

LEGENDARY BIRD

SOMMAIRE

Présentation générale ----- 1

Bilan personnel ----- 1

L'explication du programme

Initialisation des variables ----- 2

Gestion des obstacles ----- 3-4

Gestion d'animation ----- 5

Gestion de collision ----- 6-7

Camouflage ----- 8

Boucle du jeu ----- 8

Annexe ----- 9

Présentation générale

Nous sommes partis sur un jeu classique: Flappy Bird. L'idée vient d'un camarade de classe nommé Hakim. Pygame est un module en python, spécialisé dans la création des jeux 2D, mais nous avons choisi le module Turtle car nous ne sommes pas comme les autres et Turtle est plus facile à maîtriser que Pygame avant le jour du Bac.

Le groupe est composé de 3 personnes: Ghillas, Xing et Clément.

Ghillas s'occupe de la gestion des obstacles, Xing réalise la collision entre le personnage et les obstacles mais également l'animation du personnage. Et enfin, Clément s'occupe de l'affichage du score et la musique du jeu.

En cours de la création, nous sommes contraint à charger moins d'images possible au risque de ralentissement et nous avons rencontré plusieurs problèmes. Notre programme est organisé sous la structure classique d'un jeu vidéo, c'est à dire des fonctions et une boucle s'exécutera en continue. Mais on peut pas modifier une variable dans une fonction il faudra mettre l'instruction GLOBAL devant la variable souhaitée. De plus on a voulu mettre l'initialisation des variables dans une fonction initialisation() mais ces variables initialisées deviennent locales, pour récupérer une variable, initialisée dans une fonction il faut encore utiliser GLOBAL mais le nombre de variables étant nombreux, donc finalement on a décidé de laisser ces variables hors-fonction.

Bilan personnel

J'ai vraiment apprécié réaliser ce projet. En effet, nous avons utilisé le module Turtle qui est un module que je ne maîtrisais pas avant de réaliser ce projet. Ce projet était donc comme un « défi » pour moi car j'ai dû apprendre comment fonctionnait ce module, découvrir toutes les fonctions que ce module propose. Cela n'a pas été une chose facile, mais su apprendre petit à petit comment il fonctionne. Ce projet fait à l'aide de Turtle m'a permis d'aller plus loin dans la programmation que simplement faire des jeux avec des lignes de code. Cela m'a permis d'approfondir mes connaissances et mes capacités du côté graphique dans la programmation d'un jeu vidéo, qui était une compétence que je n'avais pas énormément développé au cours de mon apprentissage de la programmation.

De plus, le fait de nous lancer dans un jeu comme Flappy Bird, nous a permis de réaliser un jeu que nous apprécions nous trois, ainsi que de nombreuses autres personnes, un jeu qui est ni trop facile, ni trop difficile à coder.

Ce projet m'a permis d'en apprendre encore un peu plus sur la programmation, d'aller toujours plus loin que ce qu'on peut faire et que ce qu'on voulait faire à l'origine. Cela m'a permis de gagner en rigueur et en organisation pour le travail de groupe, car nous avons dû bien nous répartir les tâches pour être le plus efficace possible et que chacun doit progresser au même rythme dans la réalisation de sa partie pour ne pas ralentir les autres et ne pas ralentir la progression dans la réalisation du programme.

L'explication du programme

Flappy Bird est jeux vidéo où le joueur doit contrôler un petit oiseau pour éviter les obstacles. Le joueur doit cliquer sur l'écran du jeu pour faire sauter l'oiseau et il doit avoir le plus de score possible.

On va vous expliquer les mécanismes essentiels de notre programme:

- l'initialisation des variable
- la gestion des obstacles
- la gestion d'animation
- la gestion de collision
- la boucle du jeu

