Desktop Pet

Eileen Rheinboldt-Tran

Table des matières

Table des matières 1
Introduction 2
Planning 2
Arborescence dossier 4
Explication du code 4
__init__ 5
change_color 8
do_popup 10
move_window 11
swim_back 11
update_swim 11
go_back 11
update 12
is_updating 14
no_eat 15
random_update 15
Améliorations 16

Introduction

Un desktop pet est un animal virtuel qui va se balader sur la barre des tâches de l'ordinateur. Les plus connus sont les personnages *Shimeji*[1] ou le *desktop goose* inspiré du jeu *Untitled Goose*[2].

Mon animal est un axolotl tout mignon. Je me suis inspirée de cette image : https://pbs.twimg.com/media/EzZm4pXVoAQJSw-.jpg:large , pour créer les gifs sur *Piskel*[3].

J'ai choisi ce sujet, car j'ai toujours voulu faire un desktop pet moi-même, mon propre animal personnalisé.

J'ai utilisé le langage de programmation Python.

Les modules à télécharger sont énoncés dans le fichier requirements.txt.

Planning

24 janvier : finir gif axolotl rose

31 janvier : axolotl apparaît sur l'écran sur la barre des tâches

7 février : on peut placer l'axolotl où on veut sur l'écran (nage pour se replacer)

14 février : animations de bases fonctionnent toutes

–vacances–

28 février : animations sur commande, arriver à faire un clic droit pour modifier une action

7 mars : animations sur commande fonctionnent toutes

14 mars : finir toutes les animations pour toutes les couleurs

21 mars : tous les axolotls peuvent faire toutes les actions et on peut choisir sa couleur

28 mars : possibilité de donner un nom à l'axolotl (+ choisir le nombre d'axolotl)

4 avril : "bonus" mini-jeu (attraper des pommes ou autre) / interagir avec les fenêtres

– vacances pâques–

25 avril : continuer "bonus"

2 mai : continuer "bonus"

Fin : 9 mai

---

[1] https://shimejis.xyz/
[2] https://goose.game/
[3] https://www.piskelapp.com/

Les animations de « bases » devaient être les animations qui se font toutes seules et les « sur commande » devaient s'exécuter avec un clic droit, finalement j'ai fait un mix d'un peu tout.

Animations :

- normal
  - frames : 3
  - fps : 1
- dormir → se lever
  - frames : 10
  - fps : 3
- normal → dormir
  - frames : 8
  - fps : 3
- marcher (droite et gauche)
  - frames : 3
  - fps : 2
- dormir
  - frames : 5
  - fps : 2

- manger
  - frames : 8
  - fps: 5
- refus
  - frames : 4
  - fps : 3
- heureux
  - frames : 4
  - fps: 3
- fantôme
  - frames : 5
  - fps : 4
- nage
  - frames : 5
  - fps : 3
- colère
  - frames : 5
  - fps : 2

"bonus":

- mini-jeu
- interagir avec les fenêtres

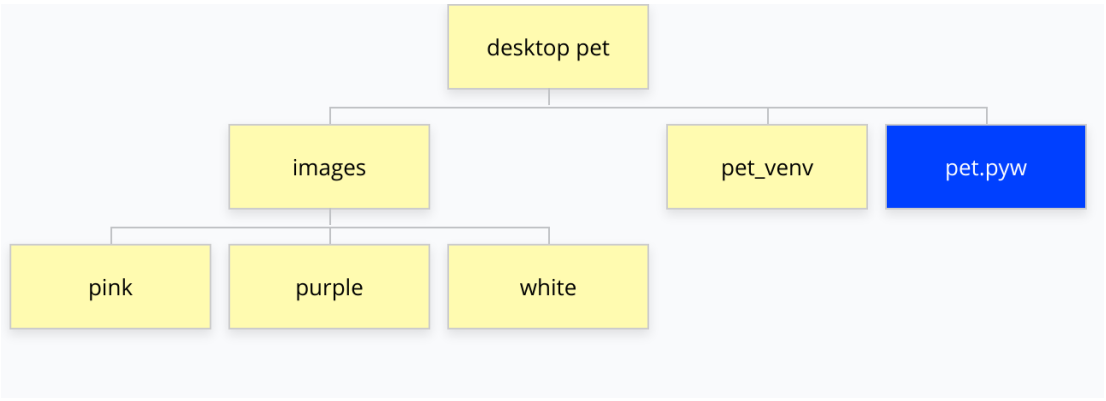
Malheureusement, je n'ai pas eu le temps de faire un bonus.

