

INFOB318  
PROJET INDIVIDUEL  
VINCENT ENGLEBERT

UNIVERSITÉ DE NAMUR  
ANNÉE ACADÉMIQUE 2019-2020

---

Ca\_\_py\_\_taineDurable

Programmer's guide

Version 1.0.

---



Promoteur : VOISIN Adrien - Étudiant : POLET William

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Sources</b>	<b>3</b>
<b>3</b>	<b>Environnement de développement et pré-requis</b>	<b>4</b>
3.1	PyCharm . . . . .	4
3.2	Pygame . . . . .	4
3.3	Tiled . . . . .	6
3.4	Autres outils . . . . .	6
<b>4</b>	<b>Code</b>	<b>7</b>
4.1	Arborescence . . . . .	7
4.2	Exécution du fichier main . . . . .	7
4.3	Fichiers globaux . . . . .	8
4.3.1	globvars.py . . . . .	8
4.3.2	settings.py . . . . .	8
4.3.3	persistence.py . . . . .	8
4.3.4	capylib.py . . . . .	9
4.4	Menu() . . . . .	10
4.5	Game() . . . . .	11
4.5.1	tilemap.py . . . . .	12
4.5.2	sprites.py . . . . .	13
4.5.3	inventory.py . . . . .	15
4.5.4	story.py . . . . .	15
4.5.5	game_display.py . . . . .	17
4.5.6	Boucler sur Game . . . . .	18
<b>5</b>	<b>Annexe</b>	<b>19</b>
<b>6</b>	<b>Bibliographie</b>	<b>20</b>

# 1 Introduction

Le développement durable peut être défini comme un développement qui répond aux besoins du présent sans compromettre la capacité des générations futures de répondre aux leurs. Afin de répondre à cette crise environnementale, sociale et économique, l'UNEP a structuré le développement durable autour de 17 objectifs à remplir. Le but de ce projet, intitulé `Ca__py__taineDurable`, a été de développer un jeu vidéo écrit en Python permettant de sensibiliser et de former les plus jeunes (10-14 ans) à ces 17 objectifs. Ce projet a été réalisé en tant que projet individuel de fin de bachelier de la faculté d'informatique de l'université de Namur.

Les **17 objectifs de développement durable**, que j'appellerai par la suite **ODD**, ont été définis par l'**UNEP** (United Nations Environment Programme) pour parvenir à un avenir meilleur et plus durable d'ici 2030.<sup>[1]</sup> Pour faire comprendre ces 17 ODD aux enfants, l'idée a été de faire un jeu dans lequel le joueur devra compléter des quêtes qui sont chacune liées à un ODD. Le joueur pourra à tout moment voir l'objectif en lien avec la quête actuelle et en apprendre plus sur celui-ci.

Dans ce guide je vais commencer par décrire les technologies et outils utilisés pour la réalisation du projet et expliquer certaines notions que j'utiliserai par la suite pour expliquer le programme en lui-même.

## 2 Sources

Lien Github : [github.com/UNamurCSFaculty/1920\\_INFOB318\\_Ca-py-taineDurable](https://github.com/UNamurCSFaculty/1920_INFOB318_Ca-py-taineDurable)

Structure du projet :

- bin
  - build
  - CapytaineDurable
    - ...
    - CapytaineDurable.exe
  - compile.bat
- code
  - img
  - maps
  - media
  - tests
  - main.py
  - \*.py
  - capytaine.cfg
  - capytaine.save
  - requirements.txt
- doc
  - programmer's guide
  - user's guide
  - tools
- legacy
- planning
  - gantt.xlsx
- poster
  - source
  - poster.pdf
- .gitignore
- README.md

## 3 Environnement de développement et pré-requis

Avant de commencer, le premier pré-requis pour réaliser ce projet est évidemment le langage de programmation, python, et ses aspects orientés objets. Plusieurs cours sont disponibles sur OpenClassrooms pour se rafraîchir la mémoire si nécessaire.

Ensuite, sont expliqués ci-dessous toutes les technologies et les outils utilisés pour la conception du projet.

### 3.1 PyCharm

Ce projet a été réalisé avec l'éditeur pycharm version 2019.3.4. Le code du projet respecte au maximum le standard PEP8 à l'exception de 3 règles. La première est la taille maximale des lignes (72 caractères) car je n'aime pas couper des lignes de code mais j'essaie qu'elles soient les plus courtes possible. La deuxième est le fait de toujours mettre un espace après une virgule, ce que je respecte tout le temps sauf pour les tuples symbolisant une coordonnée (x,y). Enfin la dernière règle non respectée est la présence de plusieurs "import \*" qui sont fortement déconseillés mais les seuls qui sont présents dans les différents fichiers du projet sont utilisés uniquement sur le fichier **settings.py** qui ne comprend que des constantes. Comme la plupart des fichiers utilisent bon nombre de ces constantes le "import \*" s'est avéré être la solution la plus simple et ce n'est pas tellement dangereux sur ce genre de fichier.

Toutes les librairies utilisées par le programme ont été inscrites dans le fichier **requirements.txt** grâce à la commande "pip freeze". Vous pouvez installer directement toutes ces librairies à l'aide de la commande "pip install -r requirements.txt".

