

Chargement du joueur du jeu avec Arcade

I/ Préparation du joueur

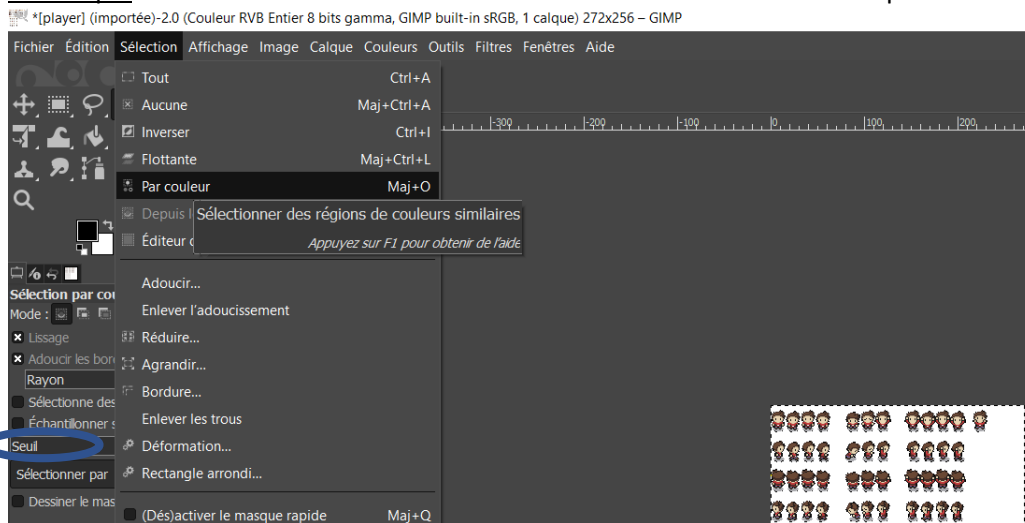
1/ Avec le logiciel GIMP

Tout comme la map, il faut rendre la couleur affichée non transparente.

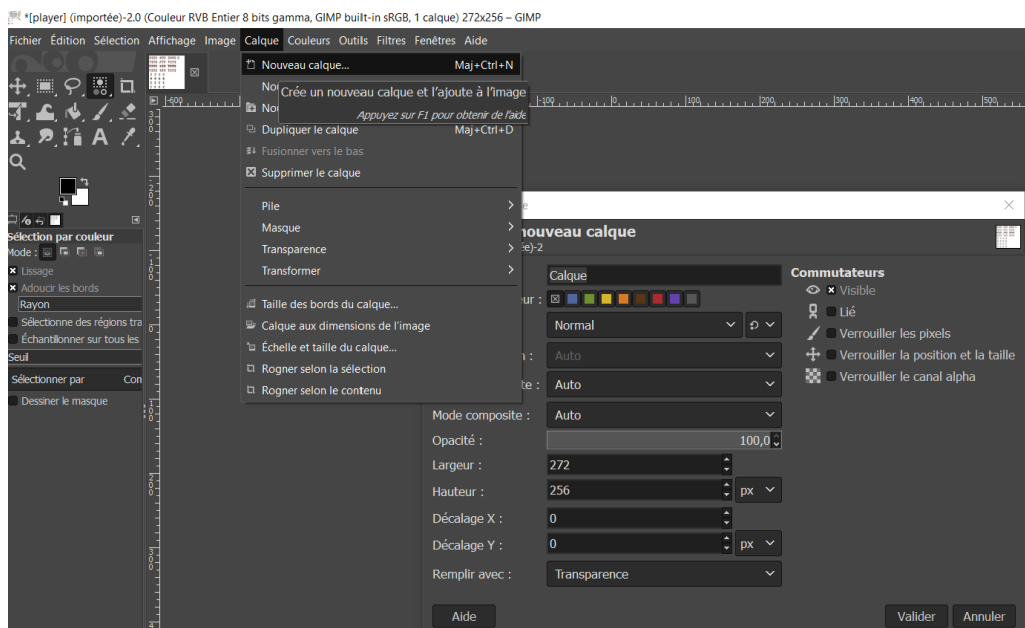
1/ **Charger** le fichier *player.png* avec *GIMP*.

2/ **Opérer** une sélection par couleur (**cliquer sur la couleur à rendre transparente**) et mettre le **seuil** au minimum (encadré en bleu sur l'image ci-dessous).

Remarque : les zones de la couleur concernée doivent être entourées de 'pointillés'.

