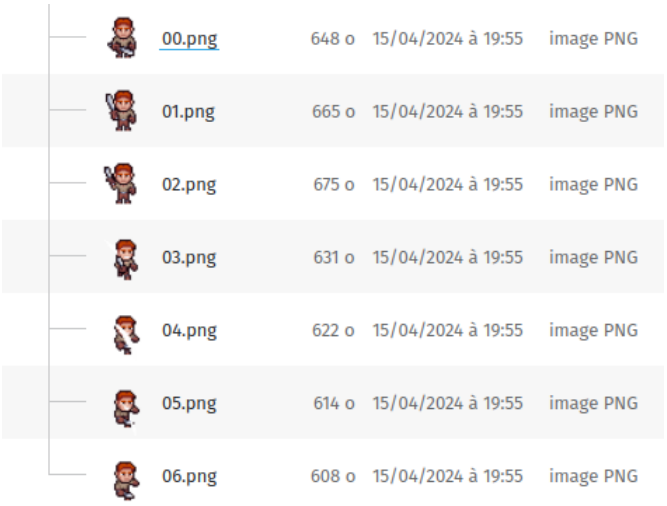
Projet Jeu Vidéo (Solo downgrading)

Je part tout d'abord sur la librairie pygame, avec un système d'animation de personnage basique, mais cela va se compliquer rapidement, effectivement, pour l'adapter à la façon dont je gère les sprites, j'écris un programme en bash afin de séparer les images censés former des animations en images seules:

Exemple:
Ainsi, on passe d'un seul fichier:

À plusieurs fichiers comme cela:



Avec ce script (je l'ai fait rapidement donc il y a des petits bug, mais ça a fait le travail donc je suis assez content):

```
#!/bin/bash
SPRITESHEET_PATH="Character_Walk.png"
OUTPUT_BASE=""
WIDTH=32
HEIGHT=32
COLUMNS=4
ROWS=4
INDEX=0

for ((row=0; row<$ROWS; row++)); do
  for ((col=0; col<$COLUMNS; col++)); do
    X=$(($col*$WIDTH))
    Y=$(($row*$HEIGHT))
    OUTPUT_FILE="${OUTPUT_BASE}$(printf "%02d" $INDEX).png"
    convert "$SPRITESHEET_PATH" -crop ${WIDTH}x${HEIGHT}+${X}+${Y} "$OUTPUT_FILE"
    INDEX=$((INDEX+1))
  done
done

echo "Images séparées avec succès."
```

Explication du code du joueur (classe Player):

Gestion des imports des dépendances:

```
#Import des librairies externes au programmes
import pygame as pg
import sys, os

#Import des classes internes
import settings #Fichier de paramétrage ou sont stocké les valeurs et constantes communes
```

```
sys.path.append(settings.BASE_DIR) # Utilisation du module sys pour éviter les problèmes de compatibilité dû aux changements de système et donc de chemin de dossiers (chemin relatifs)
from game.support import *  #Import de la classe support qui contient des fonctions essentielles au bon fonctionnement du programme.
```

Note, j'ai choisis d'utiliser des classes pour faciliter la répartition du projet dans plusieurs fichiers (et aussi pour la RAM, étant donné que le projet aurait du se faire sur Raspberry Pi 4, et qu'on ne dispose *que* de 4GO pour faire tourner le systeme Raspberry Pi OS basé sous Debian.

Création du constructeur (fonction `__init__`)

```
class Player(pg.sprite.Sprite):
    def __init__(self, pos, group):
        super().__init__(group) # Appel du constrcteur de la classe parente (pg.sprite.Sprite)
        self.import_assets() # Import des assets (image du joueur)
        self.status = 'down_idle'  # Création du status du joueur par défaut
        self.frame_index = 0
        self.all_sprites = group # Encapsulation du paramètre group dans le constructeur

        self.image = self.animations[self.status][self.frame_index] # Création de l'image par défaut
        self.rect = self.image.get_rect(center = pos)

        self.direction = pg.math.Vector2()
        self.pos = pg.math.Vector2(self.rect.center) # Position du joueur dans l'espace 2D
        self.speed = 300 # Vitesse du protagoniste
        self.attack_pulse = False
        self.attack_status = False
        self.z = settings.LAYER['main'] # Définit la profondeur (potentiellement utile pour le placement des éléments)
        self.player_rect = pg.Surface.get_rect(self.image)
        self.feet = pg.Rect(0, 0, self.player_rect.width * 0.5,
                                                self.player_rect.width * 0.25) # Création du rectange de colisions
        self.old_position = self.pos.copy() # Sert pour les colisions
```

