**5. Code du Jeu : Explication des implémentation**

**widget Button**

from tkinter import \*

from tkinter import font as fontstyle

from PIL import Image, ImageTk

class custom\_Button(Frame):

    def \_\_init\_\_(self, parent, image, bg = None, fg = None, width = 100,

height = 100, command = None, text="", font=None,

sticky=None, rowspan=1, columnspan=1 , column=0, row=0,

padx=0, pady=0, ipadx=0, ipady=0, \*\*kwargs):

        super()**.**\_\_init\_\_(parent, bg=bg, \*\*kwargs)

        self**.**parent = parent

        self**.**image = image

        self**.**width, self**.**height = width, height

        self**.**text = text

        if font == None:

            self**.**font = fontstyle**.**Font(size=25, weight="bold")

        else:

            self**.**font = font

        self**.**command = command

        self**.**bg, self**.**fg = bg, fg

        self**.**column, self**.**row = column, row

        self**.**padx, self**.**pady = padx, pady

        self**.**ipadx, self**.**ipady = ipadx, ipady

        self**.**rowspan, self**.**columnspan = rowspan, columnspan

        self**.**sticky = sticky

        self**.**create\_buttons()

    def create\_buttons(self):

        def buttonClicked():

            if self**.**command and callable(self**.**command):

                self**.**command()

        photo = ImageTk**.**PhotoImage(

            Image**.**open(self**.**image)**.**resize((self**.**width, self**.**height),

Image**.**LANCZOS)

        )

        button = Button(self**.**parent, command=buttonClicked, text=self**.**text,

image=photo,

                        bg=self**.**bg, fg=self**.**fg, font=self**.**font,

                        cursor="hand2", compound=CENTER,

                        bd=0, highlightthickness=0,

highlightbackground="white",

                        activebackground=self**.**bg, activeforeground=self**.**fg)

        button**.**image = photo

        button**.**grid(column=self**.**column, row=self**.**row,

                    rowspan=self**.**rowspan, columnspan=self**.**columnspan,

                    padx=self**.**padx, pady=self**.**pady,

                    ipadx=self**.**ipadx, ipady=self**.**ipady,

                    sticky=self**.**sticky)

Classe pour le widget du bouton personnalisé. Ce bouton permet l'inclusion d'une image, ce qui évite la répétition du code. Cette fonctionnalité est souvent utilisée pour créer des boutons avec des apparences différentes dans le programme. Pour la gestion des images, le module Pillow est utilisé, tandis que la police du bouton est gérée grâce aux fonctionnalités de Tkinter.

**widget Image**

from tkinter import \*

from PIL import Image, ImageTk

class custom\_Image(Frame):

    def \_\_init\_\_(self, parent, image, text="", font = None, fg = None,

bg = None, width = 100, height = 100, sticky=None,

wraplength=None, rowspan=1, columnspan=1 , column=0, row=0,

padx=0, pady=0, ipadx=0, ipady=0, \*\*kwargs):

        super()**.**\_\_init\_\_(parent, bg=bg, \*\*kwargs)

        self**.**parent = parent

        self**.**image = image

        self**.**width, self**.**height = width, height

        self**.**text, self**.**font, self**.**wraplength = text, font, wraplength

        self**.**bg, self**.**fg = bg, fg

        self**.**column, self**.**row = column, row

        self**.**padx, self**.**pady = padx, pady

        self**.**ipadx, self**.**ipady = ipadx, ipady

        self**.**rowspan, self**.**columnspan = rowspan, columnspan

        self**.**sticky = sticky

        self**.**create\_image()

    def create\_image(self):

        photo = ImageTk**.**PhotoImage(

            Image**.**open(self**.**image)**.**resize((self**.**width, self**.**height),

Image**.**LANCZOS)

        )

        self**.**custom\_Image = Label(self**.**parent, image=photo, text=self**.**text,

compound="center", font=self**.**font,

fg=self**.**fg, bg=self**.**bg,

wraplength=self**.**wraplength)

        self**.**custom\_Image**.**image = photo

        self**.**custom\_Image**.**grid(column=self**.**column, row=self**.**row,

                               rowspan=self**.**rowspan,

columnspan=self**.**columnspan,

                               padx=self**.**padx, pady=self**.**pady,

                               ipadx=self**.**ipadx, ipady=self**.**ipady,

                               sticky=self**.**sticky)

Classe pour un widget d'image personnalisé. Ce label permet l'affichage d'une image, évitant ainsi la répétition du code. Cette fonctionnalité est souvent utilisée pour afficher des images dans le programme. Pour la gestion des images, le module Pillow est utilisé.

