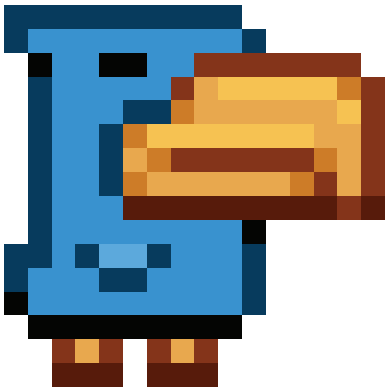
Flappy Bird



SOMMAIRE

I. Introduction 1

II. Principe du jeu 1

III. Réalisation du jeu 1

A. Répartition des tâches 1

B. Structure globale 2

C. Explication des fonctions codées 2

D. Explication de la gestion des obstacles et problèmes rencontrés 4

1. Gestion de l'image des obstacles 4
2. Initialisation des obstacles 4
3. Déplacement des obstacles 7

IV. Conclusion 9

V. Annexes 10

A. classique.py 10

B. fonctions_classique.py 12

I. Introduction

Cette année scolaire, en Terminale S, option Informatique et Sciences du Numérique (ISN), nous avons eu l'occasion d'entreprendre un projet en groupe. En effet, notre groupe, composé de Ludovic FISCHER, Loïc MOSSLER et Antoine REGIN, a choisi de faire un jeu du type « Flappy Bird ». Nous avons pour cela codé en Python et en utilisant la bibliothèque PyGame. Nous nous sommes répartis les tâches suivantes :

- La gestion du personnage
- La gestion des obstacles
- La gestion des collisions
- La gestion du score
- La gestion du « Gameover »

Nous avons divisé le programme en deux fichiers : « classique.py » et « fonctions_classique.py ». Pour que le jeu fonctionne il faut donc ces deux fichiers, les images et les sons, le tout dans un même dossier. De plus Python et PyGame doivent être installés sur l'ordinateur.

II. Principe du jeu

Notre jeu est constitué d'un personnage qui est soumis à la gravité et qui se déplace uniquement verticalement, et d'obstacles qui défilent horizontalement. Le personnage réalise un saut lorsque l'utilisateur appuie sur la touche « ESPACE ».

L'utilisateur améliore alors son score à chaque fois que le personnage passe un obstacle, le but étant donc d'aller le plus loin possible pour améliorer son score.

Lorsque le personnage entre en collision avec les obstacles ou le sol, la partie et terminée. L'image « Game over » apparait alors.

Pour recommencer une partie, l'utilisateur appuie sur « ↑ »

III. Réalisation du jeu

A. Répartition des tâches

Au début du projet, nous avons tous les trois codé ensemble. Nous nous sommes vite aperçu que c'était une perte de temps et que nous n'étions pas performants de cette manière. Nous nous sommes donc réparti les tâches :

- Ludovic s'est occupé de la gestion des obstacles ;
- Loïc s'est occupé de la gestion du personnage ;
- Antoine s'est occupé de la gestion des collisions (sol et obstacles), du score, des sons et du « Game Over ».

B. Structure globale

Après les imports, les chargements des images et différentes initialisations, on entre dans une boucle while qui est parcourue toutes les 60 ms. On parcourt les évènements de la file des évènements :

- Si on clique sur la croix, la fenêtre se ferme.
- Si on appuie sur une touche et que cette touche est la touche ESPACE et qu'on n'est pas en Game Over, on passe en phase de saut. Sinon, si on appuie sur la touche ↑ et que le jeu est en Game Over, on réinitialise le jeu avec la fonction initialisation_jeu.

Ensuite il y a les actions qui se font une fois par image (toutes les 60ms) :

- On teste si la gravité est activée, si c'est le cas, le personnage descend de 3 pixels. Sinon (on est en phase de saut) le personnage monte de 7pixels, pendant dix passages dans la boucle while et après la gravité revient à 1.
- On teste si on est en Game Over, si oui, une image s'affiche. Sinon, on déplace les obstacles, on teste les collisions, on incrémente le score et on rafraichit l'écran.