### 3.2 Pygame

Pygame est la librairie qu'il est absolument nécessaire de maîtriser si l'on veut modifier ce programme et plus généralement, si l'on veut créer un jeu vidéo en python, notamment tout le côté graphique. Pour apprendre à utiliser cette librairie vous pouvez commencer par un petit cours sur OpenClassrooms<sup>[2]</sup> pour apprendre les notions de base mais personnellement j'ai appris le plus à l'aide des leçons **KidsCanCode** et de sa playlist youtube "Game Development with Pygame".<sup>[3]</sup>

Je vais tout de même expliquer ci-dessous les notions de base de pygame pour pouvoir les utiliser par la suite dans ce guide.

#### Surface et Rect :

Tout l'aspect graphique de pygame est géré à l'aide de 2 types d'objets. Le premier, créé via la méthode **pygame.Surface(largeur, hauteur)**, sont les objets qui seront affichés à l'utilisateur. Toutes les images sont des surfaces mais on peut également créer des surfaces et les remplir d'une couleur grâce à la méthode **fill((R,G,B))**, utile par exemple pour repartir d'un écran noir.

Il existe également une surface unique, **Display**, qui est la fenêtre du programme. Cet objet doit être initialiser juste après l'initialisation de pygame avec la méthode **pygame.display.set\_mode(largeur, hauteur)**

Le second type d'objets est **Rect**, créé via la méthode **pygame.Rect(position X, position Y, hauteur, largeur)**. Ces rectangles sont utilisés pour positionner les surfaces grâce aux coordonnées (x,y). Notez que les coordonnées sont en pixels avec l'origine (0,0) dans le coin supérieur gauche et, par déduction, la coordonnée du coin inférieur droit est (hauteur, largeur). Quand on parle de la position d'un objet Rect, on parle donc de son coin supérieur gauche (.topleft).

Pour construire l'écran à afficher à l'utilisateur, on colle toutes nos images sur la surface **Display** avec la méthode **blit()** qui prend en argument l'image à coller et une coordonnée (x,y) directement ou un objet Rect (son point "topleft").

De manière générale, pour afficher une image on la charge (**img = pygame.image.load(...)**), on crée un objet **Rect** à partir de cette surface pour avoir les mêmes dimensions que celle-ci (**rect = img.get\_rect()**), on positionne le rectangle où l'on souhaite afficher l'image (**rect.center = (50,50)**) puis on colle cette image sur une autre surface à la bonne coordonnée (**display.blit(img, rect)**).

**Attention :** on peut utiliser la méthode blit sur n'importe quelle surface, autre que l'objet Display, mais à ce moment là l'origine (0,0) est le coin supérieur gauche de cette surface et pas de la fenêtre du jeu donc pensez à bien positionner les rectangles par rapport à la surface en question.

Pour terminer, une fois toutes les surfaces collées sur l'objet Display, on l'affiche en appelant la méthode **flip()** -> **pygame.display.flip()**.

### Events :

Les événements de Pygame servent à gérer les inputs de l'utilisateur. Chaque méthode du programme garde la même structure pour gérer les différents types d'événements qui sont au nombre de 4 :

- QUIT : quand l'utilisateur clique sur la croix rouge pour fermer la fenêtre
- MOUSEMOTION : quand il déplace la souris
- MOUSEBUTTONDOWN : quand il clique à un endroit, l'événement se déclenche quand il relâche le clic.
- KEYDOWN : quand il appuie sur une touche du clavier

### Autres modules :

Les autres modules de pygame utilisés dans le programme sont les suivants :

- **key.set\_repeat()** : permet de définir si le joueur peut maintenir une touche appuyée ou pas. Cette option est activée en jeu pour permettre au joueur de maintenir une touche appuyée pour se déplacer. Dans les menus, l'option est désactivée.
- **mouse.set\_visible(Bool)** : True -> souris visible dans les menus, False -> souris invisible dans le jeu.
- **mixer.music** : permet de charger et gérer une musique.<sup>[4]</sup> La musique est coupée automatiquement dans le menu principal et en jeu le joueur peut la couper via le menu pause.

### 3.3 Tiled

Pour créer les maps du jeu j'ai utilisé le programme Tiled version 1.2.5<sup>[5]</sup>, c'est un outil assez facile à prendre en main. J'ai appris à l'utiliser grâce aux vidéos youtube de **KidsCanCode** sur le **Tile-based Game**. Je vous conseille de regarder ces vidéos pour apprendre cet outil mais pour la suite du guide sachez qu'il permet de créer différents tileset (ensemble de tiles) à partir d'images, à condition que leurs tiles soient de la même taille (32x32 dans notre cas) et de créer plusieurs couches pour placer et donc superposer des tiles ainsi que des objets. Les différentes couches de tiles vont être chargées et aplaties pour créer une image de la map et les objets, invisible sur la map, vont permettre de créer des objets python.

Toutes les maps créées sont stockées en fichier ".tmx" dans "code/maps" et les tilesets utilisés pour créer ces maps sont stockées dans "code/maps/assetsPack".

Vous pouvez trouver des assets gratuits sur les sites repris dans la bibliographie.<sup>[6]</sup>

### 3.4 Autres outils

#### **Alferd Spritesheet Unpacker :**

Cet outil permet de couper une image importée en plusieurs images, typiquement obtenir des images de 32x32 à partir d'un tileset. Notamment utilisé pour obtenir toutes les images des PNJ qui étaient à la base toutes sur une même image.

