Tutoriel utilisant la programmation objet

Programmer un shooter avec Pyxel : 1/6

Doc officielle de Pyxel : https://kitao.github.io/pyxel/web/api-reference/

Le but de ce tutoriel est de programmer un jeu de type shooter à l'aide de la bibliothèque `pyxel` en utilisant la programmation objet. Les méthodes vues ici sont adaptables à d'autres types de jeu

Le principe global est d'utiliser une classe `Jeu` pour faire les initialisations et lancer le jeu ainsi que différentes classes pour chaque type d'objet intervenant dans le jeu (Vaisseau, Ennemis, Tirs, Explosions, …). Pour minimiser la quantité de code à écrire et simplifier leur mise en jeu, ces objets seront basés sur une classe `Objet` dont ils hériterons et qui contiendra les méthodes communes à tous les types d'objets.

1. Création des classes de base

La première classe à créer est la classe Jeu. En dehors de son constructeur, elle contient deux méthodes d'instance, `update(self)` et `draw(self)` ainsi que 3 méthodes de classe (que nous utiliserons plus loin).

```
import pyxel
LARGEUR_ECRAN, HAUTEUR_ECRAN = 128, 128

class Jeu:
    def __init__(self):
        Jeu.jeu = self            # Référence vers l'instance de jeu
        self.liste_objets = []    # Liste des objets du jeu
        pyxel.init(LARGEUR_ECRAN, HAUTEUR_ECRAN, "Shooter")
        pyxel.run(self.update, self.draw)

    def update(self):
        for objet in self.liste_objets:
            objet.update()

    def draw(self):
        pyxel.cls(0)
        for objet in self.liste_objets:
            objet.draw()

    def ajoute_objet(objet):
        Jeu.jeu.liste_objets.append(objet)

    def retire_objet(objet):
        Jeu.jeu.liste_objets.remove(objet)

    def get_liste_objets():
        return Jeu.jeu.liste_objets

Jeu()   # Ceci doit être la dernière ligne de code !
```

Il n'y aura qu'une seule instance de la classe `Jeu` de créée. Elle a en charge les initialisations ainsi que la liste des objets du jeu dont elle appelle les méthodes `update` et `draw`.

Remarque : `ajoute_objet` et `retire_objet` et `get_liste_objets` sont des **méthodes de classe**. Elles vont permettre d'ajouter ou de supprimer des objets à la liste d'objets à partir de n'importe où dans le code en appelant la méthode de classe par `Jeu.ajoute_objet(`*`objet`*`)` ou `Jeu.retire_objet(`*`objet`*`)`.

Par la suite nous ferons évoluer un peu la classe Jeu et ajoutant les initialisations nécessaires ou bien des méthodes liées à la liste d'objet.

La classe `Objet` va contenir les objets du jeu. Celle-ci contient deux méthodes : `update(self)` et `draw(self)` qui seront appelées en boucle par la classe `Jeu`. C'est une classe abstraite, ce qui veut dire qu'on ne créera aucun objet de cette classe, mais qu'elle va servir de base pour les autres classes du jeu :

```
class Objet:
    def update(self):
        pass

    def draw(self):
        pass
```

On remarque que pour l'instant elle ne contient pas grand-chose d'utile et ne possède même pas de constructeur, mais nous la compléterons au fur et à mesure pour rajouter les fonctionnalités communes à tous les objets du jeu (comme la suppression de l'objet, la détection de collisions, …)

2. Création du vaisseau

On va maintenant créer une classe `Vaisseau` qui hérite de la classe `Objet` et représentera le vaisseau spatial du joueur :

```
class Vaisseau(Objet):
    def __init__(self, x:int, y:int):
        self.x, self.y = x, y
        self.largeur, self.hauteur = 16, 12

    def update(self):
        # On modifie la position du vaisseau en fonction des touches appuyées
        pass

    def draw(self):
        # On dessine un rectangle aux coordonnées x, y de taille 16x12
        pass
```

⇒ Compléter la méthode `draw` pour qu'elle dessine un rectangle de la taille du vaisseau à ses coordonnées. Dans le constructeur du jeu insérer les instructions pour ajouter un vaisseau aux coordonnées 60,100 à la liste des objets et valider que celui-ci s'affiche bien. Compléter alors la méthode `update` pour pouvoir déplacer le rectangle avec les touches du clavier.