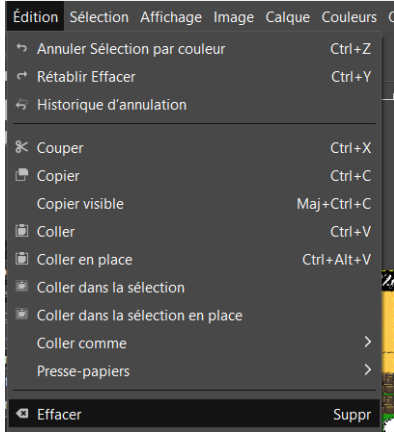


3/ Dans le menu *Calque*, **créer un nouveau calque** puis **valider** (voir ci-dessous) :

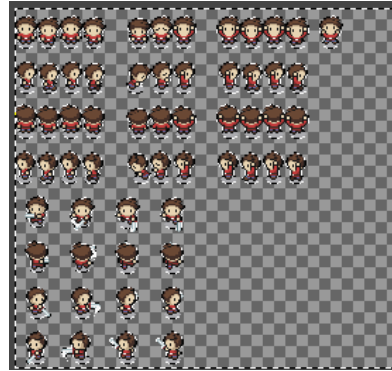


4/ Dans le menu *Edition*, cliquer sur *Effacer* : la couleur sélectionnée devient transparente.

Remarque : la transparence est représentée par des carrés grisés.

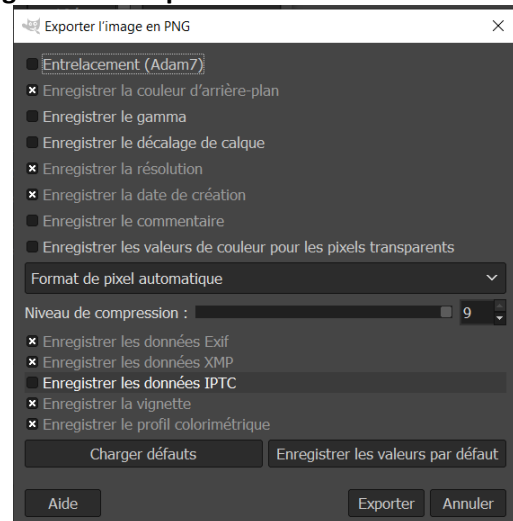
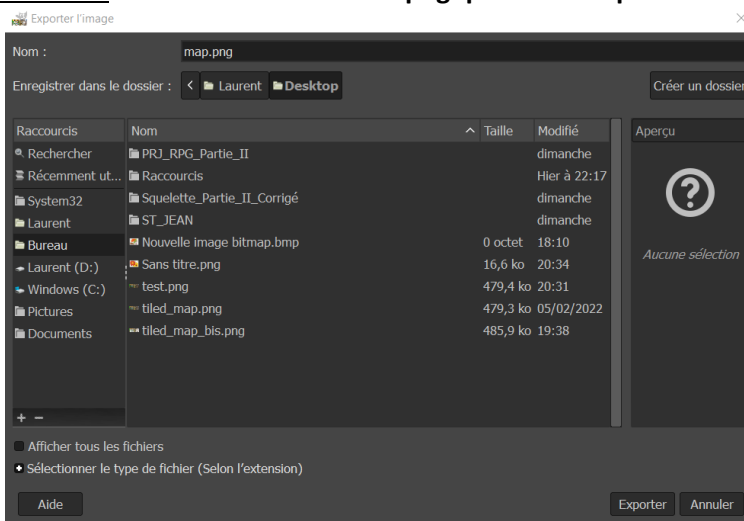


La couleur sélectionnée devient transparente 😊.



5/ Dans le menu *Fichier*, cliquer sur *Exporter sous* (voir ci-dessous). **Changer** le nom du fichier si besoin puis cliquer deux fois sur le bouton *Exporter*.

Attention : conserver l'extension **.png** qui assure la prise en charge de la transparence !



Le joueur est désormais sauvegardé avec la transparence nécessaire pour la bibliothèque **Arcade**.

