

Partie II : Interactivité avec la carte

I/ La bibliothèque Arcade

La bibliothèque **Arcade** est une alternative à **Pygame**, bien connue désormais. Elle est davantage ciblée sur la gestion graphique des jeux, y inclut notamment la 3D et l'appel aux langages dédiés aux cartes graphiques : les performances s'en ressentent et sont globalement bien meilleures 😊.

Elle reste à ce jour très largement en version alpha/beta avec tous les bugs que cela peut comporter mais elle s'améliore de jour en jour. De gros bugs sous Windows ont ainsi été corrigés durant février 2022 ce qui permet une utilisation tout à fait raisonnable.

On utilisera ici l'EDI **Spyder** pour gérer les fichiers, cela sera bien plus efficace qu'avec **Jupyter**.

Lien vers la bibliothèque Arcade ici :

https://api.arcade.academy/en/latest/examples/platform_tutorial/step_01.html

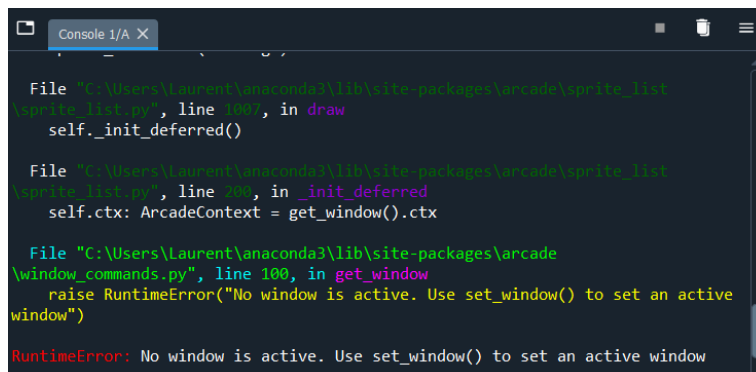
Avantages :

- Elle comporte un module de chargement très simple et optimisé pour des cartes générées avec le logiciel **Tiled Map** sauvegardées sous le *format .json* y compris les **collisions**.
- Elle possède différents types de **moteurs physiques** qui permettent de gérer efficacement les **déplacements** d'un joueur et son interaction avec l'environnement. On utilisera ici le plus simple.
- Elle gère réellement la **transparence** (pas de couleur à assigner comme dans Pygame).
- Elle possède un système de caméra performant qui permet par exemple de la centrer sur le joueur.
- Il y a également une IHM (Interface **H**omme **M**achine, *GUI pour Graphical User Interface en anglais*).
- Il y a un réel gain de vitesse dans l'exécution, cette bibliothèque est particulièrement adaptée pour générer des RPG 😊.

Inconvénients :

- La jeunesse de cette bibliothèque (2020, lien ici <https://www.geeksforgeeks.org/arcade-library-in-python/>) ainsi que son aspect ambitieux induisent la présence de bugs : la bibliothèque Pygame version 2 date de 2009 à titre de comparaison ; le langage C, créé en 1972 verra sa norme définitive (C99) soit 27 ans après.
- La documentation est intégralement en anglais. Cela étant, l'anglais est LA langue de l'informatique donc à comprendre.