#### **paint.net :**

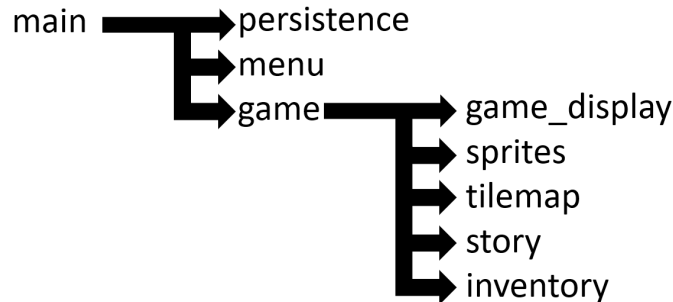
J'ai utilisé paint.net pour modifier certaines images car je trouvais cet outil simple et suffisamment complet pour ce que j'avais à en faire.

Point intéressant avec Tiled : vous pouvez modifier avec paint.net l'image source d'un fichier ".tsx" et, une fois la modification enregistrée, le tileset dans Tiled sera directement mis à jour.

## 4 Code

### 4.1 Arborescence

Les fichiers repris ci-dessous définissent des classes et l'arborescence montre dans quel fichier ces classes sont instanciées.



Tous ces fichiers, ainsi que les autres non repris ci-dessus, sont détaillés plus bas.

### 4.2 Exécution du fichier main

En lançant l'exécution de **main.py** le programme commence par initialiser la classe **Persistence** qui s'occupe de récupérer les paramètres de configuration du jeu et les différentes parties sauvegardées. Cette classe est initialisée dans une des variables de **globvars.py** qui reprend plusieurs variables utilisées dans différents fichiers du programme.

Le programme continue en initialisant **Pygame**, ce qui est nécessaire si on veut utiliser ses différents modules, ainsi que son attribut **display** (la fenêtre du jeu). l'objet **Display** est initialisé avec le mode d'affichage sauvegardé dans le fichier de configuration, à savoir soit en mode fenêtré soit en mode plein écran.

Vient ensuite la création de l'objet **Menu** et la boucle principale du **main**. La boucle principale qui comprend une autre boucle, en effet, le programme utilise deux classes principales, **Menu** et **Game**. La première s'occupe de l'écran de démarrage et de la création/chargement d'une partie pour ensuite passer la main à la seconde qui s'occupe du déroulement d'une partie en elle-même. L'imbrication de la boucle de **Game** permet de pouvoir revenir au menu principal, avec la même instance de **Menu**, quand l'utilisateur quitte la partie, alors que l'objet **Game** est réinitialisé à chaque lancement d'une nouvelle partie ou d'une partie sauvegardée avec les paramètres correspondants.

Le programme crée l'écran à afficher en fonction des interactions de l'utilisateur et rafraîchit cet écran uniquement lors d'un nouvel input. Tous les événements possibles de l'utilisateur sont gérés grâce à différentes méthodes **\*\_events()** qui sont chacune appelée suivant l'état dans lequel se trouve le programme. On peut voir dans le main une variable **active\_screen** qui permet de connaître cet état. Que ce soit dans le menu principal ou en jeu, on peut avoir plusieurs écrans (écran de chargement d'une partie, de sauvegarde, écran de pause,...) et j'ai essayé de décomposer le plus possible pour que chaque écran ait sa propre méthode qui gère les inputs possibles de l'utilisateur.

Enfin, notons que l'utilisateur peut fermer le programme à partir du menu principal mais également quand il est en jeu sans revenir dans le menu, ce qui a pour effet de quitter la boucle du jeu ainsi que la boucle principale.



## 4.3 Fichiers globaux

Pour commencer je vais expliquer ici les fichiers qui sont utilisés à plusieurs endroits dans le programme pour ensuite pouvoir détailler les classes **Menu** et **Game**.

### 4.3.1 globvars.py

Dans ce fichier sont définis plusieurs variables utilisées dans différents fichiers du programme.

- `screen` : variable objet `Display` de PyGame défini dans le `main` et modifié quand l'utilisateur change de mode d'affichage dans le menu principal.
- `persistence` : variable objet `Persistence` initialisé dans `main` et utilisé dès qu'on veut charger ou sauvegarder des données du jeu.
- `active_screen` : `String` gardant le nom de l'écran courant. `user_selection` : variable entière indiquant l'option actuellement sélectionnée par l'utilisateur dans les écrans nécessitant une sélection. Remis à 0 dès qu'on change d'écran.
- `saved_games` : liste de toutes les parties sauvegardées.
- `user_config` : dictionnaire gardant le mode d'affichage (par défaut : fenêtré) et celui de la musique (par défaut : allumé). Sauvegardé dans `'capytaine.cfg'` (voir **Persistence**).

### 4.3.2 settings.py

Dans ce fichier sont définies toutes les constantes utilisées dans le programme.

#### Attention

- `WIDTH` et `HEIGHT` = les dimensions de la fenêtre à modifier avec précautions.
- `TILESIZE` : ne pas modifier cette constante car les tiles des maps et les sprites sont de taille 32x32.
- `GAME_STARTUP_PARAMS` = tous les paramètres du jeu qui sont sauvegardés. Si vous modifiez la structure de ce dictionnaire, n'oubliez pas de modifier également le dictionnaire dans **Persistence.save\_this\_game()**.