II/ Chargement du joueur dans le jeu RPG

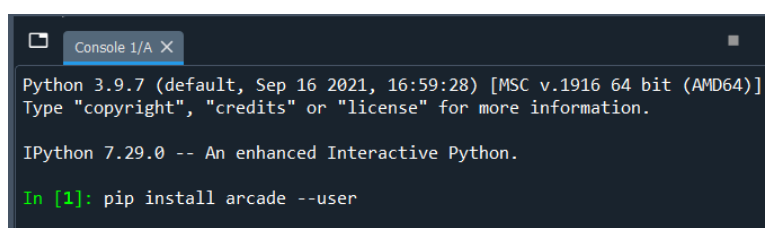
1/ Le fichier `constants.py`

Préalable nécessaire : s'assurer que les fichiers nécessaires sont dans le répertoire /Mobs.

1/ Ouvrir l'EDI **Spyder** et charger les fichiers « squelette ».

Important : dans la partie `console`, importer le bibliothèque **Arcade** avec l'instruction suivante :

pip install arcade --user. Appuyer sur la touche `Entrée` pour exécuter l'instruction (cela peut prendre quelques minutes).



2/ Dans le fichier *constants.py*, adapter les données au fichier *player.png*.

```
# Caractéristiques du joueur
PLAYER_WIDTH,PLAYER_HEIGHT = 16, 32
PLAYER_SCALING = 2
PLAYER_FILE = "Mobs/player.png"

# Indices des animations du joueur
### NE PAS CHANGER ###
PLAYER_WALK_DOWN, PLAYER_WALK_LEFT, PLAYER_WALK_RIGHT,PLAYER_WALK_UP = 0, 1, 2, 3
PLAYER_ATTACK_DOWN, PLAYER_ATTACK_LEFT, PLAYER_ATTACK_RIGHT, PLAYER_ATTACK_UP = 4, 5, 6, 7

# Coordonnées des images de chaque animation du joueur
### A COMPLETER / MODIFIER si autre image ###
PLAYER_WD_COORDS = [(0,0), (16,0), (32,0), (48,0)] # Marche vers le bas
PLAYER_WR_COORDS = [] # Marche vers la droite
PLAYER_WU_COORDS = [] # Marche vers le haut
PLAYER_WL_COORDS = [] # Marche vers la gauche

PLAYER_AD_COORDS = [] # Attaque vers le bas
PLAYER_AU_COORDS = [] # Attaque vers le haut
PLAYER_AR_COORDS = [] # Attaque vers la droite
PLAYER_AL_COORDS = [] # Attaque vers la gauche

# Regroupement des coordonnées précédentes dans une liste
### NE PAS CHANGER ###
PLAYER_SPRITE_COORDS = [PLAYER_WD_COORDS, PLAYER_WL_COORDS, PLAYER_WR_COORDS, PLAYER_WU_COORDS, \
    PLAYER_AD_COORDS, PLAYER_AL_COORDS, PLAYER_AR_COORDS, PLAYER_AU_COORDS ]
```

Remarque : mettre la constante *MAP_SCALING* à 2. De même pour *PLAYER_SCALING* (sauf si déjà fait).

2/ Le fichier *entity.py*

Ce fichier contient la classe parente de toute animation du jeu.

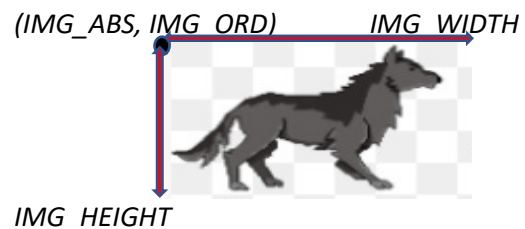
Lien vers des explications :

https://github.com/lmayer65/NSI_T/blob/main/Projets/PRJ_RPG/Partie_III_NPC/Pr%C3%A9sentation_Partie_III.pdf

(Page 6 et 7).

Les paramètres de l'image en schéma

La bibliothèque Arcade dispose de la méthode ***arcade.load_texture(FILE_NAME, IMG_ABS, IMG_ORD, IMG_WIDTH, IMG_HEIGHT)*** crée une **texture** à partir d'une image.



Source : www.freepng.com

Voici le détail des **paramètres** :

- **FILE_NAME** : chemin relatif du fichier des animations du joueur.
- **IMG_ABS** : abscisse du bord supérieur gauche de la partie de l'image à afficher. Par exemple, celle de la deuxième texture du joueur marchant vers le bas est 16 (voir ci-dessus).
- **IMG_ORD** : ordonnée du bord supérieur gauche de la partie de l'image à afficher. Par exemple, celles correspondant au joueur marchant vers le bas est toujours 0 (voir ci-dessus).
- **IMG_WIDTH** : largeur de la **partie** de l'image à afficher (16, 32 ou 64 normalement) et **non de l'image en entier**.
- **IMG_HEIGHT** : hauteur de la **partie** de l'image à afficher (16, 32 ou 64 normalement) et **non de l'image en entier**.