Important : il faudra systématiquement **relancer le noyau** à chaque modification du programme sous peine d'obtention de messages d'erreurs très sympathiques comme celui-ci (avec un plantage à la clé bien sûr) :



```
File "C:\Users\Laurent\anaconda3\lib\site-packages\arcade\sprite_list.py", line 1007, in draw
    self._init_deferred()

File "C:\Users\Laurent\anaconda3\lib\site-packages\arcade\sprite_list.py", line 200, in _init_deferred
    self.ctx = ArcadeContext = get_window().ctx

File "C:\Users\Laurent\anaconda3\lib\site-packages\arcade\window_commands.py", line 100, in get_window
    raise RuntimeError("No window is active. Use set_window() to set an active window")

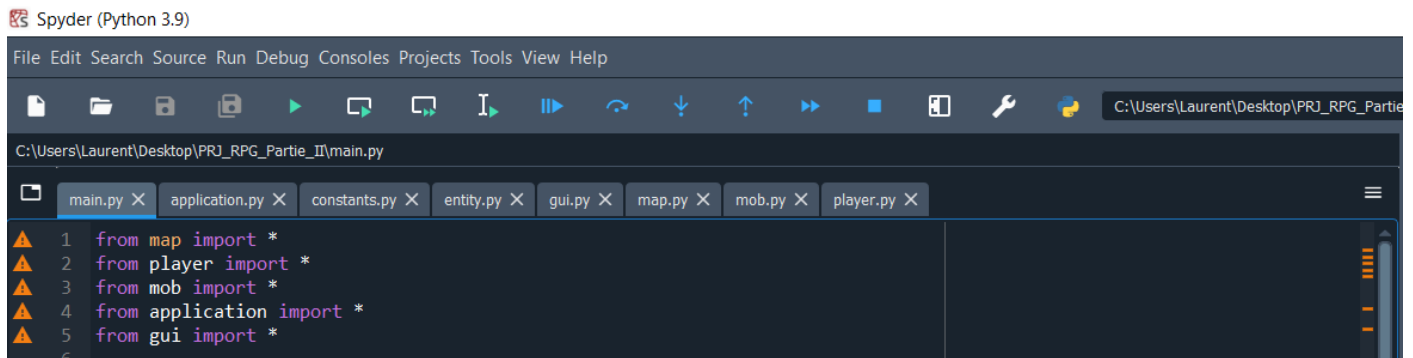
RuntimeError: No window is active. Use set_window() to set an active window
```

II/ Présentation du squelette du programme

1/ Version de base fournie

Lancer l'EDI **Spyder** et charger tous les fichiers en **.py**.

Une fenêtre comme celle-ci doit s'afficher :



A l'exécution, on obtient ceci :



Ce fichier squelette, une fois complété, permettra :

- De **charger** la **map créée** avec la gestion des **collisions** pour le joueur (version de base).
- De **charger** le **joueur** avec ses animations (fournies / créées).
- De **charger** des **monstres** avec leurs animations (fournies / créées).
- De **créer** une **IHM** de base (création de boutons, d'affichage de caractéristiques du joueur / monstre cliqué).
- De **gérer** les **interactions** du joueur : clics sur l'IHM, les monstres, déplacements.

2/ Le fichier « *main.py* »

Comme son nom l'indique, il s'agit du fichier principal du jeu, c'est lui qui va agréger tous les autres.

Il se présente sous forme d'une classe `MyGame`` qui hérite de la classe `Window`` issue de la bibliothèque Arcade.

On peut définir un **héritage** comme une classe bénéficiant de tous les attributs et méthodes de sa classe dite parente : en Python, l'instruction `super()` permet d'y accéder directement.

Ici, la classe `MyGame` **hérite** de la classe `Window` ce qui lui permet donc d'afficher la scène à l'écran. Pour se faire, on déclare la classe ainsi, avec l'instruction `super()` dans le constructeur `__init__()` :

```
# Classe de base du jeu
class My_Game(Window): # Hérite de la classe `Window`
    def __init__(self):
        # Création de la fenêtre
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_TITLE)

        self.map = None          # Carte du jeu

        self.scene = None        # Permet d'afficher la carte (Arcade)
        self.physics_engine = None # Moteur physique d'Arcade
        self.camera = None       # Vue sur la carte (Arcade)

        self.player = None       # Gestion du joueur
        self.mobs = []           # Gestion des monstres (d'où la liste)
        self.sprites_list = None  # Affichage des mobs (Arcade)

        self.gui = None          # Gestion de l'IHM
```

Remarque : il faut bien sûr se renseigner sur la classe parente (paramètres exigés pour sa construction, attributs, méthodes) d'où le rôle de la documentation.