C. Explication des fonctions codées

Nous avons commencé par coder la partie principale : Tout d'abord nous avons commencé par charger les modules que nous allons utiliser et initialiser Pygame. Afin d'éviter les erreurs liées aux caractères spéciaux on utilise « #-*-coding:utf8-*- » :

```
#-*-coding:utf8-*-
from fonctions_classique import *
import pygame
from random import randint
from pygame.locals import *
import time
import traceback
pygame.init()
```

Ensuite, nous avons mis au programme principal un «try» qui se termine par «finally» ce qui permet de quitter le jeu sans le faire planter. Nous avons défini la fenêtre de couleur (116, 197, 205) en RGB et de taille 800px (largeur)*600px (hauteur) :

```
background = (116, 197, 205)
fenetre = pygame.display.set_mode((800,600))
```

On charge alors les images et les sons : le personnage, les obstacles, l'image du « Game Over » et les sons de saut, collision, Game Over et score. La fonction clock.tick(60) permet de générer une image toute les 60ms. Lorsque l'utilisateur demande à quitter le jeu, la variable « continuer » passe à 0. Ainsi, le programme s'arrête car il sort de la boucle « while continuer=1 ».

De plus nous avons créé le fichier « fonctions_classique », contenant toutes les fonctions nécessaires dans le programme, dans le but d'alléger le programme principal.

Dans ce fichier on trouve notamment la fonction « initialisation_jeu » qui s'occupe de l'initialisation du programme. On y initialise les variables et on crée les rectangles autour des images :

```
def initialisation_jeu(background, fenetre, perso, obs):
    fenetre.fill(background)                On remplit la fenêtre avec la couleur background
    fenetre.blit(perso, (0,0))              On colle le personnage en haut à gauche
    perso_rect = perso.get_rect()           On crée le rectangle du personnage
    perso_rect.left = 70                    On place le rectangle du personnage à 70px
    liste_obs = init_obs(obs)               On crée une liste d'obstacles
    score = 0                               On initialise le score
    gameover = 0                            On initialise la fin du jeu
    gravite = 1                             On initialise la gravité
    saut = 0                                On initialise le saut
    pygame.display.flip()                   On actualise la fenêtre
    continuer = 1                           On initialise la variable continuer
    clock = pygame.time.Clock()             On initialise le temps
    return perso_rect, liste_obs, score, gameover, gravite, saut, continuer, clock
```

Il y a aussi la fonction « affichage_fenetre » dans laquelle on colle les images dans la fenêtre et on l'actualise. Cela se fait une fois par image :

```
def affichage_fenetre(fenetre, background, score, police, perso, perso_rect, obs,liste_obs):
    fenetre.fill(background)                On remplit la fenêtre de la couleur background
    fenetre.blit(perso,perso_rect)          On colle le personnage dans son rectangle
    for i in range (4):                     Pour les 4 éléments de la liste d'obstacles :
        fenetre.blit(obs,liste_obs[i])          On les colle dans la fenêtre
    affichage_score(score, police, fenetre) On affiche le score
    pygame.display.flip()                   On actualise la fenêtre
```

D. Explication de la gestion des obstacles et problèmes rencontrés

1. Gestion de l'image des obstacles

- Pour commencer il faut charger l'image des obstacles à l'aide de la fonction « pygame.image.load ».
  L'image sera nommée « obs » dans l'ensemble du programme.
  Nous avons donc choisi d'utiliser l'image « tuyau.png » placée dans le sous-dossier « img ». Ce sous-dossier, placé dans le même dossier que le programme, permet ainsi de regrouper toutes les images du jeu et d'éviter les complications.
  Le format png nous a semblé être le format d'image le plus adapté à nos attentes puisqu'il n'y a pas de pertes d'informations lors d'une compression et qu'il est adapté aux images possédant peu de couleurs différentes, d'où une image peu lourde (3,10 Ko pour tuyau.png). Ce format gère en plus les effets de transparence.

  Dans le programme cela donne :

```
obs = pygame.image.load("img/tuyau.png")
```

- Ensuite nous avons redimensionné l'image en 50px par 1000px grâce à la fonction « pygame.transform.scale » de pygame. Cette dernière prend en paramètre le nom de l'image et ses nouvelles dimensions.