3/ L'attribut **coords** est une liste qui rassemble toutes les parties de l'image représentant l'affichage de l'entité en fonction de ses actions. Par exemple, pour le joueur, **coords** vaut **PLAYER_SPRITE_COORDS** et on a ceci :

```
# Coordonnées des images de chaque animation du joueur
### A COMPLETER / MODIFIER si autre image ###
PLAYER_WD_COORDS = [(0,0), (16,0), (32,0), (48,0)] # Marche vers le bas

# Regroupement des coordonnées précédentes dans une liste
### NE PAS CHANGER ###
PLAYER_SPRITE_COORDS = [PLAYER_WD_COORDS, PLAYER_WL_COORDS, PLAYER_WR_COORDS, PLAYER_WU_COORDS, \
    PLAYER_AD_COORDS, PLAYER_AL_COORDS, PLAYER_AR_COORDS, PLAYER_AU_COORDS ]
```

On remarquera que les **valeurs des états de l'entité** (ici le joueur) ne sont rien d'autre que les **indices** de la liste **PLAYER_SPRITE_COORDS** :

```
# Indices des animations du joueur
### NE PAS CHANGER ###
PLAYER_WALK_DOWN, PLAYER_WALK_LEFT, PLAYER_WALK_RIGHT, PLAYER_WALK_UP = 0, 1, 2, 3
PLAYER_ATTACK_DOWN, PLAYER_ATTACK_LEFT, PLAYER_ATTACK_RIGHT, PLAYER_ATTACK_UP = 4, 5, 6, 7
```

Il s'agit donc plus précisément d'une liste de listes de tuples.

Voici un exemple de programme naïf chargeant les textures concernant le joueur marchant vers le bas. Il se situe dans la méthode **setup(self)** de la classe **Entity** :

```
def setup(self) :
    # Chargement des animations de l'entité en fonction de son statut

    # Ce code permet de charger l'animation du joueur lorsqu'il marche vers le bas.
    lst = []
    lst.append(arcade.load_texture(self.file_name, 0, 0, 16, 32)
    lst.append(arcade.load_texture(self.file_name, 16, 0, 16, 32)
    lst.append(arcade.load_texture(self.file_name, 32, 0, 16, 32)
    lst.append(arcade.load_texture(self.file_name, 48, 0, 16, 32)

    self.textures.append(lst)
```

Remarque : dans un premier temps, ce code suffira pour charger le joueur et sa texture de base

(*) Ecrire la méthode **setup(self)** de la classe **Entity** chargeant les textures nécessaires dans l'attribut **self.textures**. On utilisera **UNIQUEMENT** l'attribut **coords**.

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

Rappel : l'attribut **self.textures** est une liste de listes.

Appel au professeur

3/ Le fichier `player.py`

Ce fichier contient la classe `Player` qui **hérite** de la classe `Entity`. Cette dernière s'occupe notamment du chargement des animations de l'entité (donc de celles du joueur 😊).

Toutes les informations nécessaires se trouvent à ce lien :

https://github.com/Imayer65/NSI_T/blob/main/Projets/PRJ_RPG/Partie_III_NPC/Pr%C3%A9sentation_Partie_III.pdf
(Page 7 à 9).

Voici le fichier *player.json* qui regroupe les **caractéristiques** du joueur. Il s'agit d'un dictionnaire de dictionnaires (une seule clé ici mais on peut imaginer l'ajout d'autres clés en fonction de la classe du joueur : warrior, mage, rogue etc.).

4/ **Ajouter** la clé « *Level* » et la mettre à la valeur de « 1 ».

```
{ "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  
  }  
}
```

Par exemple, la **position initiale du joueur** est donnée par les valeurs de clés *Init_x* (en abscisse) et *Init_y* (en ordonnées).

Ce fichier est chargé dans la méthode *setup(self)* et les couples (clé, valeur) sont mises dans l'attribut *attributes* (voir à droite).

5/ Dans le fichier *constants.py*, **ajouter** le chemin relatif du fichier *player.json* dans une variable appelée *PLAYER_CHARACTERISTICS_FILE*.

6/ **Initialiser** correctement les attributs *center_x*, *center_y*, *init_x_pos* et *init_y_pos* à l'aide des données du fichier JSON chargées.