Organisation de la fonction input

- La fonction input gère les entrés claviers et souris du joueur:

```
    def input(self):
        input_keys = pg.key.get_pressed()
        if (input_keys[pg.K_z] and input_keys[pg.K_s]):
                self.direction.y = 0

        elif (input_keys[pg.K_q] and input_keys[pg.K_d]):
                self.direction.x = 0
        else:
            if input_keys[pg.K_z]:
                self.direction.y = -1
                self.status = 'up_run'
            elif input_keys[pg.K_s]:
                self.direction.y = 1
                self.status = 'down_run'
            else:
                self.direction.y = 0

            if input_keys[pg.K_q]:
                self.direction.x = -1
                self.status = 'left_run'
            elif input_keys[pg.K_d]:
                self.direction.x = 1
                self.status = 'right_run'
            else:
                self.direction.x = 0

        if pg.mouse.get_pressed()[0]:
            self.attack_pulse = True
```

```
        else:
            self.attack_pulse = False
```

- **Déplacement**:
  - Si les touches `Z` et `S` sont pressées ensemble, la variable gérant la direction de joueur sur l'axe Y est réinitialisé, ce qui signifie un arrêt vertical.
  - Si les touches `Q` et `D` sont pressées ensemble, même système, mais sur l'axe X
  - Sinon, les touches `Z` et `S` contrôlent le mouvement vertical (`up_run` ou `down_run`), et les touches `Q` et `D` contrôlent le mouvement horizontal (`left_run` ou `right_run`).
- **Attaque**:
  - Si le bouton gauche de la souris est pressé (`pg.mouse.get_pressed()[0]`), `self.attack_pulse` est mis à `True`, indiquant une attaque (non implémenté à ce stade).
  - Si le bouton gauche de la souris n'est pas pressé, `self.attack_pulse` est mis à `False`.

Organisations des fonctions annexes:

```
    def get_status(self):
        if self.direction.magnitude() == 0: # norme du vecteur
            self.status = self.status.split('_')[0] + '_idle'
        if self.attack_pulse == 1:
            self.status = self.status.split('_')[0] + '_attack'


    def saveloc(self):
        self.old_position = self.pos.copy()


    def move(self, dt):
        if self.direction.magnitude() > 0:
            self.direction = self.direction.normalize()

        self.pos.x += self.direction.x * self.speed * dt
        self.rect.centerx = self.pos.x - 10

        self.pos.y += self.direction.y * self.speed * dt
        self.rect.centery = self.pos.y - 5


    def move_back(self):
        self.pos = self.old_position
        self.player_rect.topleft = self.pos
        self.feet.midbottom = self.player_rect.midbottom()

    def import_assets(self):
        self.animations = {'up_run' : [], 'down_run' : [], 'right_run' : [], 'left_run' : [], 'up_idle' : [],
'down_idle' : [], 'right_idle' : [], 'left_idle' : [], 'up_attack' : [], 'down_attack' : [], 'right_attack' :
[], 'left_attack' : [], 'up_death' : [], 'down_death' : [], 'right_death' : [], 'left_death' : []}

        for animation in self.animations.keys():
            path = os.path.join(settings.IMAGES_DIR, "RPG_Hero/") + animation
            self.animations[animation] = import_folder(path)
```

`get_status(self)`:

- **Description**: Met à jour l'image du personnage en fonction de sa direction et de son état d'attaque.

`saveloc(self)`:

- **Description**: Sauvegarde la position actuelle du personnage pour un éventuel retours en arrière lors du système de collision.

`move(self, dt)`:

- **Description**: Déplace le personnage en fonction de sa direction et de sa vitesse, en tenant compte du temps écoulé (`dt`).
- **Paramètres**: `dt` (float) - Temps écoulé depuis la dernière itération de la boucle.

`move_back(self)`:

- **Description**: Restaure la position précédente du personnage.

`import_assets(self)`

- **Description**: Importe les animations pour chaque état du personnage à partir d'un répertoire spécifié.

Résumé: Ces fonctions sont conçues pour gérer l'état du personnage, sa position, son mouvement, et l'importation de ses animations. Elles permettent de mettre à jour l'état du personnage en fonction de ses actions, de sauvegarder sa position pour des mouvements réversibles, de le déplacer en fonction de sa direction et de sa vitesse, et d'importer les animations nécessaires pour chaque état du personnage (déplacement, immobilisation, etc...).