  Dans le programme cela donne :

```
obs = pygame.transform.scale(obs, (50,1000))
```

2. Initialisation des obstacles

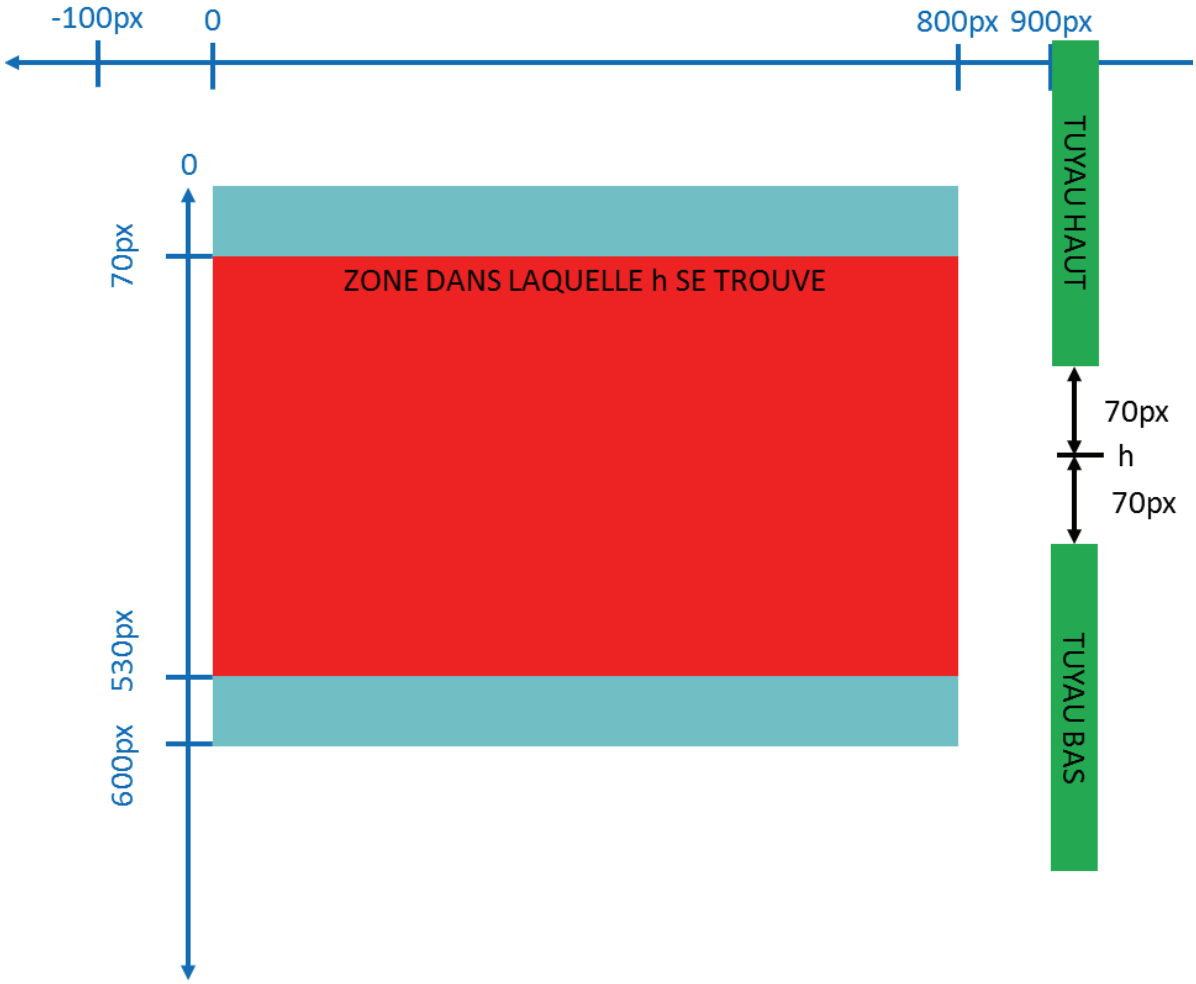
➢ **Pour commencer nous avons affiché un seul obstacle constitué d'une partie haute et d'une partie basse.**

- Il faut tout d'abord obtenir les rectangles des images « obs » à l'aide de la fonction « .get_rect() ». On pourra ainsi facilement bouger ces rectangles et réafficher l'image « obs » dans la fenêtre dans le reste du programme.