Voici la méthode `setup()` qui permet le **chargement des différents éléments** constituant le RPG :

```
def setup(self):
    # Couleur de fond de la map.
    arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

    # Création de la caméra

    # Création de la map
    self.map = Map()
    self.map.setup()
    self.scene = arcade.Scene.from_tilemap(self.map.tile_map)

    # Liste des mobs à afficher

    # Création du joueur

    # Ajout du joueur dans la list de mobs à afficher
    # ATTENTION : avant l'appel au setup(), plantage sinon !!

    # Création de X orcs placés aléatoirement, X au choix :)

    # ATTENTION : avant l'appel au setup(), plantage sinon !!

    # Création du moteur physique (déplacement et collision du joueur)

    # Création de l'IHM
```

Remarque : Il faudra **compléter** au fur et à mesure cette méthode aux endroits indiqués !

Dans sa version de base, cette méthode ajoute une couleur de fond (bleu ciel ici) et permet de charger le terrain.

Les méthodes qui suivent correspondent à de la **programmation événementielle**, c'est-à-dire des fonctions appelées lorsque l'utilisateur clique sur une touche du clavier ou un bouton de la souris par exemple.

Les méthodes ``on_key_press(self,key,modifiers)`` et respectivement ``on_key_release(self,key,modifiers)`` sont relatives à des événements de type clavier. Lorsque l'on appuie sur une touche, la première méthode est appelée puis la seconde au relâchement de la touche.

La combinaison permet de gérer l'appui en continu sur une (ou plusieurs) touche du clavier par une succession de *pressed / released*.

La méthode ``on_update(self,delta_time)`` permet une mise à jour à intervalle régulier (*delta_time* vaut environ 0.0166 seconde ce qui correspond à un FPS de 60). Cela correspond par exemple à un déplacement, une animation mais aussi l'affichage.

La méthode ``on_draw(self)`` permet -pour simplifier- d'effacer l'écran puis d'afficher le nouveau contenu.

Enfin, voici la fonction d'entrée, appelée au début de l'exécution du programme :

```
# Fonction principale
# NE PAS CHANGER
def main():
    # Instance de My_Game()
    window = My_Game()
    window.setup()

    # Lancement du jeu
    arcade.run()

if __name__ == "__main__":
    main()
```

Elle permet notamment la construction du contexte dans lequel le jeu évoluera et la création de tous les objets nécessaires (carte, joueur, monstres, IHM ...).

Cette partie ne sera pas à modifier.

3/ Le fichier « *constants.py* »

Ce fichier répertorie certaines constantes qui seront utilisées dans le programme. Par convention, elles sont écrites en lettres majuscules.

Voici le contenu de base de ce fichier :

```
import arcade
from random import randint

# L'écran en général
SCREEN_WIDTH = 800 # Largeur de la fenêtre
SCREEN_HEIGHT = 600 # Hauteur de la fenêtre
SCREEN_TITLE = "NSI_RPG" # Titre de l'écran (jeu)

# Map
MAP_FILE = "Maps/map.json" # Nom du fichier de la carte
MAP_SCALING = 0.5 # Mise à l'échelle souhaitée d'un tile
MAP_WIDTH = 1280 # Largeur de la map créée
MAP_HEIGHT = 1280 # Hauteur de la map créée

# Caractéristiques du joueur

# Caractéristiques d'un orc (mob)
```

On y retrouve :

- Pour l'écran (screen) : la **taille** (largeur et hauteur) de fenêtre, son **titre**. Dans la mesure du possible, conserver un ratio 4 : 3 pour l'écran (ici, 800 : 600 = 4 : 3), il pourrait y avoir une image déformée sinon. En cas d'écran 16 : 9, on choisira les mesures adéquates (ce n'est pas le cas au lycée).

- Pour la carte (map) : le **chemin relatif** de la carte, l'**échelle** à appliquer (ici c'est une réduction de 2 en largeur / hauteur) et les **dimensions** de la carte créée (largeur / hauteur).

A noter : ces valeurs sont bien sûr à modifier en fonction des données du terrain généré.

4/ Le fichier « map.py »

Ce fichier est dédié à la **création** et à l'**affichage** du terrain. La bibliothèque *Arcade* dispose nativement d'un module permettant son chargement avec la gestion des collisions, le fichier devant être sauvegardé sous format *.json* (natif avec le logiciel Tiled Map).

