Chargement du joueur du jeu avec *Arcade*

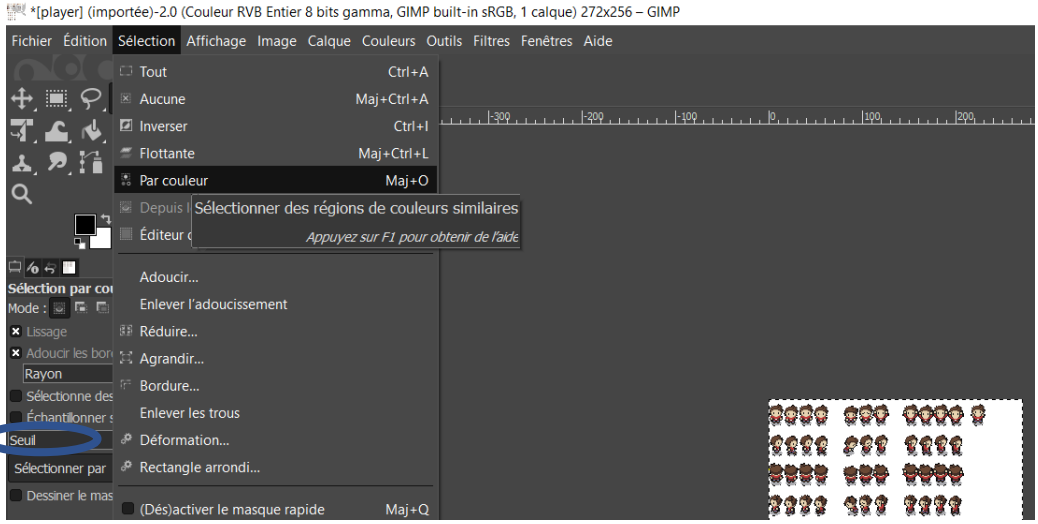
I/ Préparation du joueur

1/ Avec le logiciel GIMP

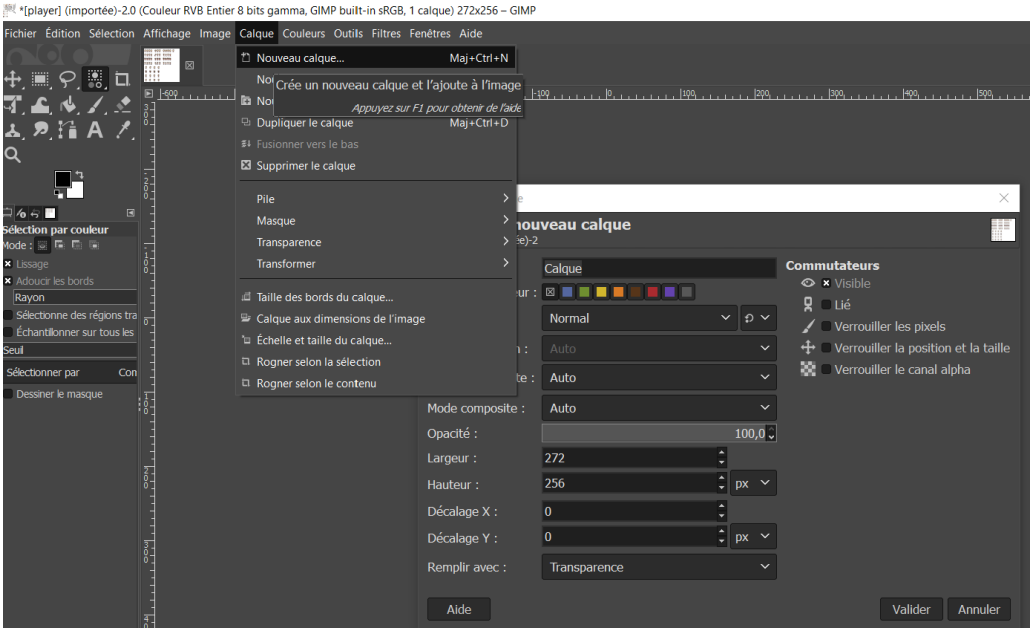
Tout comme la map, il faut rendre la couleur affichée non transparente.