Initialisation des variables

```
ecran = Screen()    Instanciation de la classe Screen
ecran.register_shape("resources/bird_up.gif")  on enregistre l'image pour pouvoir ensuite utiliser
avec turtle.shape()
ecran.register_shape("resources/bird_down.gif") l'image de bird
ecran.register_shape("resources/tuyau_up.gif") l'image de tuyau
ecran.register_shape("resources/tuyau_down.gif")
ecran.register_shape("resources/background.gif") l'image de fond
ecran.register_shape("resources/sol.gif") l'image du sol
ecran.register_shape("resources/bg_water.gif") l'image du sol
ecran.register_shape("resources/foreground.gif") l'image au premier plan
ecran.bgpic("resources/background.gif") on ajoute une image de fond
ecran.screensize(400,500)
ecran.setup (800,800) on définit la dimension de la fenêtre en 800x800 pixels.
ecran.title("Flappy bird") le titre de la fenêtre
```

```
bird = Turtle(shape="resources/bird_up.gif") instanciation de bird.
bird.penup() on leve le crayon pour éviter les traits lorsque bird se déplace
bird.speed(0) on met speed = 0 pour le déplacement instantané
bird.setpos(-130,0) la position de bird est x=-130 et y=0, c'est à dire à gauche de l'écran du jeu
bird_hauteur = 35 la hauteur de bird vaut 35 pixels
bird_largeur = 41 la largeur de bird vaut 41 pixels
La taille de bird est utilisé dans la collision entre bird et les obstacles
```

Gestion des obstacles

`listTuyaux = [Turtle() for t in range(4)]` On crée une liste qui comporte 4 tuyaux.

`for tuyau in listTuyaux:` Pour tous les tuyaux de la liste

`tuyau.speed(0)` on met `speed = 0` pour le déplacement instantané

`tuyau.penup()` on évite les traits

`tuyau_largeur = 26` le largeur d'un tuyau vaut 26 pixels

`tuyau_hauteur = 400` le largeur d'un tuyau vaut 400 pixels

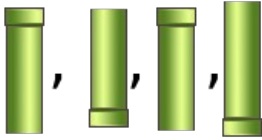
`ecart_Minimum = bird_hauteur` l'écart minimum entre le tuyau haut et le tuyau bas est la hauteur de *bird*.

`for i in range(0,4,2):` on a utilisé la boucle *For* pour parcourir la liste. *i* avance de 2 en 2 car les obstacles sont par paires, tuyau du haut et celui du bas. Ainsi *i* va de 0 à 3 (le 4 est exclu) et *i* prend la valeur 0 et 2.

`listTuyaux[i].shape("resources/tuyau_up.gif")` On met l'image du tuyau haut aux 1er et 3eme obstacles, c'est à dire `listTuyaux[0]` et `listTuyaux[2]`

`listTuyaux[i+1].shape("resources/tuyau_down.gif")` on met l'image aux ceux du 2eme et du dernier, donc `listTuyaux[1]` et `listTuyaux[3]`

Ainsi on obtient une list comme ceci:

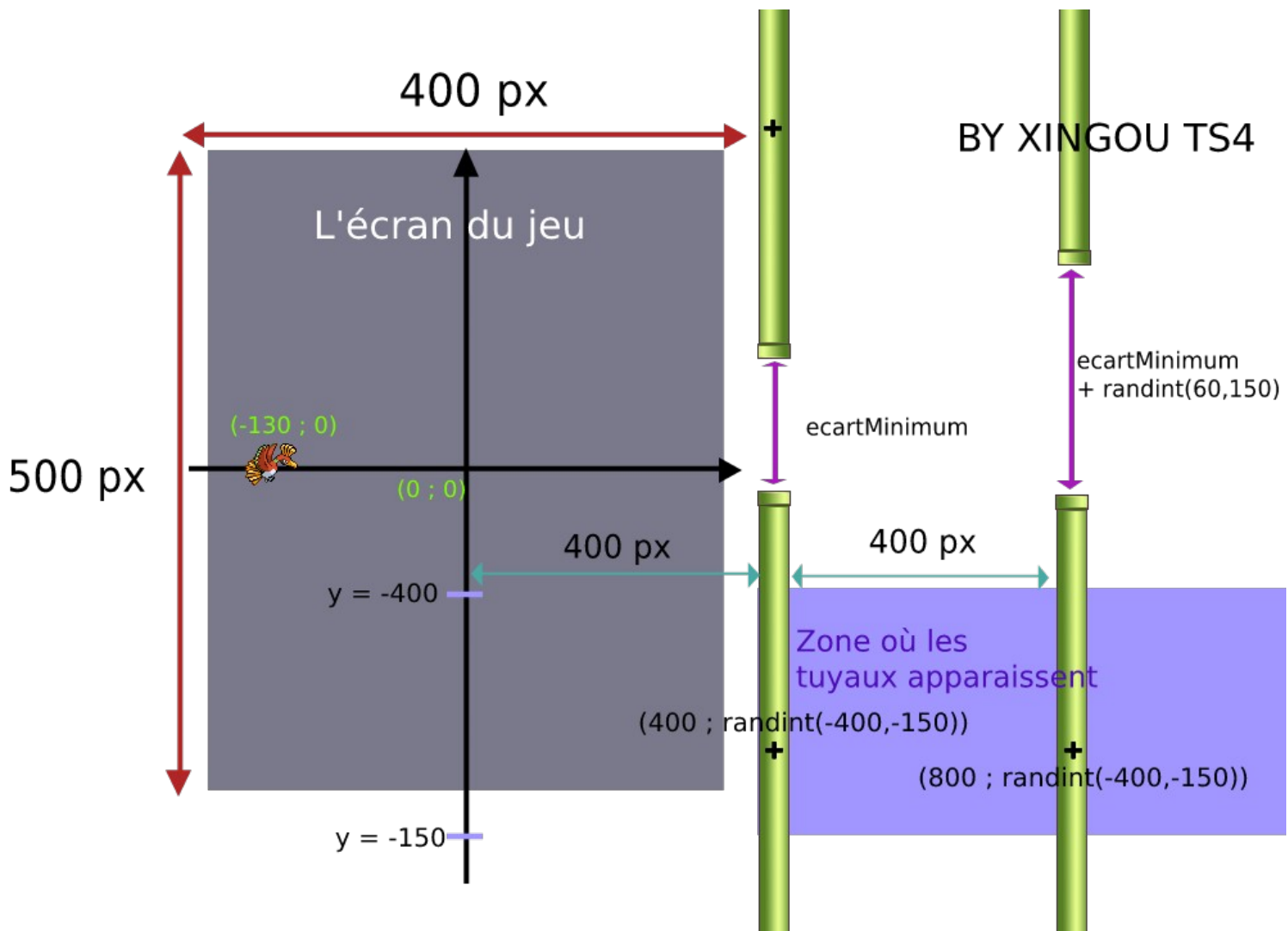
`listTuyaux = [`  `]`

0 1 2 3

`listTuyaux[i].setpos(listTuyaux[i-1].xcor() + 400,randint(-400,-150))` pour les tuyaux du bas, on détermine leurs coordonnées + 400 de la coordonnée x du précédent tuyau du bas pour éloigner les 2 paires de tuyaux. Le premier pair de tuyaux n'a pas de tuyaux précédents donc *i-1* vaut 0



`listTuyaux[i+1].setpos(listTuyaux[i].xcor(),listTuyaux[i].ycor() + tuyau_hauteur + ecart_Minimum + randint(60,150))` la coordonnée x des tuyaux du haut est la même que celle du bas. Ensuite pour la coordonnée y, on additionne d'abord la coordonnée y de l'obstacle du bas et la hauteur du tuyau. Ainsi ce paire de tuyaux sont "collés" entre eux, c'est pour cela qu'on doit ajouter l'écart minimum et d'une hauteur aléatoire.

L'image ci-dessous illustre le mécanisme des obstacles.



Gestion d'animation

`listAnim = ["resources/bird_down.gif", "resources/bird_up.gif"]` On crée une liste qui contient 2 images
`delay = 1.5` le délai entre chaque image est de 1.5
`index_img = 0` l'index de l'image est de 0, c'est à dire la première image dans la list
`compteur = 0` la mise en place d'un compte artificiel(sans module)

`listAnim = [ , ]`
0 1

```
def bird_animation():
    global compteur
    global index_img
    if compteur > delay:
        compteur = 0
        bird.shape(listAnim[index_img])
        index_img += 1 #on passe à l'image suivante
        index_img %= len(listAnim)
    else:
        compteur += 1
```

cette fonction contient l'ensemble de code qui gère l'animation
on doit mettre "global" devant la variable que l'on souhaite modifier dans la liste

On met une variable qui est le délai d'animation, c'est à dire le délai entre chaque changement d'image.

La variable `compteur` s'additionne tant qu'elle n'est pas supérieure à la variable `delay`

Si la condition `compteur > delay` est vérifiée, on change l'image de `bird`.

Le shape de `bird` est tiré de la `listAnim`.

`index_img` étant l'index de la liste, on l'ajoute 1 pour passer à la prochaine image et on fait `index_img` modulo de 2 pour le rebouclage de l'animation.

Le modulo % correspond au reste d'une division euclidienne, il permet le bouclage de l'animation.