### 4.3.3 persistence.py

Cette classe utilise le module python **Pickle** pour sérialiser ou dé-sérialiser certaines données du jeu.

Commençons avec le paramètre de configuration `user_config` qui est sauvegardé **automatiquement** dans le fichier `capytaine.cfg` quand le programme est arrêté et qui est rechargé au lancement du programme lorsque l'objet **Persistence** est initialisé. Cela me paraissait évident de sauvegarder automatiquement ce paramètre pour le mode d'affichage pour que le jeu se lance directement soit en fenêtré soit en plein écran selon le souhait de l'utilisateur et pour ce qui est de la musique, cela peut paraître bizarre d'avoir un jeu qui se lance tout le temps sans musique mais de nouveau c'est selon le souhait de l'utilisateur.

Les autres données du jeu sauvegardées sont évidemment les paramètres permettant de recharger l'état d'une partie. Pour ce faire on utilise la méthode **save\_this\_game** qui enregistre tous les paramètres de l'objet **Game** courant, en respectant la structure du dictionnaire comme défini dans **settings.py**.

Notons que la variable **SAVESLOTS** est une constante définie dans **settings.py** et fixée à 6. Elle correspond au nombre de sauvegardes disponibles et donc au nombre de lignes affichées à l'écran de chargement et de sauvegarde. Cette constante peut être modifiée sans problème, il n'y a pas de limite au nombre d'emplacements de sauvegarde, mais il faut alors modifier les 2 méthodes affichant ces sauvegardes.

#### 4.3.4 capylib.py

Ce fichier regroupe plusieurs méthodes utilisées dans différents fichiers du programme.

**draw\_text()** et **draw\_multilines\_text()** sont 2 méthodes permettant d'écrire du texte sur une surface mais la première ne peut écrire un texte que sur une ligne (si le texte est trop grand il dépassera) et toujours en utilisant l'objet **Display** comme surface tandis que la deuxième peut écrire sur n'importe quelle surface et va à la ligne avant de sortir du cadre.

## 4.4 Menu()

La classe **Menu** s'occupe de l'affichage et des événements relatifs au démarrage du programme et au lancement d'une partie. L'objectif du menu est d'arriver à ce que l'utilisateur détermine les paramètres nécessaires à la création de l'objet **Game** que ce soit les paramètres par défaut ou ceux d'une partie sauvegardée.

### Écran principal :

Toutes les options affichées à l'écran sont celles définies dans **MAINMENU\_ITEMS** dans **settings.py**. Si on rajoute une option dans cette liste, elle sera directement affichée à l'écran mais il ne faudra pas oublier de gérer le cas où l'utilisateur sélectionne cette option dans la méthode **main\_menu\_validation\_handler()**.

### Écran de sélection de l'avatar :

En sélectionnant **Nouvelle partie** les paramètres de la future partie sont ceux par défaut mais il reste à définir l'avatar du joueur qui est défini à l'aide de la variable **user\_selection** permettant de savoir quel élément l'utilisateur a choisi. Ce qui permet de charger toutes les images de l'avatar sélectionné lors de la création du jeu (voir classe **Player**).

### Écran de chargement :

L'autre possibilité pour lancer une partie est d'en charger une sauvegardée. De la même manière que pour la sélection de l'avatar, la partie chargée est sélectionnée à l'aide de la variable **user\_selection** qui, si elle est comprise entre 0 et le nombre de sauvegardes possibles (**SAVESLOTS**), charge les paramètres de la partie correspondante.

Dans les 2 cas les paramètres du jeu sont alors définis et peuvent être passés en argument à la classe **Game** pour être initialisée.

## 4.5 Game()

L'objectif de cette classe est de créer tous les éléments du jeu et l'écran à afficher en fonction des inputs de l'utilisateur. C'est la classe principale du jeu qui va initialiser toutes les autres classes nécessaire au bon déroulement d'une partie. Les classes disponibles dans les fichiers **tilemap.py** et **sprites.py** vont servir à construire respectivement la map et les sprites (PNJ ou objets) présents sur cette dernière. Les classes présentes dans les fichiers **story.py** et **inventory.py** vont être utilisées pour gérer les éléments du gameplay, à savoir les quêtes, les objectifs et l'utilisation d'outils. Enfin la classe **GameDisplay** va utiliser tous ces éléments pour construire l'écran à afficher au joueur.

### Initialiser Game() :

Le programme commence par rendre la souris invisible et permettre au joueur de maintenir appuyée une touche, 2 options qui seront changées à chaque fois qu'on rentre dans un menu et de nouveau changées quand on revient en jeu.

Le programme continue en définissant quelques variables utilisées dans différentes méthodes de la classe puis une série de variables booléennes qui vont servir à savoir quel écran afficher à chaque appel à la méthode **display()**.

Viens ensuite la création des différentes variables objets à qui on va passer en argument l'objet **Game** lui-même. Notons que 'map' et 'camera' sont instanciés dans la méthode **load\_map()** mais ces 2 variables doivent tout de même être initialisées dans le **\_\_init\_\_** de la classe. La méthode **load\_map()** justement est appelée tout à la fin de l'init avec comme paramètre le nom de la map par défaut si c'est une nouvelle partie ou le nom de la map sauvegardée sinon.

Détaillons maintenant les classes utilisées par la classe **Game** avant d'expliquer l'utilisation de cette dernière.