  Dans le programme cela donne :

```
obs_haut_rect=obs.get_rect()
obs_bas_rect=obs.get_rect()
```

- Dans un premier temps l'obstacle est placé hors de la fenêtre. Son bord gauche est alors situé à 900px.
  On détermine ensuite une hauteur aléatoire h de l'orifice où devra passer le personnage. Nous avons choisi que l'orifice devrait avoir une taille de 140px. Sachant que la hauteur de la fenêtre est de 600px, on en déduit que h doit être compris entre 70px et 530px.



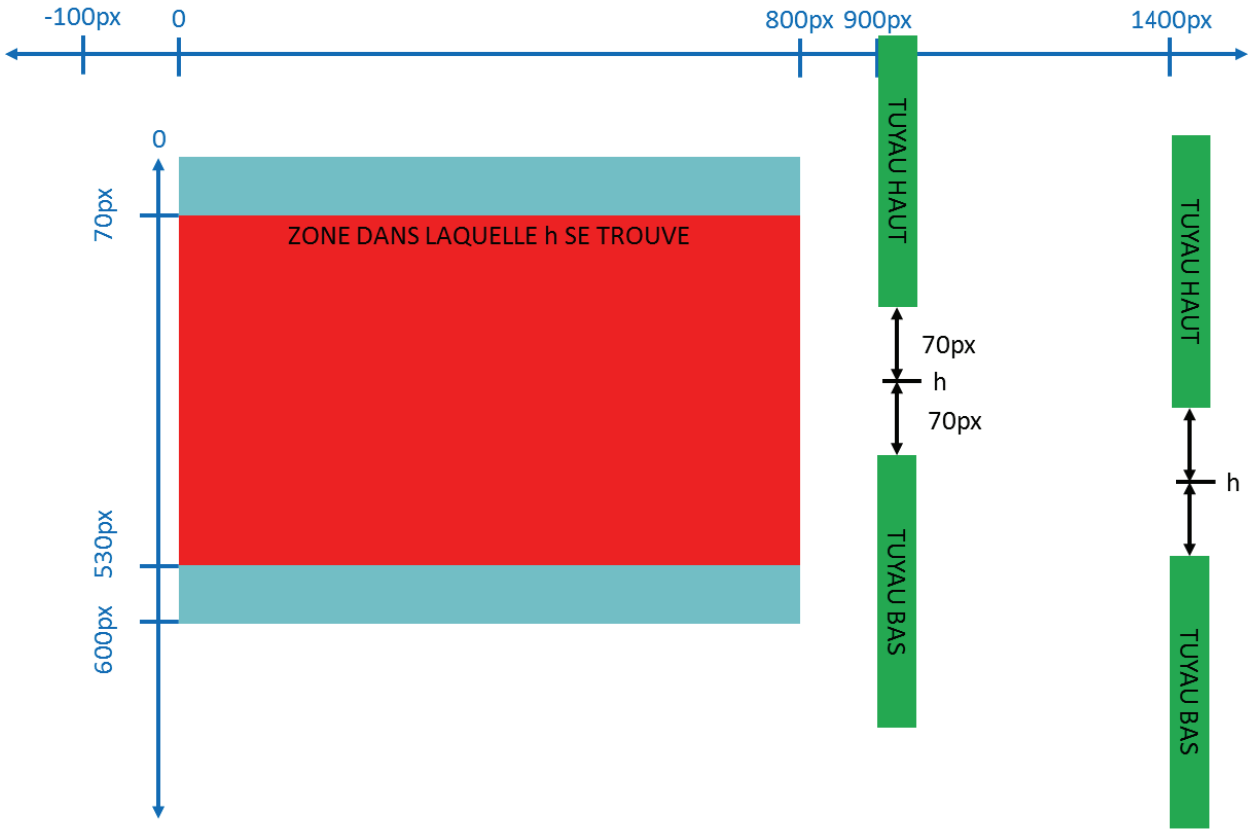
Dans le programme cela donne :

```
obs_haut_rect.left=900
obs_bas_rect.left=900
h=randint(70,530)
obs_haut_rect.bottom=h-70      Le bord bas de l'obstacle du haut vaut donc : h-70
obs_bas_rect.top=h+70          Le bord haut de l'obstacle du bas vaut donc : h+70
```