Gestion des animations:

```
    def animate(self, dt):
        if "idle" in self.status:
            self.frame_index += 3 * dt
        elif "attack" in self.status:
            self.frame_index += 10 * dt
            self.attack_pulse = False
        else:
            self.frame_index += 10 * dt
        if self.frame_index >= len(self.animations[self.status]):
            self.frame_index = 0

        self.image = self.animations[self.status][int(self.frame_index)]
        img_width, img_height = self.image.get_size()
        coef = 1.3
        new_img_width = coef * img_width
        new_img_height = coef * img_height
                self.image = pg.transform.scale(self.image,
                                                                                (new_img_width,
new_img_height))
```

- **Description**: Met à jour l'animation du personnage en fonction de son état
  Exemple: (`idle` , `attack` , etc.) et du temps écoulé (`dt`).
- **Paramètres**: `dt` (float) - Temps écoulé.

Au niveau de la logique:

- **Vitesse d'animation**: Vitesse variable en fonction de l'état (`idle` ou `attack`).
- **Réinitialisation de l'index**: Si l'index dépasse la longueur de l'animation, il est réinitialisé.
- **Redimensionnement de l'image**: L'image est redimensionnée à 1.3 fois sa taille originale.

Fonction `update` :

- **Entrées**: Appelle `self.input()` pour gérer les entrées utilisateur.
- **État**: Appelle `self.get_status()` pour mettre à jour l'état du personnage.
- **Mouvement**: Appelle `self.move(dt)` pour déplacer le personnage.
- **Animation**: Appelle `self.animate(dt)` pour animer le personnage.
- **Positionnement**: Arrondit les coordonnées de position et met à jour les rectangles du personnage et de ses pieds. (<- Très utile pour les collisions.)

Explication de la classe `Level` :

- Cette classe gère un niveau dans un jeu, incluant la configuration du niveau, la mise à jour des sprites, la détection des collisions, et l'affichage des dialogues.
- **Méthodes principales**:
  - `__init__` : Initialise le niveau.
  - `setup` : Prépare le niveau en chargeant les données du fichier TMX (fichier provenant du logiciel TILED, contenant ainsi les boites de collisions et les images de la map).
  - `run` : Met à jour le niveau à chaque frame, en gérant les entrées, les mises à jour des sprites, la détection des collisions, et l'affichage des dialogues.
  - `collision_detect` : Détecte les collisions entre le joueur et les objets du niveau.
  - `draw_tiles_layer` : Dessine une couche de tuiles à partir des données TMX.
  - `event` : Gère les événements spécifiques aux objets du niveau.
  - `teleport_player` : Déplace le joueur à un point spécifique du niveau.
  - `get_object` : Récupère un objet spécifique du niveau par son nom (abrège la fonction fournie par le module `pytmx` .

Explication de la classe `Camera_grp` :

```
class Camera_grp(pg.sprite.Group):
    def __init__(self):
        super().__init__()
        self.display_surface = pg.display.get_surface()
        self.offset = pg.math.Vector2()
        self.camera = 0

    def customize_draw(self, player):
        self.offset.x = player.rect.centerx - settings.WINDOWS_WIDTH  /2
        self.offset.y = player.rect.centery - settings.WINDOWS_HEIGHT /2

        for layer in settings.LAYER.values():
            for sprite in sorted(self.sprites(), key=lambda sprite : sprite.rect.centery):
                if sprite.z == layer:
                    offset_rect  = sprite.rect.copy()
                    offset_rect.center -= self.offset
                    self.display_surface.blit(sprite.image, offset_rect)
```

Gère tout les sprites avec une caméra, permettant de suivre le joueur et de centrer l'affichage sur lui.

Classe `DialogBox` :

Cette classe gère l'affichage des boîtes de dialogues, permettant de rendre le texte visible lettre par lettre (animation).

Fonctions communes contenues dans le `support.py`

```
from os import walk
import pygame as pg
import settings
import colorama

def import_image(img_path):
        sprite = pg.image.load(img_path).convert_alpha()
        img_width, img_height = sprite.get_size()
        new_img_width = settings.COEF_SCAL * img_width
        new_img_height = settings.COEF_SCAL * img_height
        sprite = pg.transform.scale(sprite, (new_img_width, new_img_height))
        return sprite

def upscale_img(sprite):
    img_width, img_height = sprite.get_size()
    img_width = img_width * settings.COEF_SCAL
    img_height = img_height * settings.COEF_SCAL

    new_img_width = img_width
    new_img_height = img_height
    sprite = pg.transform.scale(sprite, (new_img_width, new_img_height))
    return sprite


def import_folder(path):
    surface_list = []

    for folder_name, sub_folder, img_file in walk(path):
        for image in img_file:
            full_path = path + '/'+ image
            image_surface = import_image(full_path)
            surface_list.append(image_surface)
    return surface_list

def printred(text: str):
    print(colorama.Fore.RED, text, colorama.Fore.RESET)

def printgreen(text: str):
    print(colorama.Fore.GREEN, text, colorama.Fore.RESET)
```