### 4.5.1 tiledmap.py

La classe **TiledMap** va construire la map à partir du fichier ".tmx" du même nom en créant un objet **Surface** reprenant chaque couche du fichier ".tmx" pour ainsi être utilisé par la suite comme une simple image sans devoir recoller tous les éléments de la map à chaque fois.

Pour trouver le bon fichier ".tmx" à charger, la classe a besoin du nom de la map mais également de sa version. La variable **map\_version** est, par défaut, un caractère nul mais peut prendre une valeur en fonction de l'avancement du scénario.

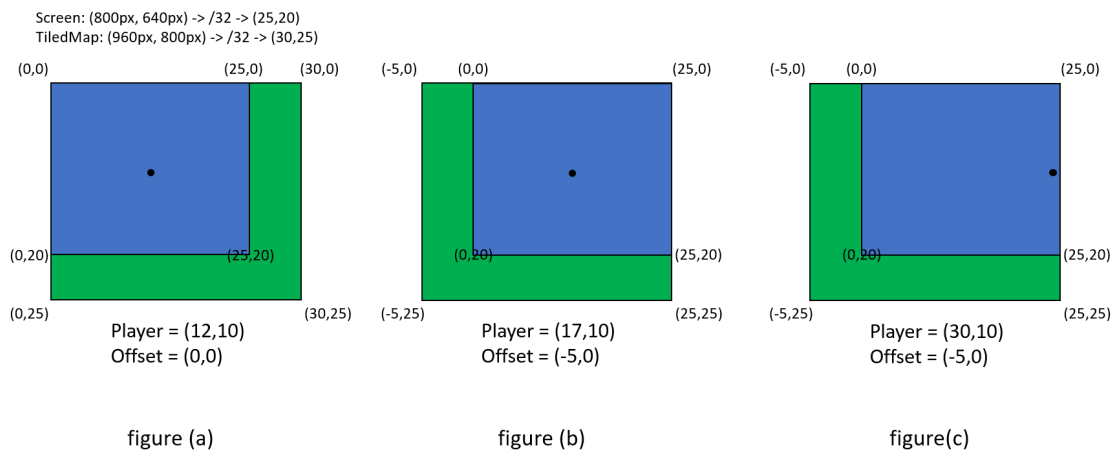
Exemple : "mapD" -> map\_version devient '2' => "mapD2". "mapD.tmx" et "mapD2.tmx" étant 2 fichiers différents mais relatifs au même village.

la classe **Camera** est utilisée pour calculer le décalage à appliquer à la map en fonction de la position du joueur. En effet, l'objectif étant de garder le joueur au centre de l'écran, pour les maps plus grande que celui-ci, lorsque le joueur se déplace c'est en réalité la map qu'on fait bouger dans le sens inverse.

Exemple ci-dessous avec l'écran en bleu qui fait 25 tiles sur 20 et la map en vert qui en fait 30 sur 25. Le joueur se situe initialement en (12,10) (figure a).

**Attention :** la position du joueur est relative à l'origine (0,0) de la map, pas de l'écran.

Il se déplace de 5 cases vers la droite (figure b). On peut voir que le joueur en réalité ne bouge pas et reste au centre de l'écran mais que la map, elle, est décalée de 5 cases vers la gauche. Par contre, une fois que l'écran a atteint le bord de la map, si le joueur continue vers la droite il ne sera plus au centre de l'écran (figure c).



A l'inverse, si la map est plus petite que l'écran, celle ci est centrée et il n'est pas nécessaire de la décaler. C'est le cas des maps intérieures comme l'hôpital.

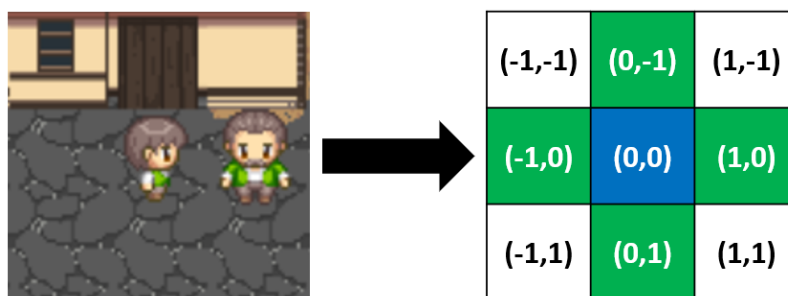
La méthode **update()** de cette classe est utilisée pour calculer le décalage à appliquer à la map tandis que la méthode **apply()** est utilisée sur tous les objets au moment de l'affichage pour les placer sur la map avec le même décalage que celle-ci.

### 4.5.2 sprites.py

Les différentes classes présentes dans ce fichier permettent d'instancier tous les sprites possible du jeu. Notons l'utilisation de groupes de sprites dans les méthodes d'initialisation de ceux-ci. Ces groupes permettent d'utiliser certaines méthodes sur tous les sprites du même groupe. Comme la méthode `get_image()` qui est utilisée sur le joueur, les PNJs et les objets sans savoir précisément sur quel type d'objet elle est appelée.