➢ **Nous avons ensuite ajouté une deuxième paire d'obstacles.**

- Nous avons tout d'abord ajouté tout simplement la paire d'obstacles de manière analogue pour avoir le bord gauche à 1400px.

  obs_haut_rect1.left = 1400
  obs_bas_rect1.left = 1400
  h = randint(70,530)
  obs_haut_rect1.bottom = h-70
  obs_bas_rect1.top = h+70



- Puis finalement nous avons rassemblé les obstacles dans la liste « liste_obs » pour économiser des lignes de code et alléger suite du programme.

  Dans le programme cela donne :

  liste_obs = [obs_haut_rect, obs_bas_rect, obs_haut_rect1, obs_bas_rect1]

- Le tout est alors placé dans la fonction « init_obs » qui prend en paramètre l'image « obs » et retourne la liste « liste_obs ».

  Dans le programme cela donne :

  def init_obs(obs):
  …

```
return liste_obs
```

3. Déplacement des obstacles

- Après différents tests nous avons finalement décidé de déplacer les obstacles vers la gauche de 3px par image. Le bord gauche de l'obstacle est ensuite replacé à 900px lorsque ce dernier atteint -100px (c'est-à-dire hors de la fenêtre) et la hauteur h de l'orifice est redéfinie aléatoirement. L'utilisateur perçoit alors un défilement infini d'obstacles avec un orifice d'une hauteur variable.
  De plus si on reprend le schéma de la page précédente, on remarque ainsi qu'il devrait toujours y avoir un écart de 500px entre chaque paire d'obstacles.

  Dans le programme cela aurait dû donner cela pour une seule paire d'obstacles :

```
obs_haut_rect=obs_haut_rect.move(-3,0)
obs_bas_rect=obs_bas_rect.move(-3,0)
if obs_haut_rect.left==-100:
    obs_haut_rect.left=900
    obs_bas_rect.left=900
    h=randint(70,530)
    obs_haut_rect.bottom=h-70
    obs_bas_rect.top=h+70
```

- Néanmoins après avoir codé le déplacement de la deuxième paire d'obstacles de façon similaire, seule la deuxième paire d'obstacles revenait en position initiale à 900px.
  En effet le bord gauche de la première paire d'obstacles n'atteignait jamais exactement la valeur de -100px, car (900-(-100))/3=333,33.
  Le bord gauche de la première paire d'obstacles passait ainsi de -99px à -102px.
  Pour détourner ce problème nous avons remplacé « ==-100 » par « <=-100 ».
- Avec cette modification le déplacement des obstacles est devenu correct même si l'écart entre les bords gauche des deux paires d'obstacles varie entre 498px et 501px.

- Nous avons finalement adapté notre programme pour utiliser la liste « liste_obs ».
  Nous utilisons donc une boucle For pour parcourir la liste. Mais étant donné que la « liste_obs » est constituée de **paires** d'obstacles (haut + bas), il faut utiliser un pas de **2**. Il a fallu en effet plusieurs essais avant de réussir à parcourir cette liste dans de bonne condition.
  Nous avons alors rassemblé le tout dans la fonction « deplac_obs » qui prend en paramètre et retourne « liste_obs ».