⇒ Comment faire pour que le vaisseau ne sorte pas de l'écran ? Ecrire les lignes de code qui permettent de faire en sorte que le vaisseau ne puisse pas sortir de l'écran.

Remarque : On peut également déplacer le vaisseau à la manette en testant les « touches » `GAMEPAD1_BUTTON_DPAD_xxx` où `xxx` est parmi `LEFT`, `RIGHT`, `UP` et `DOWN`.

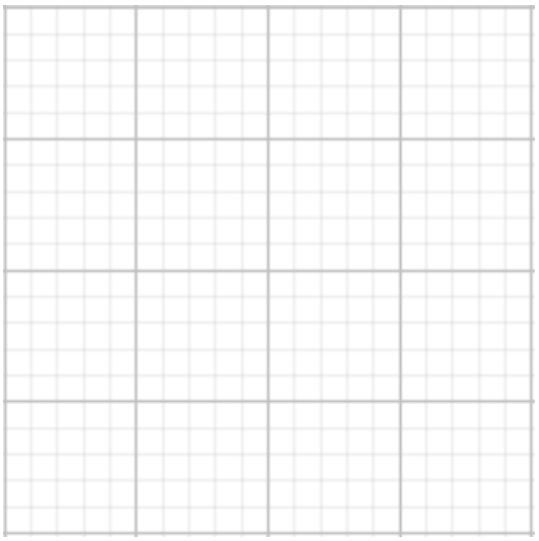
Programmer un shooter avec Pyxel : 2/6

3. Ajouter des tirs

a) Créer les objets tirs, et dessiner les tirs

Dans cette partie, nous allons représenter chaque tir par un rectangle jaune, de largeur 1 pixel et de hauteur 4 pixels. A chaque fois que l'utilisateur appuiera puis relâchera la touche Espace, cela créera un nouveau tir qui sera ajouté à la liste des objets.
Concrètement, le tir sera représenté par un objet de la classe `Tir`, elle-même héritière de la classe `Objet`.
Un objet `Tir` possèdera deux attributs `x` et `y` correspondant aux coordonnées de son coin supérieur droit et deux attributs `largeur` et `hauteur` valant respectivement 1 et 4.
`Tir(18, 5)` permet de définir un tir aux coordonnées x = 18 et y = 5.
Exemple : dessiner sur la fenêtre ci-contre (20x20 pixels) les tirs représentés en mémoire par :

```
Tir(2,16)
Tir(6,12)
Tir(10,8)
Tir(14,4)
```



Pour gérer le déplacement des tirs (qui montent vers le haut de l'écran) et les dessiner à l'ordinateur, on redéfinit les méthodes `update` et `draw` de la classe `Tir`.

```
class Tir(Objet):
    def __init__(self, x:int, y:int):
        self.x, self.y = x, y
        self.largeur, self.hauteur = 1, 4

    def update(self):
        pass

    def draw(self):
        pass
```

⇒ Compléter la méthode `draw` pour qu'elle dessine en jaune le rectangle du tir. Dans le constructeur du jeu ajouter les instructions pour ajouter à la liste des objets (temporairement, uniquement pour les tests) des tirs aux coordonnées définies plus haut et valider que ceux-ci s'affichent bien.

⇒ Supprimer les instructions de tests précédentes (qui ajoutaient des objets `Tir` à la liste des objets).

⇒ Modifier la méthode `update` de la classe `Vaisseau` pour qu'à chaque appuie sur la touche espace un nouvel objet `Tir` soit créée aux coordonnées du vaisseau (plus précisément juste au-dessus et au milieu du vaisseau).

⇒ On voudrait ajouter l'objet `Tir` à la liste des objets. Dans quelle classe cette liste est-elle gérée ? Comment peut-on y accéder à partir de la classe `Vaisseau` (voir début du tutoriel) ?

⇒ Rajouter l'appel de méthode nécessaire pour ajouter l'objet `Tir` et vérifier qu'il s'affiche correctement.

```
class Jeu:
    def __init__(self):
        self.liste_objets = []     # Liste des objets du jeu
        Jeu.ajoute_objet(Vaisseau(60, 100))
[…]
class Vaisseau(Objet):
    […]
    def update(self):
        […]
        # Si on appuie sur <espace> cela créée un tir juste au-dessus du vaisseau
        if pyxel.btn(pyxel.KEY_SPACE):
            Jeu.ajoute_objet(Tir(self.x + self.largeur//2, self.y - 4))
```

b) Faire bouger les tirs

On va maintenant faire bouger les tirs pour qu'ils montent au fur et à mesure vers le haut de l'écran

⇒ Quel attribut doit-on modifier pour chaque objet tir ?