J'ai rencontré plusieurs problèmes durant mon projet.

Tout d'abord un problème causé par le matériel de l'école. Les éditeurs de textes python sont nuls ou ne sont pas mis à jour. Il arrive que l'éditeur de texte s'arrête subitement et ne se relance plus ou bien qu'il faille que je le redémarre pour pouvoir exécuter un fichier python. De plus, je ne pouvais donner aucune commande dans l'invite des commandes. Je ne pouvais donc ni télécharger un module avec la commande *pip*, ni changer de chemin, ni créer un environnement virtuel. J'ai perdu énormément de temps à cause de ça et ai consacré de nombreux cours à dessiner mon axolotl sur *Piskel* et non à programmer. Finalement, je me suis acheté un ordinateur portable que j'ai amené en cours.

Mon second problème était l'animation des gifs. Je suis restée bloquée pendant plusieurs cours dessus. C'était problématique, car c'est un élément très important du projet, mais à force de persévérer j'ai trouvé une solution. La solution est présentée dans *Explication du code*.

Arborescence dossier



Explication du code

Voir code pet.pyw.

L'extension .pyw permet d'exécuter le fichier python avec *pythonw.exe* et non *python.exe*, ce qui va supprimer le terminal au lancement.

```
#!C:/Users/eilee/Documents/Cours/informatique/desktop pet/pet_venv/Scripts/pythonw.exe
# first line selects a python interpreter
# the file is in .pyw to hide terminal
import tkinter as tk
import os
import PIL as pil
from PIL import Image, ImageTk
import random
os.chdir(f"{os.path.dirname(__file__)}")
```

La première ligne me permet d'utiliser l'environnement virtuel[4] *pet_venv* (que j'ai créé précédemment) sans avoir à le sélectionner sur l'éditeur de texte, ou pour qu'il soit automatiquement choisi quand le fichier est exécuté sans éditeur de texte. Il ne faut pas oublier de changer ou supprimer cette ligner si le code est exécuter sur un autre ordinateur.

Les lignes suivantes importent différents modules et la dernière ligne change le répertoire de travail actuel dans le dossier *desktop pet*.

Sautons directement à la fin du fichier.

```
if __name__ == '__main__':

    # we create our interface
    window = tk.Tk()
    interface = Interface(window)

    interface.mainloop()
    interface.destroy()
```

« if __name__ == '__main__': » autorise ou empêcher l'exécution de parties de code lors de l'importation des modules. Dedans, je crée mon interface graphique, pour ce faire, j'ai utilisé le module *Tkinter*. J'ai créé une classe nommée « Interface » qui prend en paramètre le cadre (« frame », container nous permettant de placer des objets et widgets) de tkinter. `class Interface(tk.Frame):`

Les différentes méthodes de cette classe sont :

- __init__
- change_color
- do_popup
- move_window
- swim_back
- update_swim
- go_back
- update
- is_updating
- no_eat
- random_update

__init__

__init__ est le constructeur de la classe, c'est-à-dire que la fonction est appelée à chaque fois qu'une classe est créée.

[4] Un environnement virtuel est un environnement d'exécution isolé, ça permet entre autres d'isoler des paquets.

```
def __init__(self, window, **kwargs):
        tk.Frame.__init__(self, window, width=128, height=128, **kwargs)
        self.pack(fill=tk.BOTH)

        self.master.configure(bg="#00ff08")
        # make window frameless
        self.master.overrideredirect(True)
        # make window draw over all others
        self.master.attributes('-topmost', True)
        self.master.bind("<B1-Motion>", self.move_window)
        self.master.bind("<ButtonRelease-1>", self.swim_back)
        # Create transparent window
        self.master.attributes('-transparentcolor', '#00ff08')

        self.h = self.master.winfo_screenheight()
```

Dedans, je spécifie la couleur de fond de la fenêtre, j'enlève le cadre de la fenêtre, je commande à la fenêtre de superposer toutes les autres afin que l'axolotl soit toujours visible. J'attache des action et boutons à des fonctions, et décide que la couleur verte #00ff08 devienne transparente. Mes gifs ont un fond vert et non transparent afin que le fond de la fenêtre soit transparent. A la dernière ligne, je stocke la hauteur de l'écran dans la variable self.h (self représente l'instance de la classe, ça nous permet d'accéder aux attributs et aux méthodes de celle-ci).

```
try:
      from win32api import GetMonitorInfo, MonitorFromPoint, GetSystemMetrics

      monitor_info = GetMonitorInfo(MonitorFromPoint((0, 0)))
      monitor_area = monitor_info.get("Monitor")
      work_area = monitor_info.get("Work")

      # find something better for self.taskbar_y, something more precise
      self.taskbar_y = self.h - (monitor_area[3]-work_area[3]) - 92
      x = f"+400+{self.taskbar_y}"

      # width screen
      self.width_screen = GetSystemMetrics(0)
      # height screen
      self.height_screen = GetSystemMetrics(1)
      # task bar y
      self.tby = self.taskbar_y
except:
      print("win32api doesn't work")
      x = "+400+625"

      # task bar y
      self.tby = 625
finally:
      window.geometry(x)       # x = position au début
```