Tous ces objets sont créés dans la méthode `load_map()` de la classe **Game** en fonction des objets présents dans le fichier ".tmx" de la map à créer. **Attention** : les paramètres de position passés en argument lors de la création de ces instances sont en pixels (notés `x_px` et `y_px`) car ce sont des propriétés venant de l'éditeur Tiled. Cependant, dans les méthodes d'initialisation de ces instances, ces paramètres sont divisés par **TILESIZE** pour avoir le numéro de la case comme position. Excepté pour **Player** qui prend directement les paramètres de position du jeu qui ne sont pas en pixels, c'est d'ailleurs pour cette raison qu'il n'est pas nécessaire de créer un objet dans Tiled pour le joueur.

Classe **Player** :



- Se déplacer : Le joueur ne peut se déplacer que horizontalement et verticalement et à conditions qu'il ne sorte pas de la carte et qu'il ne rentre pas en collision avec un autre objet. Dans l'image ci-dessus, on peut voir qu'une maison (**Obstacle**) se trouve au dessus de lui et qu'un PNJ (**NPC**) se trouve à sa droite donc les seuls mouvements possibles pour lui sont ceux l'amenant sur la case de gauche ou celle du dessous.
- Attribut `look_at` : permet de savoir vers quelle case le joueur regarde. Cet attribut est toujours égal à une des valeur présente sur les cases vertes dans l'image. Sur l'image, le joueur se trouve en (0,0) mais s'il se trouve sur une autre case il suffit d'additionner les valeurs x et y des 2 tuples pour avoir la case vers laquelle il regarde. Exemple : `Player.pos = (12,10)` et `Player.look_at = (1,0)` => il regarde vers la case (13,10). Cet attribut est utilisé pour savoir quelle image du joueur affiché (4 positions différentes) mais également pour interagir avec un PNJ ou un objet.
- Interagir : Pour interagir avec un PNJ ou un objet, le joueur ne doit pas seulement se trouver à coté de lui mais il doit aussi regarder dans sa direction. On peut facilement tester l'égalité entre la case du PNJ ou de l'objet et la case vers laquelle le joueur regarde grâce à l'attribut `look_at`.





Comme dit plus haut, la classe **Player** est la seule à ne pas avoir d'objet correspondant dans l'éditeur Tiled étant donné que sa position n'est pas définie en fonction de la map à sa création. Quand le joueur rentre dans une nouvelle map et donc que le programme charge cette map, le joueur est placé sur la case de l'objet **Gate** portant le nom de la map d'origine. Exemple : le joueur est dans la map A et rentre dans la porte qui l'amène dans la map B, il se trouvera alors sur l'objet **Gate** de la map B qui sert à se rendre en map A.

Pour ce qui est des autres sprites, ils ont tous un objet correspondant dans Tiled qu'on peut différencier par l'attribut **type**. Ci-dessous la liste des propriétés à respecter dans Tiled pour pouvoir créer l'objet désiré.

- NPC :
  - Nom : l'identifiant de l'objet
  - Type : "npc" suivis d'un numéro utilisé pour charger les images correspondantes dans le dossier image. Exemple : "npc16"
  - look\_to : similaire à l'attribut **look\_at** de **Player** sauf qu'il n'est pas possible de créer un tuple dans Tiled et est donc égale à une lettre parmi les suivantes : z, q, s ou d. Chaque lettre étant la touche sur laquelle appuyer pour faire le déplacement correspondant au tuple souhaité.
- Object :
  - Nom : l'identifiant de l'objet
  - Type : le type de l'objet (tree, rock,...) suivis de la version. Exemple : il existe plusieurs images pour un arbre => tree1, tree2,...
- Gate :
  - Nom : l'identifiant de l'objet = le nom de la map de destination
  - Type : "gate"
  - (dx,dy) : le déplacement nécessaire pour rentrer dans la nouvelle map une fois sur la case de l'objet **Gate**. Comme on ne peut pas définir un tuple dans Tiled, dx et dy sont 2 propriétés au lieu d'une.
- Obstacle :
  - Nom : /
  - Type : "obstacle"

### 4.5.3 inventory.py

Pour interagir avec les objets présents sur la carte, le joueur a besoin d'outils. Il y a 4 outils, chacun utilisé pour un type d'objet en particulier.

- Type "tree" : la **hache** 
- Type "dirt" : la **pelle** 
- Type "rock" : la **pioche** 
- Type "trash" : le **filet** 

Ces outils sont des instances de la classe **Tool** et sont gérés par la classe **Inventory**, toutes deux présentes dans le fichier **inventory.py**.

Pour pouvoir utiliser un outil, le joueur doit l'avoir acquis et s'en être équipé. Ces deux conditions sont implémentées par les deux variables booléennes **acquired** et **in\_hand**. La deuxième variable est passée à True quand l'utilisateur appuie sur la touche correspondant à l'outil en question, sous la condition que la première variable soit également à True. Le joueur acquiert les outils en avançant dans le scénario du jeu, scénario géré dans le fichier **story.py** décrit ci-dessous.

### 4.5.4 story.py

Pour rappel, l'objectif du projet étant de faire découvrir à des enfants les objectifs de développement durable de l'ONU, l'idée a été ici que le joueur réalise plusieurs quêtes qui sont chacune liée à un ODD.

#### Les objectifs :

De manière concrète, ces ODD sont stockés sous forme de dictionnaire dans le fichier **data.py** avec les éléments suivants :

- title : le nom de l'ODD
- expl : une courte explication
- advice : un conseil que les enfants peuvent appliquer au quotidien
- facts : une liste de faits, de stats.
- color : chaque ODD a sa propre couleur et ce code est utilisé pour créer une surface de la bonne couleur.