Remarque : bien **tenir compte** de l'**échelle** de la carte (variable *MAP_SCALING*).

```
def setup(self) :  
    # Chargement des textures  
    super().setup()  
  
    # Ouverture du fichier JSON file  
    # Chargement des caractéristiques du joueur  
    f = open(PLAYER_CHARACTERISTICS_FILE)  
    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]
```

Remarque : pour le déplacement du joueur, on ne prendra dans un premier temps que la première texture de l'animation à savoir le joueur en mode *WALK_UP* (valeur 0, voir le fichier *constants.py*) et d'indice 0.

Cela est donné par l'instruction suivante :

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

Rappel : les attributs *center_x* -respectivement- *center_y* sont les coordonnées absolues en abscisse et ordonnées du joueur.

Et enfin, voici la méthode (partielle) `update(self)` qui est appelée à **chaque action du joueur** mais aussi via la méthode `on_update(self)` de la classe `'MyGame'` :

7/ **Compléter** la méthode `update(self)` aux endroits indiqués pour **empêcher le joueur de sortir de la carte**.

Remarque : bien **tenir compte** là aussi de **l'échelle** de la carte (variable `MAP_SCALING`).

```
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.
```

4/ Le fichier `'main.py'`

Ce fichier contient la classe `'MyGame'` qui gère entre autres :

- La **caméra** qui permettra de recentrer le joueur sur la carte à chacun de ses déplacements.
- Le **moteur physique** (basique ici) qui va s'occuper des collisions joueur / carte et de ses déplacements.
- L'Interface **Homme Machine** (IHM) qui permettra d'interagir avec le joueur à l'aide de boutons à cliquer ou menus à sélectionner par exemple.
- La **création** du joueur (instance de la classe `'Player'`).

Voici la méthode `setup(self)` :

L'attribut **camera** vient la classe `'Arcade'` : elle se charge de la vue sur la carte (elle **centrera** le joueur ici) :

L'attribut **sprite_list** vient également de la classe `'Arcade'` : elle se charge de **l'affichage**, des **coordonnées** et du **ciblage** des entités notamment (clic dessus par exemple).
C'est une **liste** d'entités.

La méthode de liste en Python `append(Entity)` permet logiquement d'ajouter une entité.

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

    # Création de la caméra
    self.camera = arcade.Camera(SCREEN_WIDTH, SCREEN_HEIGHT)

    # 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
    self.sprites_list = arcade.SpriteList()

    # Création du joueur
    self.player = Player(PLAYER_FILE, PLAYER_SCALING, PLAYER_WIDTH, \
        PLAYER_HEIGHT, PLAYER_SPRITE_COORDS)

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

8/ **Compléter** la méthode `setup(self)` selon le code ci-dessus. On veillera à bien **respecter l'ordre des instructions**.

Pour la gestion des **collisions** du joueur, il faut créer le **moteur physique**. Il existe différents types de moteurs dans la bibliothèque `'Arcade'`, le plus simple conviendra ici.

9/ **Compléter** la méthode `setup(self)` selon le code à droite. **Adapter les noms des calques à collision avec la carte chargée**.

```
# Création du moteur physique (déplacement et collision du joueur)
walls = [self.scene["ground_collide1"], self.scene["nature_collide2"],
        self.scene["house_collide2"]]
self.physics_engine = arcade.PhysicsEngineSimple(self.player, walls)
```

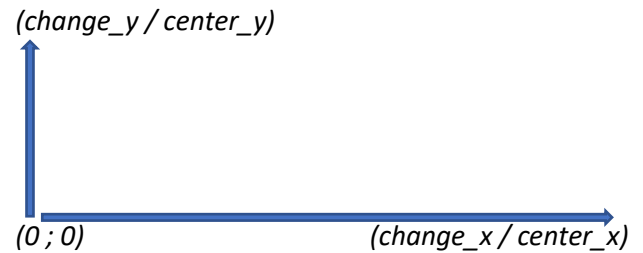
(*) Voici le lien vers les différents moteurs physiques disponibles dans la bibliothèque `'Arcade'`, lien ici :

https://api.arcade.academy/en/latest/api/physics_engines.html#id1

Pour gérer le **déplacement** du joueur (dans un premier temps), on utilisera les **flèches directionnelles du clavier**. Il s'agit ici de **programmation événementielle**, c'est-à-dire réagissant en fonction des **actions** de l'utilisateur.