Voici un extrait du fichier « map.json » ouvert avec Notepad++:

```
{ "compressionlevel":-1,
  "editorsettings":
  {
    "export":
    {
      "format":"csv",
      "target":"Test_RPG\Test.csv"
    }
  },
  "height":40,
  "infinite":false,
  "layers":[
    {
      "id":4,
      "layers":[
        {
          "data":[1345, 1346, 1346, 1346, 1346, 1346,
          "height":40,
          "id":1,
          "name":"ground",
          "opacity":1,
          "type":"tilelayer",
          "visible":true,
          "width":40,
          "x":0,
          "y":0
        },
        ],
      },
    ],
  }
```

Un fichier de type *.json* (*JavaScript **Objet** **Notation** en anglais*) est un fichier de données textuelles se présentant sous la forme d'un dictionnaire. On peut lire ici que la carte a une hauteur de 40 par exemple.

De nombreux langages disposent d'outils pour lire ce type de fichier : ainsi, le langage Python dispose d'une bibliothèque *json* qui permet de charger très facilement ce type de fichier. Ce sera utile pour charger les caractéristiques d'un monstre par exemple.

La classe `'Map'` est très simple comme le montre le code ci-dessous :

```
from constants import *

class Map :
    def __init__(self) :
        self.tile_map = None

    def setup(self) :
        # Traitement des calques à collisions.
        layers_options = {
            "ground_collide1" : {"use_spatial_hash": True },
            "nature_collide2" : {"use_spatial_hash": True },
            "house_collide2" : {"use_spatial_hash": True }
        }

        # Charge la map en format .json
        self.tile_map = arcade.load_tilemap(MAP_FILE, MAP_SCALING, layers_options)
```

La carte est tout simplement chargée dans l'attribut `tile_map`. A noter que les **collisions sont prises en compte** dans la variable `layers_options` : il s'agit tout simplement du **nom des calques consacrés aux objets à collision** dans le logiciel Tiled Map Editor.

Le contenu de cette variable est à adapter en fonction de la carte créée.

Rappel : Pour intégrer cette carte dans le jeu, trois lignes suffisent dans le programme principal :

```
# Création de la map
self.map = Map()
self.map.setup()
self.scene = arcade.Scene.from_tilemap(self.map.tile_map)
```

5/ Le fichier « *entity.py* »

Ce fichier est le squelette de la création de toutes les entités mobiles (*MOB*, **Mobile Object** en anglais pour les MMORPG).

La classe `Entity` **hérite** de la classe `Sprite` de la bibliothèque *Arcade* et bénéficie donc de tous ses attributs / méthodes. C'est un avantage majeur pour la gestion de l'affichage, déplacement, collisions etc. de ses instances.

Voici le détail du constructeur `__init__()` de la classe `Entity` :

```
from constants import *
import json

class Entity(arcade.Sprite):
    def __init__(self, file_name, scaling, img_width, img_height, coords):
        # Attention, bien indiquer la taille de l'image qui sera affichée et non
        # la taille totale !
        super().__init__(file_name, scaling, 0, 0, img_width, img_height)

        self.file_name = file_name # Nom du fichier des animations
        self.scaling = scaling # Réduction / agrandissement
        self.img_width = img_width # Largeur de la PARTIE de l'image à afficher
        self.img_height = img_height # Hauteur de la PARTIE de l'image à afficher

        self.coords = coords # Coordonnées des parties de l'image (texture) à afficher
        self.textures = [] # Ensemble des textures à charger en fonction des postures
        self.current_texture_indice = 0 # Numéro de la texture à afficher

        self.center_x = 0 # Abscisse ABSOLUE du centre de l'entité
        self.center_y = 0 # Ordonnée ABSOLUE du centre de l'entité
        self.attributes = {} # Caractéristiques de l'entité (Points de vie etc .)

    def setup(self):
        # Changement des animations du mob
        pass

    def update(self):
        pass
```

On remarque bien l'héritage de la classe `Sprite` dans la définition de la classe `Entity`. L'appel à son constructeur se fait grâce à l'instruction `super().__init__()`.