Tous ces éléments sont tirés du site officiel des Nations Unies.

Une fois en jeu, l'utilisateur va pouvoir ouvrir le livret des objectifs pour pouvoir parcourir ces ODD affichés à ce moment là sous forme de petit bandeau créé avec la méthode **generate\_card**. Il pourra ensuite sélectionner un de ces bandeaux pour ouvrir l'objectif correspondant et voir la fiche complète créée avec la méthode **generate\_sheet**.

A ce niveau là les écrans relatifs aux ODD sont un peu statiques mais ce qui les rend plus interactifs c'est la possibilité pour le joueur de compléter ces objectifs en validant les quêtes liées à chacun d'eux.



### Le système de quêtes :

Ce système est un système linéaire où chaque quête est débloquée une fois la quête précédente complétée. L'objet **Quest** a un état implémenté en variable entière et pouvant prendre les valeurs suivantes :

- 0 : quête bloquée -> état par défaut
- 1 : quête débloquée quand la quête précédente est terminée
- 2 : quête acceptée quand le joueur a interagit avec le donneur de quête
- 3 : quête complétée quand les conditions ont été remplies et que le joueur a interagit avec le PNJ validant de quête.

Les conditions de validation d'une quête peuvent être de deux types :

- QuestType1 = "Aller parler à ..." -> le joueur doit simplement se rendre à un endroit et interagir avec un PNJ en particulier.
- QuestType2 = "Couper X arbres" -> Le joueur doit interagir avec un certain nombre d'objet du même type en utilisant l'outil adéquat. Cela étant fait, la quête est validée (variable **validated** passée à True) mais pas encore complétée. Pour ce faire le joueur doit encore aller parler à un PNJ comme pour les quêtes de type 1.

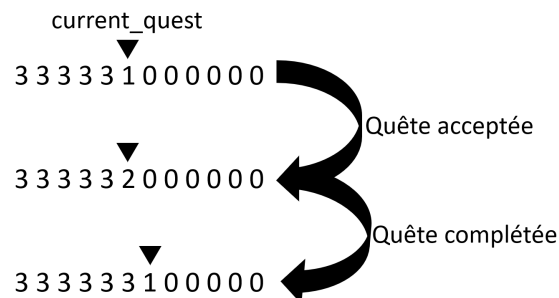
Comme pour les ODD, les quêtes sont stockées sous forme de dictionnaire dans le fichiers **data.py** avec les éléments suivant :

- quest\_type : type 1 ou type 2 -> utilisé pour créer l'objet de la bonne classe
- goal\_nbr : numéro de l'ODD -> une fois toutes les quêtes liées à un ODD complétées, celui-ci est également complété. Si pas possible de lier à un objectif : goal\_nbr = 0
- statement : intitulé de la quête -> affiché dans le journal des quêtes
- dialogue : liste de String
  - dialogue[0] = dialogue du PNJ donnant la quête
  - dialogue[1] = dialogue du PNJ ayant donné la quête si celle-ci est déjà acceptée mais pas encore complétée.
  - dialogue[2] = dialogue du PNJ validant la quête
  - (dialogue[3]) = dialogue d'interaction avec l'objet de quête dans le cas d'une quête de type 2.
- npc1\_name : l'id du PNJ donnant la quête
- npc2\_name : l'id du PNJ validant la quête
- npc2\_type : le type du PNJ validant la quête, utilisé pour afficher son image dans le journal de quêtes
- (obj\_type) : le type de l'objet de quête dans le cas d'une quête de type 2
- (obj\_nbr) : le nombre d'objets avec lesquels le joueur doit interagir dans le cas d'une quête de type 2

### Classe **Script** :

Enfin, pour gérer tous ces éléments de scénario, nous avons la classe **Script** qui va tout d'abord créer tous ces objets **Quest** et **Goal** et ensuite vérifier quand ceux-ci sont complétés et faire avancer le scénario en fonction.

Comme le système de quêtes est linéaire, **Script** utilise une tête de lecture, à savoir **current\_quest**, qui va avancer dès que la quête courante est complétée. Autrement dit, à tout instant l'état de **current\_quest** est toujours égal à 1 ou 2. En effet, une fois la quête courante complétée, la suivante (la prochaine quête courante) passe à l'état 1 et donc son état n'est jamais égal ni à 0 ni à 3, comme le montre la figure ci-dessous.



#### 4.5.5 `game_display.py`

Une fois tous ces éléments créés et gérés, il ne reste plus qu'à les afficher à l'écran. Pour rappel, l'objet **Game** possède plusieurs variables booléennes permettant de savoir quel écran afficher à chaque instant. Ces variables étant modifiées par les inputs de l'utilisateur dans la méthode **game\_events()**. Par exemple, l'utilisateur est en jeu et appuie sur la touche **F** ce qui a pour effet de faire passer la variable **show\_quest** à `True` et ainsi au moment de l'affichage, l'objet **Game** sait qu'il doit afficher le journal des quêtes et appelle donc la méthode **show\_quest\_screen** de son objet **GameDisplay**.