Ce code permet de déterminer la position de la barre des tâches et d'assigner une position x de l'axolotl. Ici, j'utilise les blocs try/ except/ finally, car parfois le module win32api rencontre certains problèmes. La variable self.taskbar_y ne marchera pas sur tous les ordinateur et je veux trouver quelque chose de plus optimal.

```
        # label containing a frame of the axolotl
        self.axolotl = tk.Label(self, bg='#00ff08')

        self.color = "pink"
        self.change_color(self.color)

        self.axolotl.pack()

        # to stock the info for the next update
        self.to_update_list = []
        # to know when the update is finished
        self.update_finished = True
```

Je crée un label dans lequel je vais stocker une image, celle de l'axolotl selon son action et quelques variables qui seront utiles plus tard.

```
self.m = tk.Menu(self, tearoff=0)

self.m.add_command(label="normal", command=lambda: self.is_updating(
     0, self.gif_file_list[0], self.gif_fps_dic[self.gif_file_list[0]], 1))
self.m.add_command(label="sleep", command=lambda: self.is_updating(
     0, self.gif_file_list[1],  self.gif_fps_dic[self.gif_file_list[3]], 1))
self.m.add_command(label="walk", command=lambda: self.is_updating(
     0, self.gif_file_list[10],  self.gif_fps_dic[self.gif_file_list[10]], 1))
self.m.add_command(label="eat", command=self.no_eat)
self.m.add_command(label="angry", command=lambda: self.is_updating(
     0, self.gif_file_list[3],  self.gif_fps_dic[self.gif_file_list[3]], 1))
self.m.add_command(label="happy", command=lambda: self.is_updating(
     0, self.gif_file_list[6],  self.gif_fps_dic[self.gif_file_list[6]], 1))
self.m.add_command(label="monster", command=lambda: self.is_updating(
     0, self.gif_file_list[8],  self.gif_fps_dic[self.gif_file_list[8]], 1))
# color
self.m.add_separator()
self.sub = tk.Menu(self.m, tearoff=0)
self.sub.add_command(
     label="pink", command=lambda: self.change_color("pink"))
self.sub.add_command(
     label="white", command=lambda: self.change_color("white"))
self.sub.add_command(
     label="purple", command=lambda: self.change_color("purple"))

self.m.add_cascade(label = "color", menu= self.sub)

self.m.add_separator()
self.m.add_command(label="kill", command=self.master.quit)

self.axolotl.bind("<Button-3>", self.do_popup)
```

Ces lignes permettent de créer le menu qui s'affiche lors du clic droit. Le menu est séparé en trois parties : les actions, les différentes couleurs et l'action « kill » qui ferme la fenêtre et supprime donc l'axolotl.

```
# True when swimming
self.is_swimming = False

# counter of how many times the axolotl eats
self.nb_eat = 0

self.is_updating(
      0, self.gif_file_list[0],  self.gif_fps_dic[self.gif_file_list[0]], 1)
```

J'initialise encore quelques variables et appelle la méthode *is_updating* afin de lancer une animation.

change_color

Cette méthode permet de changer la couleur de l'axolotl et de créer un dictionnaire contenant les images de chaque gif et un autre le nombre d'images par secondes correspondant à chaque gif.