⇒ Dans quelle méthode doit-on mettre le code pour modifier la position des tirs ?

⇒ Insérer le code permettant de faire monter les tirs de 1 pixel vers le haut à chaque frame. Vérifier que cela fonctionne correctement. (On peut éventuellement créer un attribut `vitesse` pour les tirs et faire monter les tirs de `self.vitesse` à chaque fois ce qui permet de la régler comme on le souhaite)

⇒ Que se passe-t-il pour les tirs une fois qu'ils sortent de l'écran ? Est-ce un problème ?

⇒ On souhaite supprimer les tirs une fois qu'ils sont sortis de l'écran. Comment peut-on faire cela ?

Afin de pouvoir supprimer les objets de la liste, on va ajouter une méthode `suppression(self)` à la classe `Objet` : ainsi tous les objets héritant de cette classe pourront utiliser cette méthode.

⇒ Ajouter cette méthode à la classe `Objet` (cette méthode appelle simplement la méthode `retire_objet` de la classe Jeu sur l'objet lui-même).

⇒ Ajouter un appel à la méthode `suppression` de l'objet `Tir` lorsque celui-ci atteint le haut de l'écran.

On veut vérifier que la liste des objets se vide bien. Pour cela on va afficher à l'écran des informations de débug (destinées uniquement à nous, développeurs) uniquement lorsqu'on appuie sur la touche <Control> de droite.
On utilise pour ce faire le code suivant :

```
if pyxel.btn(pyxel.KEY_RCTRL):
    pyxel.text(0,120,f"taille liste objets : {len(self.liste_objets)}",13)
```

⇒ A quel endroit du code faut-il insérer les lignes précédentes ? Faire la modification et vérifier ainsi que la liste des objets se vide bien lorsque les tirs sortent de l'écran.

Il reste un problème : lorsqu'on appuie sur espace il se peut que cela crée 2 ou 3 tirs d'affilé. Pour régler ce problème, on peut soit créé un attribut « `cooldown` » dans la classe Vaisseau qu'on initialise à une certaine valeur et qu'on décompte jusqu'à 0 pour autoriser la création d'un nouveau tir, soit on utilise l'instruction `pyxel.btnp` au lieu de `pyxel.btn` pour détecter l'appuie sur espace.

⇒ En utilisant la documentation de pyxel, expliquer la différence entre les fonctions `btn` et `btnp`. Faire la modification dans le code de la classe `Vaisseau` et valider que le nouveau comportement est celui attendu.

Programmer un shooter avec Pyxel : 3/6

Dans cette partie, nous allons créer des ennemis qui apparaîtront à intervalle de temps régulier à une position aléatoire en haut de l'écran puis descendront jusqu'en bas. Pour l'instant on ne gère pas les collisions et les tirs comme le vaisseau « traversent » les ennemis sans rien déclencher.

4. Ajouter des ennemis

a) Créer les objets ennemis, et les dessiner

Un ennemi sera représenté par un objet de la classe `Ennemi`, elle-même héritière de la classe `Objet`.
Un objet `Ennemi` possèdera deux attributs `x` et `y` correspondant aux coordonnées de son coin supérieur droit et deux attributs `largeur` et `hauteur` valant respectivement 8 et 8.

```
class Ennemi(Objet):
    def __init__(self):
        self.x, self.y = 0, -7
        self.largeur, self.hauteur = 8, 8

    def update(self):
        pass

    def draw(self):
        pass
```

⇒ Quelle bibliothèque permet d'obtenir des nombres aléatoires ? Quelle fonction utilise-t-on pour avoir un nombre entier aléatoire entre min et max ?

NB : `pyxel` propose en interne la fonction `pyxel.rndi` qui fait exactement la même chose que `random.randint`. On peut l'utiliser pour éviter d'avoir à importer la bibliothèque `random`.

⇒ Modifier le constructeur pour que la coordonnée `x` de l'ennemi soit un nombre aléatoire entre les deux bords de l'écran.

⇒ Compléter la méthode `draw` pour qu'elle dessine en rouge clair un rectangle à la taille de l'ennemi. Dans le constructeur du jeu ajouter les instructions pour ajouter à la liste des objets (temporairement, uniquement pour les tests) trois ennemis et valider que ceux-ci s'affichent bien.

b) Faire apparaître les ennemis

Pour faire apparaître un ennemi nouveau de manière régulière, on va utiliser la variable `pyxel.frame_count` qui compte le nombre de frame écoulées depuis le lancement du programme.