Les différents écrans à afficher sont les suivants :

- l'écran du jeu en lui-même
- écran de pause
- écran des commandes du jeu
- écran de sauvegarde
- écran du journal des quêtes
- écran du livret des objectifs
- écran de médaille une fois qu'un ODD est complété

Tous ces écrans ont chacun les éléments correspondants suivants : une variable booléenne, une gestion des événements et une méthode d'affichage.

Notons que dans la méthode **display()** de **Game**, le programme commence par dessiner le jeu lui-même. En effet, comme certains écrans sont soit transparents (comme l'écran de pause) soit ne remplissent pas toute la fenêtre (comme le journal de quêtes), il est donc nécessaire de dessiner au préalable le jeu en arrière plan.

**Attention :** une fois que tout est dessiné, ne pas oublier de faire **pygame.display.flip()** pour afficher l'écran créé à l'utilisateur.

#### 4.5.6 Boucler sur **Game**

Dans le fichier **main**, une fois que l'objet **Game** est créé et tant que l'utilisateur ne quitte pas la partie, on boucle en appelant les 3 méthodes suivantes :

1. `update()`
2. `display()`
3. `game_events()`

De manière logique, on gère les inputs de l'utilisateur, on met éventuellement à jour le scénario et la position de la map et des sprites dessus puis on ré-affiche à l'utilisateur l'écran mis à jour. Pourquoi alors commencer par **update()** et pas par **game\_events()** ? Tout simplement pour mettre à jour l'écran lors du premier passage dans la boucle avec les paramètres donnés à **Game** avant d'afficher cet écran. Étant donné que, comme dit précédemment, l'écran n'est pas rafraîchi tant qu'il n'y a pas un nouvel input de l'utilisateur donc en mettant **game\_events()** en première il faudrait ré-appuyer sur une touche pour voir l'écran de jeu au lancement d'une partie.

## 5 Annexe



### LES 17 OBJECTIFS DE DÉVELOPPEMENT DURABLE



Les Objectifs de développement durable (ODD) ont été adoptés par l'Organisation des Nations unies.

Ils constituent l'Agenda 2030, qui associe à chaque objectif des cibles à atteindre à l'horizon 2030, en vue d'« éradiquer la pauvreté, protéger la planète et garantir la prospérité pour tous ».

Voici la liste de ces dix-sept ODD.



Éradiquer la pauvreté sous toutes ses formes et partout dans le monde.



Fin de la faim, réaliser la sécurité alimentaire, améliorer la nutrition et promouvoir une agriculture durable.



Assurer une vie saine et promouvoir le bien-être pour tous à tous les âges.



Assurer une éducation de qualité inclusive et équitable et promouvoir des opportunités d'apprentissage pour tous tout au long de la vie.



Réaliser l'égalité du genre et l'autonomisation des femmes et des filles.



Garantir l'accès de tous à l'eau et à l'assainissement et assurer une gestion durable des ressources en eau.



Accélérer l'accès à une énergie abordable, fiable, durable et moderne pour tous.



Promouvoir une croissance économique soutenue, inclusive et durable, le plein emploi productif et un travail décent pour tous.



Construire une infrastructure résiliente, promouvoir une industrialisation inclusive et durable et favoriser l'innovation.



Réduire les inégalités dans et entre les pays.



Rendre les villes et les établissements humains inclusifs, sûrs, résilients et durables.



Assurer des modes de consommation et de production durables.



Prendre des mesures urgentes pour lutter contre le changement climatique et ses impacts.



Conserver et utiliser durablement les océans, les mers et les ressources marines pour le développement durable.



Protéger, restaurer et promouvoir l'utilisation durable des écosystèmes terrestres, la gestion durable des forêts, lutte contre la désertification et stopper et inverser la dégradation des terres et la perte de la biodiversité.



Promouvoir des sociétés pacifiques et inclusives pour le développement durable, permettre un accès à la justice pour tous et bâtir des institutions efficaces, redevables et inclusives à tous les niveaux.



Renforcer les moyens de mise en œuvre et revitaliser le partenariat mondial pour le développement durable.

## 6 Bibliographie

**[1] Les 17 ODD :**

[www.un.org/sustainabledevelopment/fr/objectifs-de-developpement-durable/](http://www.un.org/sustainabledevelopment/fr/objectifs-de-developpement-durable/)

**[2] Pygame sur OpenClassrooms :**

[openclassrooms.com/fr/courses/1399541-interface-graphique-pygame-pour-python](https://openclassrooms.com/fr/courses/1399541-interface-graphique-pygame-pour-python)

**[3] Kidscancode :**

[github.com/kidscancode/pygame\\_tutorials](https://github.com/kidscancode/pygame_tutorials)

[kidscancode.org/lessons/](https://kidscancode.org/lessons/)

[www.youtube.com/playlist?list=PLsk-HSGFjnaH5yghzu7PcOzm9NhsW0Urw](https://www.youtube.com/playlist?list=PLsk-HSGFjnaH5yghzu7PcOzm9NhsW0Urw)

**[4] Musique :**

[opengameart.org/content/town-theme-rpg](https://opengameart.org/content/town-theme-rpg)

**[5] Tiled :**

[www.mapeditor.org/](http://www.mapeditor.org/)

**[6] Assets :**

[www.kenney.nl/assets](http://www.kenney.nl/assets)

[itch.io/game-assets/free](https://itch.io/game-assets/free)

[opengameart.org/](https://opengameart.org/)

CapytaineDurable version 1.0.