  Dans le programme cela donne :

```
def deplac_obs(liste_obs):
    for i in range (0,4,2):
        liste_obs[i] = liste_obs[i].move(-3,0)
```

```
        liste_obs[i+1] = liste_obs[i+1].move(-3,0)
        if liste_obs[i].left <= -100:
            liste_obs[i].left = 900
            liste_obs[i+1].left = 900
            h = randint(70,530)
            liste_obs[i].bottom = h-70
            liste_obs[i+1].top = h+70
    return liste_obs
```

- On appelle donc cette fonction une fois dans la file d'évènements ; sans oublier de rafraîchir l'affichage des obstacles dans la fenêtre, qu'on réalise avec la fonction « affichage » (voir p.4)

- On peut noter qu'on parcourt la liste « liste_obs » aussi dans la fonction « affichage ». Mais vu qu'on y affiche juste l'image « obs » dans chaque rectangle de la liste « liste_obs » on peut simplement utiliser un pas de **1**.

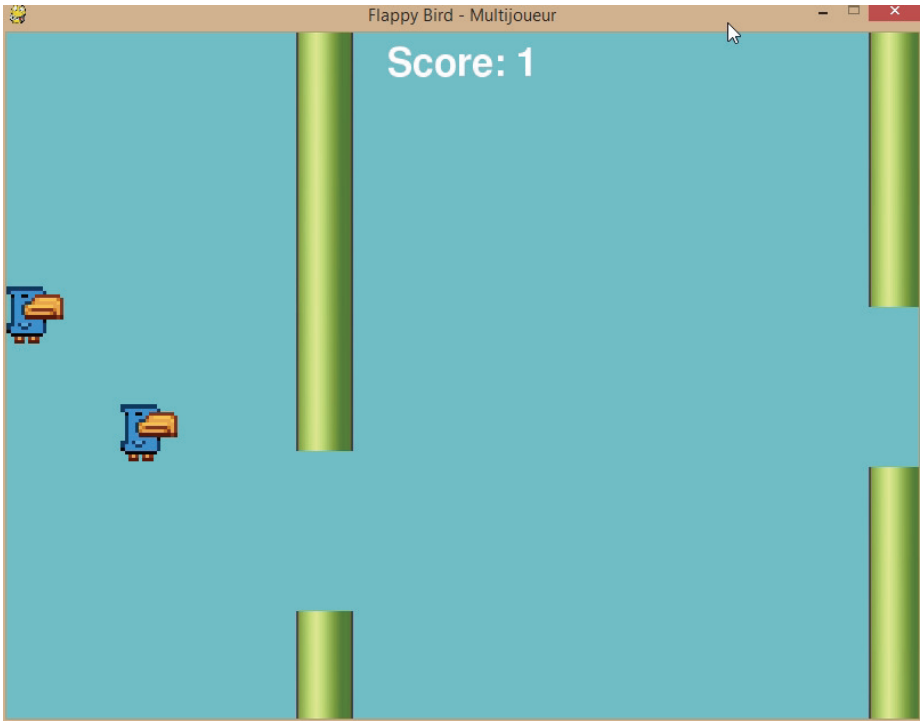
IV. Conclusion

Ainsi notre programme fonctionne sans problème majeur. Nous avons déjà envisagé quelques améliorations possibles :

- Possibilité de mettre le jeu en pause en appuyant sur la touche P
- Création d'un menu avec différents modes de jeux :
  - Un mode « inversé » où le personnage est soumis à la gravité inversée (attiré vers le haut et saut vers le bas)
  - Un mode « vengeance » où le personnage est immobile et où on contrôle les obstacles
  - Un mode « multi joueur »
  - Un mode avec des vies supplémentaires en bonus

Je suis personnellement satisfait d'avoir choisi l'option ISN. Cela a été une expérience très enrichissante, car cette option m'a fait découvrir les bases de la programmation informatique. J'ai notamment aimé découvrir comment passer des lignes de programme à une interface graphique.

J'ai aussi beaucoup apprécié travailler avec mes deux camarades Antoine et Loïc, avec qui j'ai une bonne entente. Nous avons principalement travaillé au lycée en cours d'ISN et au CDI, mais aussi à la maison par l'intermédiaire de Dropbox. Notre entraide nous a ainsi permis d'avancer rapidement et de corriger les erreurs de chacun. Cette organisation m'a également permis de commencer à développer le mode « multi joueur » en cette fin d'année.



Mode multi joueur