A noter les paramètres demandés par le constructeur :

- Le **chemin relatif du fichier** contenant les textures.
- L'**échelle** à adopter.
- Les deux zéros suivants indiquant que l'on commence en haut à gauche de l'image (pas de marge).
- Les variables `img_width` et `img_height` correspondent aux **dimensions de la partie de l'image à afficher** et non de l'image entière. En principe, il s'agit de 16, 32 ou 64 pixels.

Viennent ensuite trois attributs qui correspondent au chargement de l'image contenant les animations de l'entité.

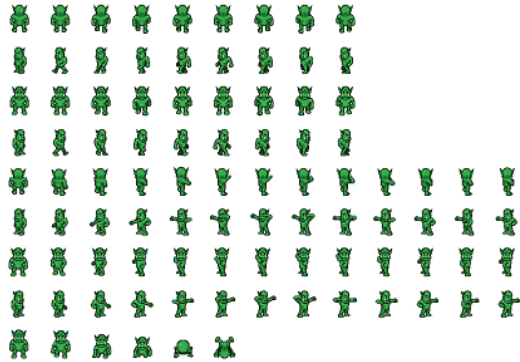
- L'attribut `coords` contient l'intégralité des **coordonnées de chaque texture** à afficher et ce en fonction de son **état** : marche vers le bas, attaque vers le haut par exemple. Il s'agit d'une liste de listes de tuples.
- L'attribut `textures` contient **toutes les parties d'images** à afficher : c'est une liste de listes de textures.
- L'attribut `current_texture_indice` permet de savoir quelle texture afficher : elle augmente de 1 à chaque appel à `update()` et permet **l'animation** de l'entité.

Remarque : on peut définir une **texture** par une image transformée pour être directement appliquée dans le contexte créé, elle peut être en 2D, 3D voire 4D : en effet, le programme ne peut pas utiliser un format *.png* par

exemple. Il existe des textures dites *procédurales* c'est-à-dire créés par un algorithme (celui de Perlin est un bon exemple).

Ce processus est automatique et effectué lors du chargement de l'image.

Dans le jeu RPG, voici l'image correspondant aux futures animations d'un orc et les constantes correspondantes :



```
# Caractéristiques d'un orc
ORC_WIDTH, ORC_HEIGHT = 32, 32
ORC_SCALING = 2
ORC_FILE = "Mobs/orc.png"

WALK_DOWN, WALK_LEFT, WALK_RIGHT, WALK_UP = 0, 1, 2, 3

ORC_WU_COORDS = [(0,0), (32,0), (64,0), (96,0), (128,0), (160,0), (192,0), (224,0), (256,0)]

ORC_SPRITE_COORDS = [ORC_WD_COORDS, ORC_WL_COORDS, ORC_WR_COORDS, ORC_WU_COORDS, \
    ORC_AD_COORDS, ORC_AL_COORDS, ORC_AR_COORDS, ORC_AU_COORDS ]
```

En conséquence, l'attribut *textures* doit charger chaque partie de l'image qui correspondent à chaque état de l'orc (WALK_UP par exemple).

Ainsi, pour accéder à la liste des coordonnées des textures correspondant à l'état WALK_UP, il suffira d'écrire l'instruction *textures[WALK_UP]*. A l'aide de l'attribut *current_texture_indice*, on peut ainsi accéder à la texture souhaitée. Par exemple, l'instruction *textures[WALK_UP][2]* donnera la partie de l'image située aux coordonnées (96,0). Il ne reste plus qu'à faire varier cet indice pour obtenir l'animation souhaitée.

Ce chargement d'images doit s'effectuer dans la méthode `setup(self)`.

Pour gérer la position des mobs ainsi que leurs caractéristiques, trois autres attributs sont disponibles :

- Les attributs *center_x* et *center_y* proviennent de la classe `Sprite`. Ils sont utilisés par le moteur physique pour le joueur et donnent leur **position absolue** sur la carte du centre de l'image.
- Le dictionnaire *attributs* qui donnent des indications sur l'entité comme par exemple sa vitesse de déplacement, sa position initiale, ses points de vie etc.

6/ Le fichier « *player.py* »

La classe `Player` représente le joueur comme son nom l'indique. Elle hérite de la classe `Entity` présentée ci-dessus.