⇒ Supprimer les instructions de tests précédentes (qui ajoutaient des objets `Ennemi` à la liste des objets).

⇒ Modifier la ligne affichant les informations de débug pour qu'elle affiche en plus la valeur de `pyxel.frame_count`. Lancer le programme et maintenir la touche control de droite appuyée pour voir l'évolution du frame count.

⇒ Quelle instruction python permet de vérifier si un nombre entier n est divisible par p ?

⇒ Modifier la méthode `update` de la classe `Jeu` pour qu'un nouvel objet `Ennemi` soit créée et ajouté à la liste des objets toutes les 35 frames (à chaque fois que `pyxel.frame_count` est divisible par 35). Valider le fonctionnement.

⇒ Créer une variable globale `ATTENTE_ENNEMI` valant 35 et modifier le code précédent pour que les ennemis apparaissent toutes les `ATTENTE_ENNEMI` frames.

c) Faire bouger les ennemis

⇒ Quelle méthode doit-on modifier pour faire descendre les ennemis à chaque frame ?

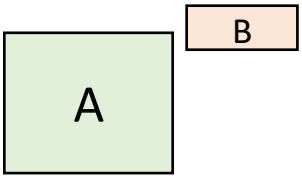
⇒ Faire les modifications nécessaires et tester le fonctionnement.

⇒ On souhaite supprimer les ennemis une fois qu'ils sont sortis de l'écran. Comment peut-on faire cela ?

⇒ Effectuer les modifications de code nécessaire et vérifier à l'aide des informations de débug que la liste des objets se vide bien au fur et à mesure.

Programmer un shooter avec Pyxel : 4/6

Dans cette partie, nous allons ajouter la détection de collision qui permettra de faire perdre le joueur si son vaisseau touche un ennemi et de faire disparaitre les ennemis touchés par les tirs du vaisseau.

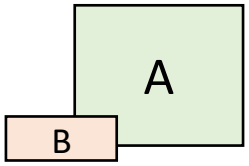


Les hitbox ne se superposent pas : pas de collision

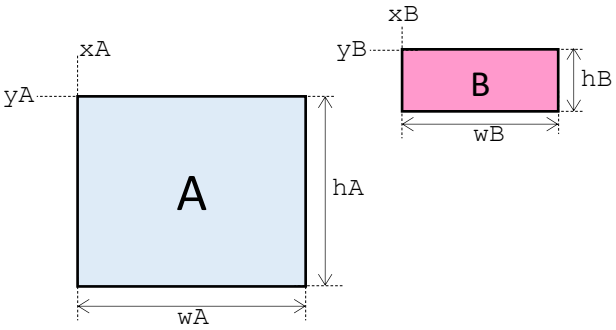
5. Ajouter les collisions

a) Principe de la détection de collisions

La détection de collision est en général faite à l'aide de hitbox. Ces hitbox sont des rectangles qui représentent les surfaces des objets dont on veut déterminer la collision. Chaque objet a une hitbox [1](un rectangle). Il y a collision entre deux objets si leurs hitbox se superposent au moins partiellement.

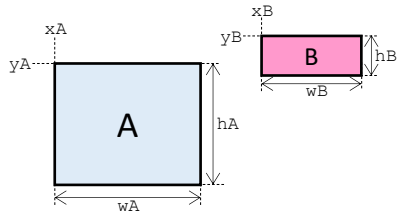
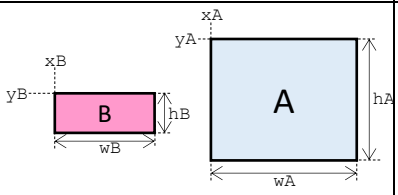
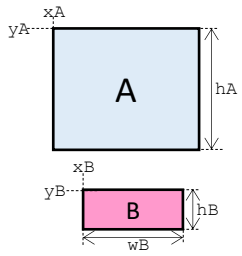
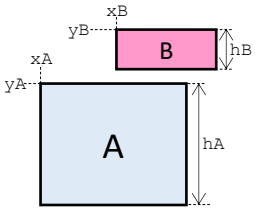


Les hitbox se superposent : il y a collision



Les hitbox sont définies par les coordonnées de leur point en haut à gauche ainsi que leur largeur et leur hauteur.