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

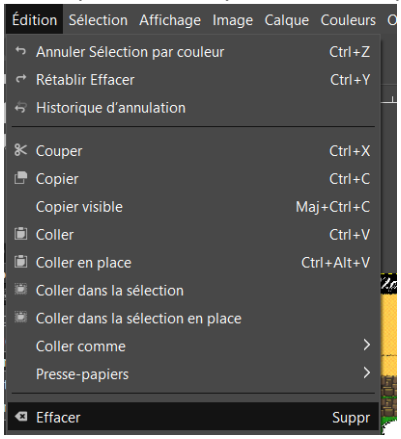
2/ **Opérer** une sélection par couleur (***cliquer sur la couleur à rendre transparente***) et mettre le **seuil** au minimum (encadré en bleu sur l'image ci-dessous).
Remarque : les zones de la couleur concernée doivent être entourées de 'pointillés'.



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

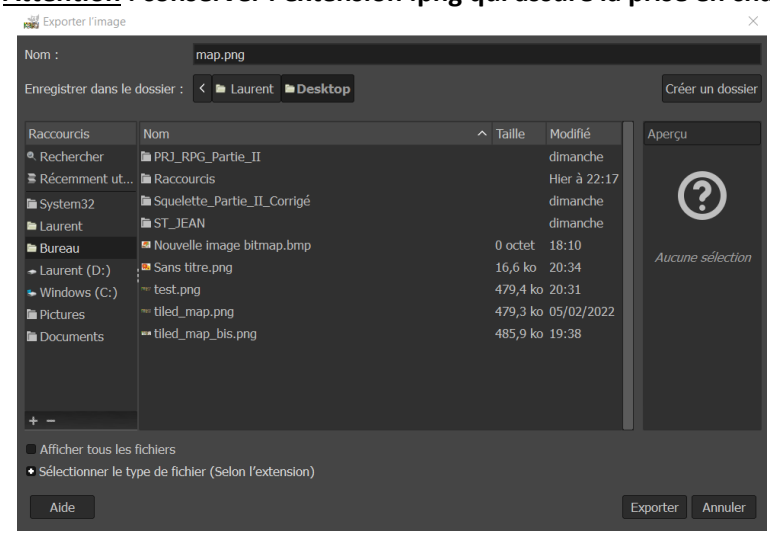


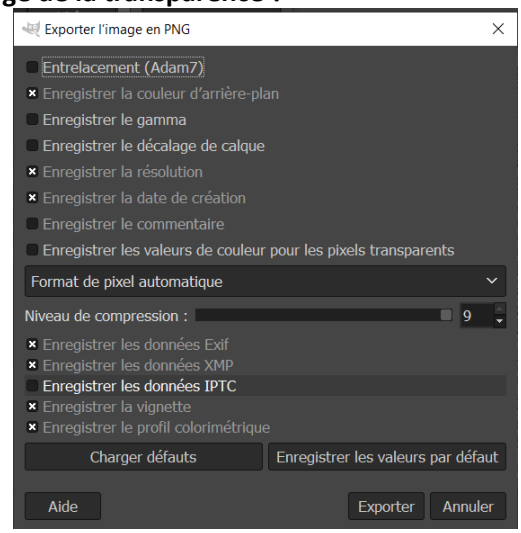
4/ Dans le menu *Edition*, **cliquer** sur *Effacer* : la couleur sélectionnée devient transparente.
Remarque : la transparence est représentée par des carrés grisés.



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*** (ou ***Visual studio***) et **charger** les fichiers « squelette ».

Important : POUR SPYDER UNIQUEMENT : 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* **<u>en fonction du joueur choisi</u>**

```
################################### PLAYER ###################################

# Caractéristiques du joueur
PLAYER_WIDTH,PLAYER_HEIGHT = 32, 32
PLAYER_SCALING = 2
PLAYER_FILE = "Mobs/Player/player.png"          # Fichier des animations
PLAYER_CAR_FILE = "Mobs/Player/player.json"     # Fichier des caractéristiques


# Coordonnées des images de chaque animation du joueur
##### A COMPLETER / MODIFIER si autre image #####
PLAYER_WD_COORDS = [(0,0), (32,0), (64,0), (96,0)]  # Marche vers le bas
PLAYER_WU_COORDS = []                               # Marche vers la droite
PLAYER_WR_COORDS = []                               # Marche vers le haut
PLAYER_WL_COORDS = []                               # Marche vers la gauche


##### NE PAS CHANGER #####
PLAYER_SPRITE_COORDS = [ PLAYER_WD_COORDS, PLAYER_WL_COORDS, PLAYER_WR_COORDS, PLAYER_WU_COORDS ]
```

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

<u>2/ Le fichier `entity.py`</u>

La méthode *setup(self)* est déjà complétée. Elle permet de charger les images d'animations des entités.