Voici son constructeur :

```
class Player(Entity):
    def __init__(self, file_name, scaling, img_width, img_height, coords):
        # Attention, bien indiquer la taille de l'image qui sera affichée !
        super().__init__(file_name, scaling, img_width, img_height, coords)

        self.status = 0 # Etat du joueur (animation) : walk_left par exemple

        # Position de départ
        self.init_x_pos = 0 # En abscisse
        self.init_y_pos = 0 # En ordonnée
```

On retrouve l'appel à la classe parente `Entity`. Quelques nouveaux attributs sont à noter :

- L'attribut *status* qui indique l'**état** du joueur pour l'animation. Si le joueur marche vers la gauche, l'animation correspondante pourra être appelée.
- Les attributs *init_x_pos* et *init_y_pos* représentent la **position** (abscisse -respectivement- ordonnée) de départ du joueur. Cela est utile pour le début du jeu mais aussi si le joueur meurt par exemple. Cette position est **absolue** et ne dépend donc pas de l'affichage du terrain !

Voici la méthode `setup(self)` de cette classe et le fichier `.json` correspondant :

```
def setup(self) :
    # Chargement des textures
    super().setup()

    # Ouverture du fichier JSON file
    # Chargement des caractéristiques du joueur
    f = open("Mobs/player.json")
    data = json.load(f)

    for key,value in data["Player"].items():
        self.attributes[key] = value

    # Fermeture du fichier
    f.close()

    # Position / Etat / Image de départ
    # ATTENTION à l'échelle de la carte.

    # Texture de départ : à la base, joueur se déplaçant vers le bas
    self.texture = self.textures[0][0]
```

Caractéristiques du joueur

```
{ "Player" :
  {
    "Attack" : 30,
    "Block" : 0.10,
    "Defense" : 20,
    "Dodge" : 0.05,
    "HitPoints" : 40,
    "Init_x" : 1200,
    "Init_y" : 110,
    "Name" : "Kang",
    "Parry" : 0.15,
    "Speed" : 3
  }
}
```

L'instruction `super().setup()` permet d'appeler la méthode `setup(self)` de la classe parente ici `Entity`.

A noter : La POO permet que les classes qui héritent d'une autre redéfinissent (ou surchargent) les méthodes de leur classe père. A l'exécution du programme, c'est bien sûr leur méthode propre qui sera appliquée et si aucune méthode n'est redéfinie, c'est celle du parent qui s'exécutera.

Ce principe s'appelle le **polymorphisme**.

Un fichier `.json` se présente comme un dictionnaire de dictionnaires. Ce format, créé initialement pour le langage JavaScript, s'est affranchi des langages en général et la plupart d'entre eux proposent une bibliothèque pour les lire simplement.

En langage Python, il suffit d'importer la bibliothèque `json`, le code qui s'ensuit est relativement simple à comprendre.

Voici la méthode `update(self)` qui est consacrée à l'animation du personnage (pour l'instant 😊). Elle est appelée automatiquement environ 60 fois par seconde :

```
def update(self) :
    # Le joueur doit rester sur la map
    # Les abscisses, ne pas dépasser la largeur de la map
    # ATTENTION à l'échelle de la carte.

    # Puis les ordonnées, ne pas dépasser la hauteur de la map
    # ATTENTION à l'échelle de la carte.

    # Animation
    # Si le joueur ne bouge pas, pas d'animation
    if not self.change_x and not self.change_y :
        return

    # Si changement de type de déplacement

    # Image suivante sauf si l'on est à la fin de l'animation (retour au début)

    # Nouvelle texture à charger
    self.texture = self.textures[self.status][self.current_texture_indice]
```


Trois attributs sont présents ici, ils proviennent de la classe `Sprite` et sont à utiliser :

- Les attributs `change_x` et `change_y` reflètent le **déplacement** du joueur (appui sur une touche de clavier par exemple). Ils seront remis à zéro lors de la fin de l'événement.
- L'attribut `texture` représente ce qui est à **afficher** en fonction de l'**état** du joueur et de l'**indice** de la texture. Par défaut, tout est à zéro et il n'y a pas d'animation.