⇒ Remplir le tableau suivant en notant les conditions sur xA, yA, wA, hA et xB, yB, wB, hB pour qu'il **n'y ait pas** de collision dans les cas suivants :

	Cas	Condition
	Trop à droite	
	Trop à gauche	
	Trop en bas	
	Trop en haut	

[1] En réalité un objet peut avoir plusieurs hitbox s'il a une forme particulière ou si on veut différencier la collision avec différentes parties de l'objet (par exemple collision avec l'arme ou avec le bas différente de celle avec le reste du corps).

Finalement, il n'y a pas de collision entre les deux hitbox (A et B) si on est dans n'importe laquelle des 4 situations précédentes. Dans le cas contraire (si aucune de ces conditions n'est vérifiée), c'est qu'il y a superposition (au moins partielle) de A et B. Ainsi il est plus simple de tester *l'absence* de collision et si il n'y a pas d'absence de collision … c'est qu'il y a collision !

⇒ Dans un fichier séparé, écrire une fonction python `collision(xA, yA, wA, hA, xB, yB, wB, hB)` qui prends en argument les coordonnées des deux hitbox et renvoie `True` si les hitbox se superposent au moins partiellement et `False` sinon. Tester la fonction avec les cas suivants :

```
assert collision(5,5,10,5, 25,12,8,8) == False   # trop à droite
assert collision(30,5,10,5, 8,5,8,8) == False    # trop à gauche
assert collision(5,5,10,5, 7,20,8,8) == False    # trop en bas
assert collision(5,35,18,10, 7,2,8,8) == False   # trop en haut
assert collision(5,5,30,40, 10,10,7,7) == True   # à l'intérieur
assert collision(5,5,10,5, 16,8,7,7) == False    # côte à côte
assert collision(5,5,10,5, 12,2,8,8) == True     # superposé en haut à droite
assert collision(5,5,10,15, 3,7,8,5) == True     # superposé à gauche
```

b) Ajout de la capacité de détection de collision à la classe Objet

De manière à ce que tous les objets puissent détecter les collisions, on va ajouter deux méthodes à la classe `Objet` de manière à ce que les classes filles (`Vaisseau`, `Tir` et `Ennemi`) en héritent et soient en mesure de les utiliser.

⇒ Dans la classe Objet, ajouter une méthode `get_hitbox(self)` qui renvoie un tuple contenant les coordonnées `(x, y, w, h)` de la hitbox de l'objet à partir de ses attributs `x`, `y`, `largeur` et `hauteur`.

On va maintenant ajouter une méthode `est_en_collision_avec(self, objetB)` que l'on pourra appeler à partir d'un objet pour savoir s'il est en collision avec un autre.

Par exemple :

```
v = Vaisseau(60, 100)
e = Ennemi()
v.est_en_collision_avec(e)   # Vaut True si l'ennemi touche le vaisseau
e.est_en_collision_avec(v)   # doit renvoyer la même chose
```

⇒ Dans la classe `Objet`, ajouter une méthode `est_en_collision_avec(self, objetB)` qui renvoie `True` si l'objet courant est en collision avec l'objet B et `False` sinon.

```
def est_en_collision_avec(self, objetB):
    """Renvoie True si les deux hitbox sont en collision, False sinon."""
    xA, yA, wA, hA = self.get_hitbox()     # coordonnées de la 1ère hitbox
    xB, yB, wB, hB = objetB.get_hitbox()   # coordonnées de la deuxième hitbox
    […]
```

Maintenant tous les objets peuvent disposer de ces deux méthodes pour détecter leurs collisions avec d'autres objets.

c) Collisions du vaisseau avec les ennemis

On va maintenant faire en sorte que le vaisseau teste ses collisions avec tous les ennemis. S'il touche un ennemi, on supprime le vaisseau et l'ennemi concerné.
Pour ce faire, on va d'abord ajouter une méthode `get_liste_ennemis()` à la classe Jeu :

```
def get_liste_ennemis():
    """Renvoie la liste de tous les ennemis en jeu."""
    liste_ennemis = []
    for objet in Jeu.get_liste_objets():
        if isinstance(objet, Ennemi):
            liste_ennemis.append(objet)
    return liste_ennemis
```

L'instruction `isinstance(a,b)` renvoie `True` si l'objet `a` est du type de la classe `b` (ou un sous-type de b).