`index_img` revient à 0 si `index_img` prend une valeur qui est dehors de la liste, ici, la variable hors liste est 2. Ainsi `index_img` bascule entre 0 et 1.

Une petite illustration de modulo:

`index_image % len(listAnim) = 2`

$$\begin{array}{r|l} 0 & 2 \\ -0 & 0 \\ \hline 0 & \end{array} \quad \begin{array}{r|l} 1 & 2 \\ -0 & 0 \\ \hline 1 & \end{array} \quad \begin{array}{r|l} 2 & 2 \\ -2 & 1 \\ \hline 0 & \end{array}$$

Lorsque `index_image = 2`
il revient à 0

Gestion de collision

```
if bird.xcor() - (bird_largeur/2) >= tuyau.xcor() + (tuyau_largeur/2)
or bird.xcor() + (bird_largeur/2) <= tuyau.xcor() - (tuyau_largeur/2)
or bird.ycor() - (bird_hauteur/2) >= tuyau.ycor() + (tuyau_hauteur/2)
or bird.ycor() + (bird_hauteur/2) <= tuyau.ycor() - (tuyau_hauteur/2):
    continue
else:
    return True
```

L'exemple ci-dessous illustre ces cas.

La box bleu est la box de collision de l'oiseau et les zones en magenta sont des zones de collision entre le tuyau et l'oiseau. Le but est de tester si les zones en magenta n'entrent pas en contact, alors la collision entre l'oiseau et le tuyau n'est pas valide.

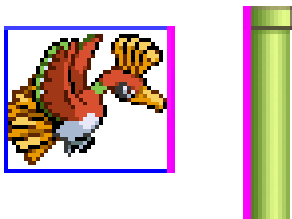
Pour savoir si *bird* est à droite du tuyau, on regarde si la coordonnée x de *bird* (son minimum en x) est plus grande que le maximum en x du tuyau (le maximum en x étant $\text{tuyau.xcor() + (tuyau_largeur/2)}$).

Une série d'image ci-dessous explique les 4 conditions de collision.

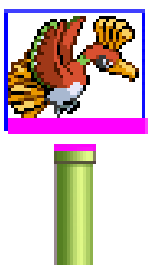
if bird.xcor() - (bird_largeur/2) >= tuyau.xcor() + (tuyau_largeur/2)



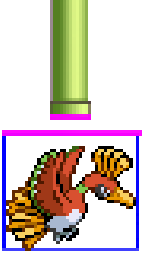
or bird.xcor() + (bird_largeur/2) <= tuyau.xcor() - (tuyau_largeur/2)



or bird.ycor() - (bird_hauteur/2) >= tuyau.ycor() + (tuyau_hauteur/2)



or bird.ycor() + (bird_hauteur/2) <= tuyau.ycor() - (tuyau_hauteur/2)



L'algorithme n'a pas réussi à exécuter sans erreur.

J'ai rencontré des soucis par rapport à *return False*. Le code ci-dessus est une version antérieure.

def testCollision():

for tuyau in listTuyaux:

if bird.xcor() - (bird_largeur/2) >= tuyau.xcor() + (tuyau_largeur/2)

or bird.xcor() + (bird_largeur/2) <= tuyau.xcor() - (tuyau_largeur/2)

or bird.ycor() - (bird_hauteur/2) >= tuyau.ycor() + (tuyau_hauteur/2)

or bird.ycor() + (bird_hauteur/2) <= tuyau.ycor() - (tuyau_hauteur/2):

return False

else:

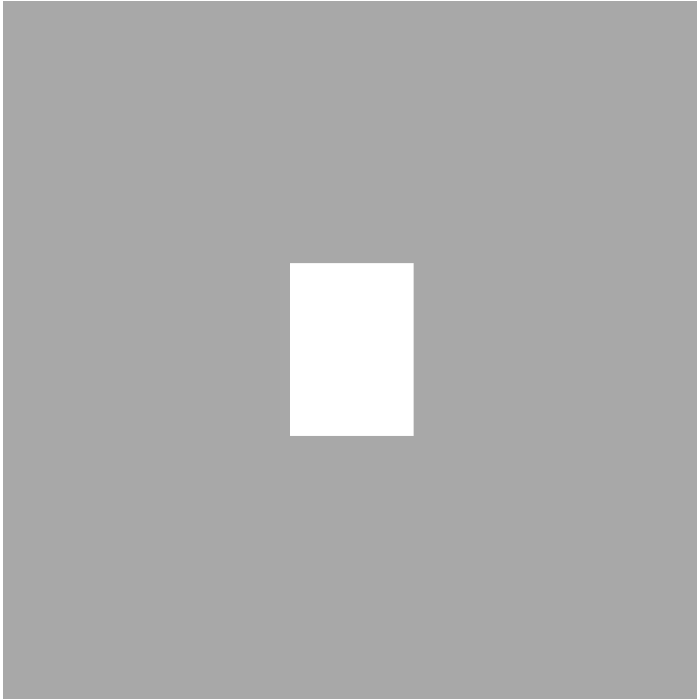
return True

Le problème c'est que *return False* quitte la boucle *for tuyau in listTuyaux* et la fonction *testCollision()*, ce qui fait que la boucle *for tuyau in listTuyaux* fonctionne qu'avec le premier objet de la liste. Pour pouvoir continuer de tester la collision avec TOUS les éléments de la liste, j'ai remplacé *return False* par *continue*. Ce dernier permet de continuer une boucle *for*, sans quitter la fonction.

Le camouflage

La fenêtre du jeu est définie en 800x800, malgré cela, le joueur peut agrandir la fenêtre et il peut voir le mécanisme de déplacement de tuyaux en dehors du terrain de jeu qui est défini en 400x500px. Pour garder le mystère du jeu, on a eu l'idée de faire une image où le centre (400x500px) est transparent, et le bord est gris.

Ainsi, on a créé l'image ci-dessous:



Puis on charge cet image tout à la fin des autres variables pour que Turtle soit en premier plan.
foreground = Turtle(shape="resources/foreground.gif") On initialise Turtle en dernier pour qu'il soit en premier plan.

La boucle du jeu

C'est dans cette fonction principale que va fonctionner les mécanismes du jeu. En effet la boucle permet de tourner le jeu en permanent.

def principale():

while gestion_Collision() != True and exitEcran() != True : le jeu continue tant qu'il n y a pas eu la collision et que bird n'a pas quitté l'écran, si l'une des conditions est fausse alors on quitte la boucle

bird.goto(bird.xcor(),bird.ycor() - 10) le bird tombe

bird_animation() on fait fonctionner l'animation des ailes de bird

deplacement_tuyaux()

deplacement_sol()

Annexe

Les différents mécanismes du jeu

La présentation de compte-rendu et le système de obstacles avec la liste sont inspirés du projet ISN de Lycée Louis Marchal(page 9 - 10)

http://isn-marchal.com/site/dossier_exemple.pdf

Le coup de modulo % dans la gestion d'animation du personnage provient de *Reddit* de Pygame.

https://www.reddit.com/r/pygame/comments/3uqpu2/what_is_the_best_way_to_animate_sprites/

La theorie de collision vient du site *jeux.developpez.com* (Partie IV: Collision entre deux Axis Aligned Bounding Box)

<https://jeux.developpez.com/tutoriels/theorie-des-collisions/formes-2d-simples/>

Les sources

Tout l'image du jeu est tiré de *spritters-resource.com*

<https://www.spritters-resource.com/>

The Spritters Resource DS / DSi - Pokémon HeartGold / SoulSilver - Johto Pokémon

https://www.spritters-resource.com/ds_dsi/pokemonheartgoldsoulsilver/sheet/26795/

The Spritters Resource Game Boy Advance - Super Mario Advance 4: Super Mario Bros. 3 - Backgrounds

https://www.spritters-resource.com/game_boy_advance/sma4/sheet/35822/

The Spritters Resource Game Boy Advance - Super Mario Advance 4: Super Mario Bros. 3 - Tiles

https://www.spritters-resource.com/game_boy_advance/sma4/sheet/33489/

The Spritters Resource Mobile - Flappy Bird - Version 1.2 Sprites

<https://www.spritters-resource.com/mobile/flappybird/sheet/59894/>

Google Pixel Mockup Vector

<https://www.vecteezy.com/vector-art/226398-google-pixel-mockup-vector>

The Sounds Resource Mobile - Flappy Bird - Everything

<https://www.sounds-resource.com/mobile/flappybird/sound/5309/>

Walking the Plains (8-Bit Remix) - New Super Mario Bros, Youtube

<https://www.youtube.com/watch?v=m5mfVmFvJlg>

Les illustrations sont réalisés avec Photopea

<https://www.photopea.com/>