J'aurais pu ne créer qu'un dictionnaire regroupant les noms des gifs, leurs images par secondes et frames mais par praticité j'ai décidé d'en créer 3 différents afin de mieux me retrouver.

```
def change_color(self, color):
    """change axolotl's color"""
    self.color = color
    # list of all gifs
    self.gif_file_list = [f"images/{self.color}/normal.gif",
      f"images/{self.color}/normal-sleep.gif", f"images/{self.color}/sleep
      normal.gif", f"images/{self.color}/angry.gif",
      f"images/{self.color}/sleep.gif", f"images/{self.color}/eat.gif",
      f"images/{self.color}/happy.gif", f"images/{self.color}/refusal.gif",
      f"images/{self.color}/scared.gif", f"images/{self.color}/swim.gif",
      f"images/{self.color}/walk_r.gif", f"images/{self.color}/walk_l.gif"]

    # dic of all fps for each gif
    self.gif_fps_dic = {f"images/{self.color}/normal.gif": 1,
      f"images/{self.color}/normal-sleep.gif": 3,
      f"images/{self.color}/sleep-normal.gif": 3,
      f"images/{self.color}/angry.gif": 2,
      f"images/{self.color}/sleep.gif": 2, f"images/{self.color}/eat.gif": 5,
      f"images/{self.color}/happy.gif": 3,
      f"images/{self.color}/refusal.gif": 3,
      f"images/{self.color}/scared.gif": 4,
      f"images/{self.color}/swim.gif": 3,f"images/{self.color}/walk_r.gif": 2,
      f"images/{self.color}/walk_l.gif": 2}
    # dictionary containing a list of frames for each gif
    self.frames_dic = {}
    for gif in self.gif_file_list:
        gif_image = Image.open(gif)

        frames = []
        try:
            while True:
                # Get the next frame of the GIF
                gif_image.seek(len(frames))
                # Convert the frame to a Tkinter-compatible format
                tk_image = ImageTk.PhotoImage(gif_image)
                # Add the frame to the list
                frames.append(tk_image)
        except EOFError:
            pass

        self.frames_dic[f"{gif}"] = frames

        self.axolotl.config(
            image=self.frames_dic[self.gif_file_list[0]][0])
```

do_popup

La méthode *do_popup* permet de placer et de bouger le menu.

```
    def do_popup(self, event):
        """place and move menu"""
        try:
            self.m.tk_popup(event.x_root, event.y_root)
        finally:
            self.m.grab_release()
```

move_window

Cette méthode permet de bouger le menu même s'il n'a pas de cadre. Elle est liée au clic gauche de la souris quand elle est en mouvement.

```
def move_window(self, event):
    """to drag window even if it is frameless"""
    window.geometry(f'+{event.x_root}+{event.y_root}')
```

swim_back

Le rôle de cette méthode est d'appeler deux autres méthodes afin de contrôler la reposition du label et l'animation de l'axolotl. Elle est liée au relâchement du clic gauche de la souris.

```
def swim_back(self, event):
    """swim back to taskbar"""
    self.is_swimming = True
    self.go_back()
    self.update_swim()
```

update_swim

Cette méthode vérifie si l'axolotl est sur la barre des tâches, s'il n'y est pas, il nage sinon il se repositionne en position normale.

```
def update_swim(self):
        """check if the axolotl isn't on the taskbar
                if not --> swim
        """
        # empty the update_list
        self.to_update_list = []

        y = self.master.winfo_y()

        if y != self.tby:
            self.update( 0, self.gif_file_list[9],
                  self.gif_fps_dic[self.gif_file_list[9]], 1)
        else:
            self.is_swimming = False
            self.update( 0, self.gif_file_list[0],
                  self.gif_fps_dic[self.gif_file_list[0]], 1)
```

go_back

Cette méthode va changer la coordonnée y de la fenêtre pour venir replacer l'axolotl sur la barre des tâches.

```
def go_back(self):
    """go back to taskbar"""
    y = self.master.winfo_y()
    x = self.master.winfo_x()

    if y < self.tby:
        y += 5
        window.geometry(f"+{x}+{y}")
        self.after(100, self.go_back)

    elif y > self.tby:
        y -= 1
        window.geometry(f"+{x}+{y}")
        self.after(100, self.go_back)
```