Par exemple `isinstance(12,int)` renverra `True` (car `12` est bien un entier de la classe `int`), tandis que `isinstance(3.4,str)` renverra `False` (car `3.4` est de la classe `float` et non de la classe `str`).

⇒ Dans la classe `Jeu`, ajouter la méthode `get_liste_ennemis`.

⇒ A quel endroit doit-on mettre le code qui détecte la collision entre le vaisseau et chaque ennemi ?

⇒ Ecrire alors au bon endroit le bout de code qui parcours la liste des ennemis, teste la collision avec le vaisseau et si elle a lieu, supprime le vaisseau et l'ennemi de la liste des objets du jeu. Tester ensuite votre programme pour voir si la collision avec les ennemis fait bien disparaître le vaisseau.

d) Pyrotechnie

Plutôt que de simplement disparaître, on voudrait qu'à la place du vaisseau et de l'ennemi apparaissent des explosions : On les simulera avec des ronds de plus en plus grands.
Pour implémenter cela, on va créer une nouvelle classe `Explosion` qui hérite de la classe `Objet` et créer 2 variables globales (au début du code) :

```
DUREE_EXPLOSION = 6
COULEURS_EXPLOSION = [8, 9, 10]
[…]
class Explosion(Objet):
    def __init__(self, x, y, rayon):
        self.x, self.y = x, y
        self.rayon = rayon
        self.duree_de_vie = 0

    def update(self):
        self.duree_de_vie += 1
        if self.duree_de_vie > DUREE_EXPLOSION:
            self.suppression()
            return
        self.rayon += 1

    def draw(self):
        pyxel.circb(self.x, self.y, self.rayon,
                    COULEURS_EXPLOSION[self.duree_de_vie%len(COULEURS_EXPLOSION)])
```

⇒ Expliquer comment fonctionne la classe `Explosion`, puis ajouter celle-ci au code source.

⇒ Ajouter une méthode `destruction(self)` à la classe `Objet` qui crée un objet `Explosion` aux coordonnées de l'objet et supprime l'objet de la liste (en appelant la méthode `suppression`).

⇒ Modifier le code de collision de la classe `Vaisseau` pour que lorsqu'une collision est détectée, la méthode `destruction` des deux objets (l'ennemi et le vaisseau) soit appelée à la place de la méthode `suppression`. Vérifier que cela fonctionne.

e) Collisions des tirs avec les ennemis

On veut maintenant que les tirs du vaisseau fassent exploser les ennemis.

⇒ Comment peut-on procéder pour atteindre l'objectif ? Quelle classe doit-on modifier ?

⇒ Effectuer les modifications nécessaires et vérifier le fonctionnement.

Programmer un shooter avec Pyxel : 5/6

Il est temps de gérer le score et la fin de partie. On souhaite afficher sur l'écran le score actuel et le nombre de vies. Le score commence à 0 et le nombre de vies à 3.
Chaque ennemi détruit rapporte 5 points. Chaque fois que le vaisseau est détruit (touche un ennemi), il perd une vie. Si le nombre de vies arrive à 0, on affiche un message de Game Over avant de recommencer.

6. Gestion du score

a) Création et affichage

Il y a plusieurs façons de gérer le score. On pourrait par exemple créer une classe `Score` héritant de `Objet`, possédant des méthodes pour modifier le score et dont la méthode draw afficherait le score à l'écran.
Ici on va faire simple et créer une variable de classe `score` dans la classe `Jeu`. Comme c'est une variable de classe, elle sera facilement accessible de n'importe quel endroit du programme.

⇒ Ajouter la création d'une variable de classe `score` (initialisée à 0) au constructeur de la classe `Jeu`.

⇒ Ajouter à la fin de la méthode `draw` de la classe `Jeu` la ligne permettant d'afficher le score en haut à gauche de l'écran. Quel est l'intérêt de mettre cette ligne à la toute fin de la méthode `draw` plutôt qu'au début ?

b) Modification du score

⇒ A quel endroit doit-on ajouter du code pour que chaque ennemi détruit rapporte 5 points ? Ecrire le code et tester son fonctionnement.

7. Gestion des vies

a) Création, affichage et modification

⇒ En s'inspirant de la partie précédente, créer une variable de classe `vies` dans la classe `Jeu`. Afficher le nombre de vies dans le coin en haut à droite et faire en sorte qu'il soit décrémenté lorsque le vaisseau est détruit.