L'instruction suivante est particulièrement intéressante :

```
# Animation
# Si le joueur ne bouge pas, pas d'animation
if not self.change_x and not self.change_y :
    return
```

Si aucun mouvement n'est détecté, aucune animation ne s'effectue : cela permet de s'assurer qu'elle est liée aux événements et de gérer également l'appui continu sur une touche de déplacement du clavier. En effet, il y a une alternance d'appels aux méthodes `on_key_press(self, key, modifiers)` et `on_key_release(self, key, modifiers)`, la dernière remettant à zéro les attributs `change_x` et `change_y`.

7/ Le fichier « `mob.py` »

La classe `Mob` gère toutes les entités de type monstre, elle hérite d'ailleurs également de la classe `Entity`. Il y a peu de différences avec la classe `Player`, il faudra simplement implémenter les positions initiales (aléatoires ou pas) ainsi qu'un déplacement automatique réaliste dans un premier temps.

Voici le constructeur de la classe `Mob` :

```
class Mob(Entity) :
    def __init__(self, file_name, scaling, img_width, img_height, coords) :
        # Attention, bien indiquer la taille de l'image qui sera affichée !
        super().__init__(file_name, scaling, img_width, img_height, coords)

        self.status = 0

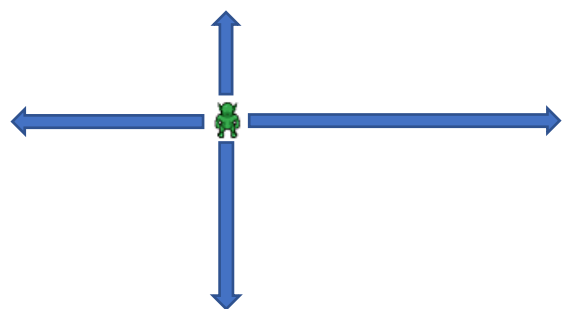
        # Position / Image de départ
        self.init_x_pos = 0
        self.init_y_pos = 0

        # Compteur pour le nombre maximal de déplacements (aléatoire)
        self.init_count_tick_move = 0
        # Compteur courant
        self.current_count_tick_move = 0
```

Génération aléatoire de déplacement en forme de croix

Pas de réelles nouveautés hormis les deux derniers attributs `init_count_tick_move` et `current_count_tick_move`. Leur intérêt est dans la génération d'un mouvement aléatoire, en forme de **croix** comme présenté à droite.

La longueur de chaque flèche est générée aléatoirement à la fin de chaque déplacement.



Une méthode permet d'initialiser la position de départ de chaque monstre (aléatoirement éventuellement) :

```
# Position initiale éventuellement aléatoire
def set_init_position(self, x_pos, y_pos):
    pass
    # Placement du mob
    # Attention : Tenir compte de MAP_SCALING

    # Initialisation des attributs `init_x_pos` et `init_y_pos`
```

Elle sera à appeler lors de la création du jeu.

```
def setup(self):
    # Couleur de fond de la map.
    arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

    # Création de la caméra

    # Création de la map
    self.map = Map()
    self.map.setup()
    self.scene = arcade.Scene.from_tilemap(self.map.tile_map)

    # Liste des mobs à afficher

    # Création du joueur

    # Ajout du joueur dans la list de mobs à afficher
    # ATTENTION : avant l'appel au setup(), plantage sinon !!

    # Création de X orcs placés aléatoirement, X au choix :)

    # ATTENTION : avant l'appel au setup(), plantage sinon !!

    # Création du moteur physique (déplacement et collision du joueur)

    # Création de l'IHM
```

A ajouter ici 😊

La méthode `update(self)` permet d'activer l'animation du monstre et est appelée à chaque mise à jour du jeu soit 60 fois par seconde.

```
def update(self) :
    # Mise à jour de l'animation

    # Si changement de type de déplacement

    # Mise à zéro du déplacement
    self.change_x, self.change_y = 0,0

    # Nouvelle texture à afficher
    self.texture = self.textures[self.status][self.current_texture_index]
```