.

Mais avant cela, nous devons déjà charger les différents sprites en mémoire.

- Définir (par exemple, tout en haut du programme) un dictionnaire `chemin_sprites_personnage` qui fait correspondre aux chaines de caractère "haut", "bas", "gauche" et "droite" le chemin de chaque sprite du personnage.
- Dans `__init__` , créer un attribut `sprite` qui est un dictionnaire vide
- Toujours dans `__init__` , parcourir `chemin_sprites_personnage` pour charger chacune des images, et ajouter ces images au dictionnaire `sprite` .
- Ajouter un attribut `direction` qui décrit la direction actuelle dans laquelle regarde le personnage, par exemple initialisée à "haut".
- Modifier la méthode `render` pour bliter `sprite[direction]` comme sprite actuelle.

Maintenant nous pouvons ajouter la méthode `look` :

- Ajouter une méthode `look` (ou `regarder`) qui permet de changer l'attribut `direction` d'apres un chaine de caractères passée en argument (par exemple : `regarder("droite")` change l'attribut `direction` en `"droite"`).
- Tester la méthode `look` en apellant par exemple `perso.look("droite")` directement après l'initialisation du perso
- Ajouter `perso.look("haut")` , `perso.look("bas")` , etc... aux actions à effectuer lorsque l'on appuie sur les touches.

2. La classe map

2.1 Ecriture de la map

Ouvrir un nouveau fichier, par exemple `map.asc` et mettre quelque chose comme :

```
##########
#........#
#........#
#........#
#........#
#........#
#........#
#........#
#........#
##########
```

(Attention : votre fichier doit avoir le meme nombre de ligne et de colonne que les dimensions `largeurMap_enTile` et `hauteurMap_enTile` de l'étape 0)

Chaque `#` et `.` représente ainsi des cases 'mur' et 'vide'

2.1 Création de la classe et chargement du fichier

- Créer une classe Map avec une methode `__init__`
- Dans la méthode `__init__` , ouvrir le fichier `map.asc` et mettre le contenu du fichier (`fichier.readlines()`) dans un attribut `map`
- Faire un `print` de `self.map` pour vérifier ce que contient cet attribut
- Tester cette classe en initialisant un objet de classe 'Map' dans le programme principal
- Modifier la méthode `__init__` pour que le nom du fichier (e.g. `map.asc`) soit passé en argument.

2.2 Chargement des tiles

Nous souhaitons maintenant charger également en mémoire les images (tiles) qui vont servir à créer le rendu de la map.

- Créer un dictionnaire `tile_paths` qui à `#` et `.` associe le chemin des images qui seront utilisées.
- Dans la fonction `__init__` , créer un nouveau attribut dictionnaire `tile_images` , parcourir le dictionnaire `tile_paths` , charger les images, et ajouter ces images au dictionnaire `tile_images` .
- Tester (avec des print) que ceci semble fonctionner

2.2 Affichage de la map

Nous voulons maintenant utiliser ces images et la map ascii pour afficher la map à l'ecran.

- Créer une méthode `render` (ou `afficher`)
- Cette méthode va parcourir toutes les coordonnées (x,y) de la map, et à chaque fois, afficher l'image correspondante au caractère ascii (`#` : une image de mur, `.` : du vide). Nous devons donc aller de (x,y) = (0,0) jusqu'à (9,9), en passant par `(0,9)` , `(1,3)` , `(4,8)` , etc..
- Pour ce faire, on peut utiliser une double boucle :

```
for x in range(10) :
    for y in range(10) :
        ...instructions...
```

- Dans cette boucle, récupérer le caractère à la coordonnée x,y dans `self.map` : `self.map[y][x]`
- Récupérer l'image associée à ce caractère dans le dictionnaire `tile_images`
- Blitter cette image sur l'écran aux coordonnées (x,y) (Attention, comme x,y sont a priori en 'nombre de cases', il faudra probablement les convertir en 'nombre de pixels')
- Tester cette méthode en appelant `laMap.render()` dans la boucle principale.

3. Gestion des collisions

Vous devriez maintenant avoir constaté que votre personnage se déplace par-dessus les murs ! Nous voudrions donc qu'il ne puisse par les traverser ... Pour ce faire, dans la méthode `move()` du personnage, nous allons demander à la map si la case de destination est vide, avant de bouger !

- Ajouter à la classe Map une méthode `isWalkable(x,y)` (ou `onPeutMarcherSur(x,y)`) qui renvoie `True` si la case `x,y` correspond à du vide (caractère `.`), et `False` sinon.
- Dans la méthode `move()` du personnage, avant d'effectuer le déplacement (modification des attributs x et y), véifier quer `laMap.isWalkable(x+dx,y+dy)` vaut bien `True` .
- Tester !