Lorsque le vaisseau est détruit, on ne peut pas le remettre en jeu tout de suite car il pourrait y avoir un ennemi à cet endroit. On va attendre un peu avant de supprimer tous les objets et de replacer le vaisseau au centre. Pendant l'attente un message indiquera au joueur qu'il va bientôt reprendre.

On souhaite en fait ajouter la possibilité pour notre programme d'être dans un état qui n'est pas celui du jeu normal (ici un état d'attente, plus tard ce sera le game over). Pour cela on va modifier la classe `Jeu` afin d'ajouter la gestion des états en créant une variable de classe `etat` qui vaudra `None` quand le jeu est en cours et référencera une classe permettant de gérer l'état correspondant quand le jeu est en attente de lancement ou en game over.

⇒ Dans la classe `Jeu`, ajouter une ligne au constructeur pour créer une variable de classe `etat` initialisée à `None`. A la fin de la méthode update ajouter un test : si `Jeu.etat` est à `None` on ne fait rien, mais si il est différent, on exécute la méthode `update` de `Jeu.etat`. Même chose à la fin de la méthode `draw` pour exécuter `Jeu.etat.draw()` si `Jeu.etat` n'est pas `None`.

On va créer une nouvelle classe `Lancement` qui va nous permettre de gérer le temps d'attente, la reprise et d'afficher également un petit texte pendant l'attente :

```
class Lancement():
    def __init__(self):
        self.attente = 120

    def update(self):
        self.attente -= 1
        if self.attente < 0:
            Jeu.vide_liste_objets()
            Jeu.ajoute_objet(Vaisseau(60, 100))
            Jeu.etat = None

    def draw(self):
        pyxel.text(40, 60, "Prepare-toi !", 7)
```

⇒ Ajouter à la classe Jeu une méthode de classe `vide_liste_objets` qui vide le contenu de la liste des objets.

⇒ Ajouter les lignes de code permettant au moment où le vaisseau touche un ennemi de décrémenter le nombre de vies puis de créer un objet `Lancement` et de mettre sa référence dans `Jeu.etat`. Pour l'instant on ne gère pas le cas où celui-ci descend en dessous de 1.

b) Game over

Tout jeu a une fin et il faut aussi la gérer. Ici on se contentera d'afficher un message de game over. Pour cela on va créer une nouvelle classe `GameOver` dont on référencera une instance avec `Jeu.etat` :

```
class GameOver():
    def __init__(self):
        self.depart = False

    def update(self):
        if pyxel.btn(pyxel.KEY_RETURN) or pyxel.btn(pyxel.GAMEPAD1_BUTTON_START):
            self.depart = True
        if self.depart:
            Jeu.vide_liste_objets()
            Jeu.ajoute_objet(Vaisseau(60, 100))
            Jeu.vies = 3
            Jeu.score = 0
            Jeu.etat = None

    def draw(self):
        pyxel.text(49, 50, "GAME OVER", 7)
        pyxel.text(30, 80, "<Enter> ou <Start>",7)
        pyxel.text(43,90,"pour rejouer", 7)
```

⇒ Expliquer le fonctionnement de cette classe puis modifier le code de la classe `Vaisseau` pour que lorsque le nombre de vies tombe à zéro, on change l'état pour pointer sur un nouvel objet `GameOver`. Tester le fonctionnement.

⇒ On peut éventuellement faire démarrer le jeu sur cet écran de game over. Il suffit pour cela d'initialiser `Jeu.etat` à un nouvel objet `GameOver`.

Programmer un shooter avec Pyxel : 6/6

8. Ajout des graphismes

Dernière étape, habiller notre jeu. Pour l'instant on a que des rectangles de couleur, mais on aimerait avoir de vrais graphismes.
Pour cela on va utiliser un fichier de ressource pyxel : 1.pyxres disponible sur le site de la nuit du code (à https://www.nuitducode.net/univers_python/2024/1.pyxres).

Ce fichier contient les graphismes, tilemaps, sons et musiques pour le jeu. Ici le fichier sélectionné ne contient que des graphismes.
On peut modifier ou simplement visualiser le contenu du fichier ressource en utilisant la commande :
`pyxel edit 1.pyxres` à partir de l'invite de commande en étant dans le répertoire du fichier pyxres

Pour charger en mémoire le contenu du fichier, il faut utiliser la commande python `pyxel.load("1.pyxres")`

Attention : Cette instruction doit obligatoirement se trouver **après** `pyxel.init` et **avant** `pyxel.run`.