update

La méthode *update* est l'une des plus importantes. C'est elle qui va gérer l'animation de l'axolotl. Elle prend en paramètres l'index de l'image, le gif à utiliser, le nombre d'images par seconde et le nombre de fois qu'il reste à répéter le gif. J'y ai rajouté des conditions afin que l'animation s'adapte au gif, par exemple si l'axolotl est allongé, j'aimerais qu'il se relève avant de faire une autre action. Si une animation est finie, alors on cherche la suivante dans *to_update_list* et si la liste est vide alors on appelle une animation au hasard avec la méthode que l'on va voir ensuite, *random_update*.

```
def update(self, frame_num, gif, fps, num_rep):
    """
            animate gif
    frame_num = index of the frame
    gif = gif to use
    fps = frame per second
    num_rep = number of times we want to repeat the gif
    """
    # not (self.is_swimming and gif != "images/swimming axolotl.gif") : so
      that the other animations stop when he swims
    if num_rep != 0 and not (self.is_swimming and gif !=
      self.gif_file_list[9]):
        self.update_finished = False
        self.axolotl.config(
            image=self.frames_dic[gif][frame_num])

        if frame_num == len(self.frames_dic[gif])-1:
            num_rep -= 1

        # if walk animation move right
        if gif == self.gif_file_list[10]:
            y = self.master.winfo_y()
            x = self.master.winfo_x()

            if x <= self.width_screen:
                x += 7
            window.geometry(f"+{x}+{y}")
        elif gif == self.gif_file_list[11]:
            y = self.master.winfo_y()
            x = self.master.winfo_x()

            if x >= 0:
                x -= 7
            window.geometry(f"+{x}+{y}")

        # int(1000/fps) --> because must be a int
        self.after(int(1000/fps), self.update, (frame_num+1) %
                   len(self.frames_dic[gif]), gif, fps, num_rep)
```

```
    elif gif == self.gif_file_list[9]:
        self.update_swim()
    # to return to the basic position
    # --> if already basic position it is useless

    elif gif == self.gif_file_list[1]:
        # if normal to sleep -- > sleep
        self.update(0, self.gif_file_list[4],
            self.gif_fps_dic[self.gif_file_list[4]], 2)

    elif gif == self.gif_file_list[4]:
        # if sleep -- > sleep to normal
        self.update(0, self.gif_file_list[2],
            self.gif_fps_dic[self.gif_file_list[2]], 1)

    elif gif != self.gif_file_list[0]:
        self.update(0, self.gif_file_list[0],
            self.gif_fps_dic[self.gif_file_list[0]], 1)

    elif len(self.to_update_list) != 0:

        fn = self.to_update_list[0][0]
        g = self.to_update_list[0][1]
        f = self.to_update_list[0][2]
        nr = self.to_update_list[0][3]

        self.to_update_list.pop(0)

        self.update(fn, g, f, nr)

    elif num_rep == 0:
        self.update_finished = True
        self.random_update()
```

is_updating

Cette méthode prend les mêmes paramètres que la méthode *update*. Son rôle est de vérifier si une animation est en cours, si oui alors elle stocke les informations dans une liste afin de lancer l'animation plus tard sinon elle lance l'animation directement. Elle permet d'éviter que deux animations se superposent.

```
def is_updating(self, frame_num, gif, fps, num_rep):
    """Check if update is running (an animation is in progress)
    if so --> stock the informations to call update later
    if not --> free to update"""
    if self.update_finished:
        self.update(frame_num, gif, fps, num_rep)

    else:
        # avoids an overload of commands
        if len(self.to_update_list) < 3:
            self.to_update_list.append([frame_num, gif, fps, num_rep])
```

no_eat

Si l'axolotl mange deux fois alors la troisième fois qu'on lui demande de manger, il refuse, car il n'a plus faim.

```
def no_eat(self):
    """if axolotl eats too much --> refuses food"""
    if self.nb_eat < 2:
        self.is_updating( 0, self.gif_file_list[5],
            self.gif_fps_dic[self.gif_file_list[5]], 1)
        self.nb_eat += 1
    else:
        self.is_updating( 0, self.gif_file_list[7],
            self.gif_fps_dic[self.gif_file_list[7]], 1)
        self.nb_eat = 0
```

random_update

Lorsque aucune animation n'est en cours alors on appelle *random_update* qui va lancer des animations au hasard. Les chances de tomber sur marcher à droite, à gauche et normal sont plus élevées que les autres actions.

```
def random_update(self):
    """when nothing is happening
    --> walk and other animations"""

    # because I don't want the animation sleep to normal
    i = 2
    while i == 2:
        # not 0 to 11 because I want there to be more chance of falling on
            walking or normal
        i = random.randrange(-5, 20, 1)

        if i > 15:
            i = 0
        if i > 11 and i < 15:
            i = 10
        if i < 0:
            i = 11

    # because i want more rep for these 2 gifs
    if i == 4 or i == 10 or i == 11:
        nbr_rep = 4
    else:
        nbr_rep = 1

    self.is_updating(0, self.gif_file_list[i],
            self.gif_fps_dic[self.gif_file_list[i]], nbr_rep)
```

Améliorations

Je pourrais encore améliorer et ajouter plein de choses.

Je peux ajouter des couleurs, une animation de mort, créer d'autre animaux qui ne sont pas des axolotls. Je peux aussi ajouter des mini-jeux ou des interactions avec les autres fenêtres.

De plus, je pourrais créer un fichier test qui va tester chacune de mes fonctions avec, par exemple, unittest.