Repère orthonormé de la bibliothèque `Arcade`

Le **déplacement** du joueur est géré par deux attributs de la classe `Sprite` de la bibliothèque `Arcade`, *change_x* et *change_y* pour ses abscisses et ses ordonnées.



La bibliothèque `Arcade` propose deux méthodes pour la gestion du clavier :

- `on_key_press(self, key, modifiers)` est **appelée** si on **presse une touche de clavier**, le paramètre *key* indique le **type de touche** :
 - o Test d'une pression sur la flèche gauche : `if key == arcade.key.LEFT`
 - o Test d'une pression sur la flèche droite : `if key == arcade.key.RIGHT`
 - o Test d'une pression sur la flèche vers le haut : `if key == arcade.key.UP`
 - o Test d'une pression sur la flèche vers le bas : `if key == arcade.key.DOWN`
- `on_key_release(self, key, modifiers)` est **appelée** lorsqu'une **touche de clavier est relâchée** ou par **intervalles réguliers en cas d'appui continu** sur une touche. Elle met le **déplacement du joueur à zéro**.

Remarque : on ne tiendra pas compte du paramètre *modifiers*.

10/ La vitesse de déplacement joueur étant donnée par la clé *Speed* du fichier JSON correspondant, **écrire** la méthode `on_key_press(self, key, modifiers)` en fonction des touches fléchées pressées en s'appuyant sur l'exemple à droite.

```
def on_key_press(self, key, modifiers):
    # Mouvements du joueur
    if key == arcade.key.LEFT :
        self.player.change_x = -self.player.attributes["Speed"]
```

11/ **Compléter** la méthode `on_key_release(self, key, modifiers)` qui permet de mettre le déplacement du joueur à zéro en s'appuyant sur l'exemple à droite.

```
def on_key_release(self, key, modifiers):
    # Fin de mouvement du joueur
    if key == arcade.key.LEFT :
        self.player.change_x = 0
```

Mise à jour du jeu (update). La caméra et le moteur physique doivent être mis à jour à chaque frame, ce sont des méthodes venant de la bibliothèque `Arcade` :

Mise à jour du **moteur physique** ici :

Mise à jour du **joueur** ici :

Mise à jour des **monstres** ici (liste vide pour l'instant) :

Mise à jour de la **caméra** ici :

```
def on_update(self, delta_time):
    # Déplace le joueur, gère les collisions avec les objets de la map
    self.physics_engine.update()

    # Mise à jour du joueur
    self.player.update()

    # Mise à jour des mobs
    for mob in self.mobs :
        mob.update()

    # Positionne la caméra sur le joueur
    self.center_camera_to_player()
```

12/ **Compléter** la méthode `on_update(self, delta_time)` comme ci-dessus.

Centrage de la caméra sur le joueur :

13/ **Recopier** cette méthode.

```
# Permet de centrer la caméra sur le joueur
def center_camera_to_player(self):
    screen_center_x = self.player.center_x - self.camera.viewport_width/2
    screen_center_y = self.player.center_y - self.camera.viewport_height/2

    if screen_center_x < 0:
        screen_center_x = 0
    if screen_center_y < 0:
        screen_center_y = 0
    player_centered = screen_center_x, screen_center_y

    self.camera.move_to(player_centered)
```

Affichage de la carte et du joueur :

14/ **Compléter** également cette méthode.

```
def on_draw(self):
    # Efface l'écran
    self.clear()

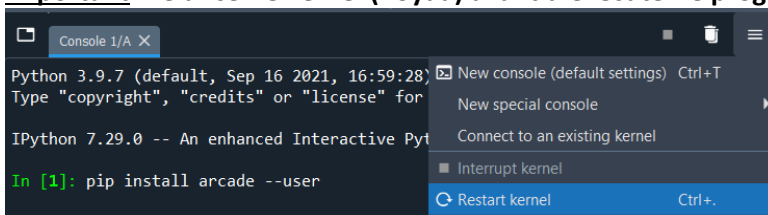
    # Activation de la caméra
    self.camera.use()

    # Affichage de la scène (map)
    self.scene.draw()

    # Affichage des mobs
    self.sprites_list.draw()
```

Cette partie du programme est terminée.

Important : relancer le kernel (noyau) avant d'exécuter le programme.



15/ **Exécuter** le programme et déplacer le joueur avec les touches fléchées du clavier pour **vérifier** si tout fonctionne.

Remarque : ne pas hésiter à changer la valeur de l'attribut texture de la classe `Player` pour vérifier si les textures sont bien chargées.