**Partie principale (main)**

from tkinter import \*

import ctypes as ct

import logzero

from logzero import logger

from pathlib import Path

from display**.**start import displayStart

from display**.**menu import displayMenu

from display**.**solo import displaySolo

from display**.**duo import displayDuo

from display**.**portail import displayPortail

from other**.**game**.**solo import solo

from other**.**game**.**duo import duo

from other**.**game**.**portail import portail

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

logzero**.**logfile(paths / "./data/logFile.log")

Ces lignes importent les éléments de base pour l'interface graphique à l'aide de Tkinter, ainsi que d'autres modules pour la gestion des chemins de fichiers avec Path, et la configuration du système de journalisation avec logzero. Ensuite, elles définissent le chemin absolu du fichier journal logFile.log dans le répertoire parent du fichier en cours d'exécution, permettant ainsi l'enregistrement des logs.

class Window(Tk):

    def \_\_init\_\_(self):

        self**.**color\_background = "#BFEA7C"

        self**.**color\_second = "#114232"

        self**.**color\_third = "#9BCF53"

        self**.**color\_fourth = "#82BA35"

        self**.**color\_text = "#ffffff"

        self**.**color\_text2 = "#000000"

        super()**.**\_\_init\_\_()

        self**.**title("Green Genius")

        self**.**geometry(f"{700}x{700}+{(self**.**winfo\_screenwidth() - 700) //

2}+40")

        self**.**resizable(False, False)

        self**.**config(bg=self**.**color\_background)

        self**.**titleBar()

    def titleBar(self):

        self**.**update()

        DWMWA\_USE\_IMMERSIVE\_DARK\_MODE = 20

        set\_window\_attribute = ct**.**windll**.**dwmapi**.**DwmSetWindowAttribute

        get\_parent = ct**.**windll**.**user32**.**GetParent

        hwnd = get\_parent(self**.**winfo\_id())

        rendering\_policy = DWMWA\_USE\_IMMERSIVE\_DARK\_MODE

        value = 2

        value = ct**.**c\_int(value)

        set\_window\_attribute(hwnd, rendering\_policy, ct**.**byref(value),

ct**.**sizeof(value))

    def display(self):

        self**.**grid\_rowconfigure(0, weight=1)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**home()

        try:

            self**.**mainloop()

        except KeyboardInterrupt:

            logger**.**info("App stopped.")

Cette classe Window hérite de Tk de Tkinter pour créer une fenêtre principale pour l'application "Green Genius". Dans son constructeur \_\_init\_\_(), elle configure les couleurs de fond et de texte, définie le titre et la géométrie de la fenêtre, et appelle la méthode titleBar() pour personnaliser la barre de titre. La méthode titleBar() utilise ctypes pour appliquer un style immersif à la barre de titre. La méthode display() configure la grille de la fenêtre et lance la boucle principale de l'interface utilisateur.

def home(self):

        self**.**display\_start = displayStart(self)

        self**.**display\_start**.**grid(row=0, column=0, sticky="nsew")

        logger**.**info("Home display")

def startGame(self):

        for content in self**.**display\_start**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**display\_menu = displayMenu(self)

        self**.**display\_menu**.**grid(row=0, column=0, sticky="nsew")

        logger**.**info("Start Game")

La méthode ‘home(self)’ crée et affiche la page d'accueil de l'application en utilisant la classe displayStart. Elle ajoute cette instance à la grille de la fenêtre principale. La méthode’ startGame(self)’ permet de passer de la page d'accueil au menu principal du jeu en supprimant tous les widgets actuellement affichés et en les remplaçant par la classe displayMenu. Elle enregistre également cet événement dans les logs.

def menuSolo(self):

        for content in self**.**display\_menu**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**display\_solo = displaySolo(self)

        self**.**display\_solo**.**grid(row=0, column=0, sticky="nsew")

        logger**.**info("Solo Game")

def menuDuo(self):

        for content in self**.**display\_menu**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**question, self**.**listQuestion = [], []

        self**.**display\_duo = displayDuo(self)

        self**.**display\_duo**.**grid(row=0, column=0, sticky="nsew")

        logger**.**info("Duo Game")

def menuPortail(self):

        for content in self**.**display\_menu**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**display\_portail = displayPortail(self)

        self**.**display\_portail**.**grid(row=0, column=0, sticky="nsew")

        logger**.**info("Portail Game")