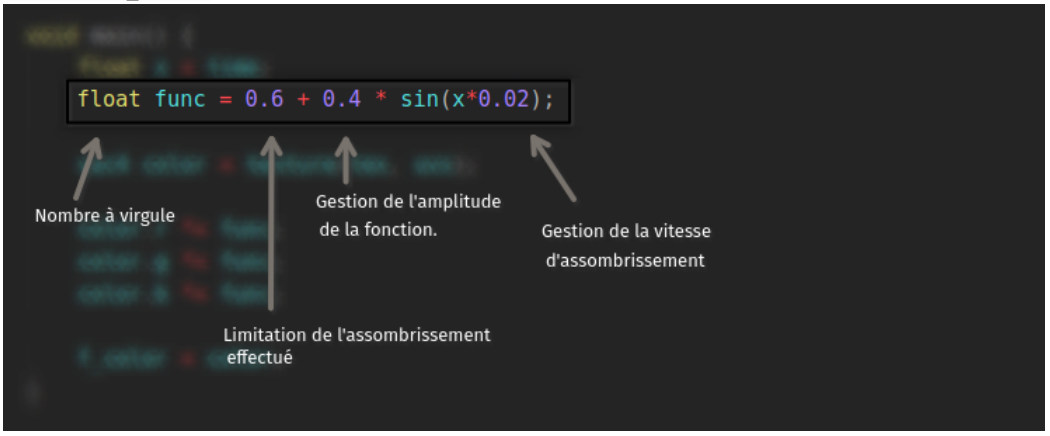
- `import_image(img_path)` : Charge et redimensionne une image.
- `upscale_img(sprite)` : Augmente la taille d'une image.

- `import_folder(path)` : Charge toutes les images d'un dossier.
- `printred(text)` : Imprime du texte en rouge.
- `printgreen(text)` : Imprime du texte en vert.

Explication du Shader en GLSL (OpenGL Shading Language)

```
#version 330 core
uniform sampler2D tex;
uniform float time;
uniform vec2 resolution;

in vec2 uvs;
out vec4 f_color;

void main() {
        float x = time;
    float func = 0.6 + 0.4 * sin(x*0.02);

    vec4 color = texture(tex, uvs);

    color.r *= func;
    color.g *= func;
    color.b *= func;

        f_color = color;
}
```

- `#version 330 core` : Cette ligne spécifie la version du GLSL qui est utilisé.
- Les variables marquées d'un `uniform` correspondent aux information provenant du script python.
- `in vec2 uvs;` : Cette ligne déclare une variable représentant les coordonnées de texture des pixels.
- `out vec4 f_color;` : Représente une variable de sortie, (vecteur à 4 dimensions) correspondant au fragment Shader de sortie.



État & Informations sur le projet:

- ~ 850 Lignes de code (commentaires inclus) :
- Temps de Recherche et Développement: 18 heures, 51 minutes, et 53 secondes
  - Logiciel utilisé pour calculer le temps: ActivityWatch.
  - Le développement s'est fait pratiquement totalement sous Linux (distribution basée sous Arch)
  - Logiciels utilisé pour le développements:
    - Tiled (Outil de création de map)
    - Visual Studio Code (Développement)
    - Konsole (Lancement du script principal)
    - Obsidian (Création de ce PDF *(non comptabilisé dans le temps)*)
  - Modules python utilisés:
    - Colorama Version 0.4.6 (Couleurs dans le terminal/console/cmd)
    - Pygame Version 2.2.5 (Outil de création de jeu)
    - ModernGL Version 5.10.0 (Implémentation des Shaders OpenGL dans pygame)
    - PyTMX Version 3.32 (Outil de gestion de map)

- J'ai voulu à la base faire le Projet sur Raspberry pi 4, mais due...
  - ... au ***faibles*** performances offerte par la carte,
  - ... au fait que le python soit un **langage interprété** (lent),
  - ... à ma volonté de me familiariser le GLSL (OpenGL Shading Language)
    -> Shader = programme qui tourne sur la carte graphique (et il n'y a pas de carte graphique sur les Raspberry (je n'inclue pas le processeur graphique)).
    J'ai du me tourner vers des machines plus puissantes comme un PC puis j'ai entrepris de le développer afin qu'il puisse tourner sur plusieurs système d'exploitation (Linux *natif* + Windows).