<u>3/ Le fichier `player.py`</u>

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/lmayer65/NSI_T/blob/main/Projets/Projet_RPG/Pr%C3%A9sentation_Programme.pdf
(Page 7 à 9).

Voici le fichier *player.json* qui regroupe les **caractéristiques** du joueur. Il s'agit d'un <u>dictionnaire de dictionnaires</u> (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_CAR_FILE (si besoin).
Compléter les instructions manquantes

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


    # Ouverture du fichier JSON file
    # Chargement des caractéristiques du joueur
    f = open(PLAYER_CAR_FILE)
    data = json.load(f)

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

    # Fermeture du fichier
    f.close()
```

6/ **Initialiser** correctement les attributs *center_y*, *init_pos_y* en suivant l'exemple pour *center_x* et *init_pos_x*.
Remarque **:** bien **tenir compte** de l'**échelle** de la carte (variable *MAP_SCALING*).

```
# Position / Etat de départ
self.center_x = self.attributes['Init_x'] * MAP_SCALING
self.init_x_pos = self.center_x
##### A COMPLETER #####
```

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) :
    ######################### DEPLACEMENT ##########################

    # Le joueur doit rester sur la map
    # Les abscisses, ne pas dépasser la largeur de la map
    # ATTENTION à l'échelle de la carte (MAP_SCALING)
    ###### A COMPLETER ######

    # Puis les ordonnées, ne pas dépasser la hauteur de la map
    ###### A COMPLETER ######
```

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'**I**nterface **H**omme **M**achine (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.

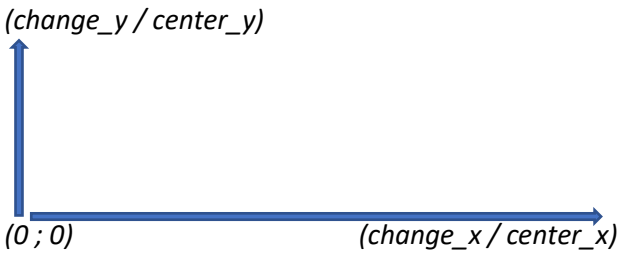
```
def setup(self):
    # Couleur de fond de la map.
    arcade.set_background_color(arcade.csscolor.CORNFLOWER_BLUE)

    # 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)
    self.map.set_collisions()

    # Liste des mobs à afficher
    self.sprites_list = arcade.SpriteList()


    # Création du joueur
    ##### A COMPLETER #####

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

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

Pour gérer le **déplacement** du joueur, 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.

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.

<u>*Repère orthonormé de la bibliothèque `Arcade`*</u>



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** :
  - <u>Test d'une pression sur la flèche gauche</u> : *if key == arcade.key.LEFT*
  - <u>Test d'une pression sur la flèche droite</u> : *if key == arcade.key.RIGHT*
  - <u>Test d'une pression sur la flèche vers le haut</u> : *if key == arcade.key.UP*
  - <u>Test d'une pression sur la flèche vers le haut</u> : *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**.

<u>Remarque</u> : on ne tiendra pas compte du paramètre *modifiers*.

9/ 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 : # Vers la gauche
        self.player.change_x = -self.player.attributes['Speed']
    ##### A COMPLETER #####
    elif key == arcade.key.LEFT : # Déciblage d'un monstre
        self.gui.update()
```

10/ **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
    ##### A COMPLETER ######
```

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()
```

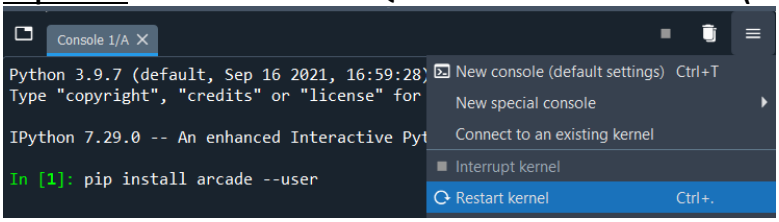
11/ **Compléter** la méthode *on_update(self, delta_time)* comme ci-dessus.

Affichage de la carte et du joueur :

12/ **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()
```

<u>Important</u> : POUR SPYDER UNIQUEMENT : relancer le kernel (noyau) avant d'exécuter le programme.



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

<u>Appeler le professeur pour vérification</u>