Pour représenter un des éléments de graphisme, on utilise ensuite :
`pyxel.blt(x, y, img, u, v, w, h, [colkey], [rotate], [scale])`
`x`, `y` sont les coordonnées du coin en haut à gauche à l'écran où on veut afficher l'image
`img` est le numéro de la banque d'image (ici ce sera 0, il peut y en avoir 3)
`u` et `v` sont les coordonnées x et y à l'intérieur de la banque d'image (par exemple 0,10 pour l'image du vaisseau)
`w` et `h` sont la largeur et la hauteur de la partie de l'image à recopier (exemple 16,11 pour l'image du vaisseau)
`colkey` est la couleur transparente (ici 5) : les pixels de cette couleur dans la banque d'image ne seront pas transférés sur l'écran (et sembleront donc transparents)
`rotate` et `scale` sont optionnels et permettent de faire des rotations et mises à l'échelle (non utilisés ici)

- ⇒ Ajouter l'instruction permettant de charger le fichier ressource dans le constructeur de la classe `Jeu`.
- ⇒ Dans la méthode `draw` de la classe `Vaisseau`, remplacer l'instruction `pyxel.rect …` par celle permettant d'afficher l'image du vaisseau dans la banque d'image. Tester le fonctionnement.
- ⇒ Dans la méthode `draw` de la classe `Ennemi`, remplacer l'instruction `pyxel.rect …` par :
  `pyxel.blt(self.x, self.y, 0, self.alien[0], self.alien[1], 8, 8, 5)`
  puis dans le constructeur de la classe `Ennemi`, rajouter les lignes :
  `liste_aliens = [(4,43),(20,43),(36,43),(4,59),(20,59),(36,59),(52,59)]`
  `self.alien = liste_aliens[random.randint(0, len(liste_aliens)-1)]`
- ⇒ Que vont faire les lignes ajoutées ? Tester et valider le fonctionnement.

Ce petit jeu est fonctionnel et terminé. Bien sûr on peut encore largement l'améliorer ou le configurer différemment vitesse du vaisseau et des ennemis, nombre de vies, sons (avec l'instruction `pyxel.play` (les sons 0 à 4 sont définis dans le fichier pyxres)), etc …

Petite proposition d'amélioration ci-dessous : l'ajout d'un fond étoilé qui défile avec la classe `FondEtoile`.

```
NB_ETOILES_FOND = 20
[…]
class Jeu:
    def __init__(self):
        self.fond = FondEtoile()
        […]

    def update(self):
        self.fond.update()
        […]

    def draw(self):
        pyxel.cls(0)
        self.fond.draw()
        […]
```

```
class FondEtoile():
    def __init__(self):
        self.nombre_etoiles = NB_ETOILES_FOND
        self.liste_etoiles = [{}]*self.nombre_etoiles
        self.couleurs_etoiles = [13,13,13,13,7,7,7,7,7,7,7,10]
        self.vitesses_etoiles = [0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1, 1, 2, 3]

        # Création des étoiles
        for i in range(self.nombre_etoiles):
            etoile = {'x': pyxel.rndi(0,LARGEUR_ECRAN-1),
                      'y': pyxel.rndi(0,HAUTEUR_ECRAN-1),
                      'coul': self.couleurs_etoiles[pyxel.rndi(0, len(self.couleurs_etoiles)-1)],
                      'v': self.vitesses_etoiles[pyxel.rndi(0, len(self.vitesses_etoiles)-1)]}
            self.liste_etoiles[i] = etoile

    def update(self):
        for i in range(self.nombre_etoiles):
            self.liste_etoiles[i]['y'] += self.liste_etoiles[i]['v']
            if self.liste_etoiles[i]['y'] >= HAUTEUR_ECRAN:
                self.liste_etoiles[i] = {'x': pyxel.rndi(0,LARGEUR_ECRAN-1),
                      'y': 0,
                      'coul': self.couleurs_etoiles[pyxel.rndi(0, len(self.couleurs_etoiles)-1)],
                      'v': self.vitesses_etoiles[pyxel.rndi(0, len(self.vitesses_etoiles)-1)]}

    def draw(self):
        for i in range(self.nombre_etoiles):
            pyxel.pset(self.liste_etoiles[i]['x'], self.liste_etoiles[i]['y'],
                       self.liste_etoiles[i]['coul'])
```