Les méthodes ‘menuSolo(self)’, ‘menuDuo(self)’, et ‘menuPortail(self)’ permettent de naviguer entre les différents écrans de menu pour jouer en solo, en duo ou avec le portail. Chacune de ces méthodes retire tout contenu affiché actuellement dans le menu principal, puis instancie et affiche l'écran correspondant au mode de jeu sélectionné. Les événements sont également enregistrés dans les logs pour suivre les actions des utilisateurs.

def startQuizSolo(self, numberQuestion=20):

        for content in self**.**grid\_slaves():

            content**.**grid\_remove()

        solo(self, numberQuestion)

        logger**.**info("Start Solo Game")

    def startQuizDuo(self, user, token):

        for content in self**.**grid\_slaves():

            content**.**grid\_remove()

        duo(self, 20, user, token)

        logger**.**info("Start Duo Game")

    def startQuizPortail(self, token):

        for content in self**.**grid\_slaves():

            content**.**grid\_remove()

        portail(self, 10, token)

        logger**.**info("Start Portail Game")

Les méthodes ‘startQuizSolo(self, numberQuestion=20)’, ‘startQuizDuo(self, user, token)’, et ‘startQuizPortail(self, token)’ lancent respectivement le jeu en solo, en duo ou avec le portail. Chacune de ces méthodes retire tout contenu affiché actuellement dans la fenêtre principale, puis instancie et démarre la classe correspondante au mode de jeu sélectionné en transmettant les paramètres nécessaires. Les événements sont également enregistrés dans les logs pour suivre les actions des utilisateurs.

if \_\_name\_\_ == "\_\_main\_\_":

    Window()**.**display()

Le code if \_\_name\_\_ == "\_\_main\_\_": vérifie si le script est exécuté directement (c'est-à-dire pas importé en tant que module dans un autre script). Dans ce cas, il crée une instance de la classe Window et appelle sa méthode display() pour afficher la fenêtre principale de l'application.

**Accueil**

from tkinter import \*

from tkinter import font

from PIL import Image, ImageTk

import cv2

import random

from pathlib import Path

from widgets**.**Button import custom\_Button

from widgets**.**Image import custom\_Image

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ce bloc de code importe les modules nécessaires pour créer une interface graphique avec Tkinter, manipuler les polices avec le module font, gérer les images avec Pillow (Image, ImageTk), effectuer des opérations sur les images avec OpenCV (cv2), générer des nombres aléatoires avec random, et gérer les chemins de fichiers avec Path de la bibliothèque standard de Python. En outre, il importe deux widgets personnalisés, custom\_Button et custom\_Image, à partir de fichiers externes. Enfin, il définit le chemin absolu du répertoire parent du fichier en cours d'exécution à l'aide de Path.

class displayStart(Frame):

    def \_\_init\_\_(self, master):

        super()**.**\_\_init\_\_(master)

        self**.**canvas\_width = 604

        self**.**canvas\_height = 340

        self**.**canvas\_border = 6

        self**.**video\_source = [str(paths / "../assets/video/Car 1.mp4"),

str(paths / "../assets/video/Car 2.mp4")]

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**grid\_rowconfigure(0, weight=5)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=2)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**addComponents()

La classe displayStart(Frame) définit l'écran principal de l'application, qui comporte une animation de modèle. Dans son constructeur \_\_init\_\_(self, master), elle hérite de Frame de Tkinter. Elle configure les dimensions du canevas pour l'animation vidéo, définit les sources vidéo à utiliser, configure le style de fond en fonction du thème de l'application, et organise les composants graphiques dans une grille. Enfin, elle appelle la méthode addComponents() pour ajouter les composants nécessaires à l'écran.

def addComponents(self):

        custom\_Image(self, image=paths / "../assets/Background.png",

bg=self**.**master**.**color\_background, width=700, height=700,

column=0, row=0, rowspan=3)

        self**.**canvas = Canvas(self, height=self**.**canvas\_height,

width=self**.**canvas\_width,

bg=self**.**master**.**color\_second,

highlightthickness=self**.**canvas\_border,

highlightbackground="white")

        self**.**canvas**.**grid(column=0, row=0, padx=10, pady=10)

        self**.**cap = cv2**.**VideoCapture(random**.**choice(self**.**video\_source))

        self**.**video\_width = int(self**.**cap**.**get(cv2**.**CAP\_PROP\_FRAME\_WIDTH))

        self**.**video\_height = int(self**.**cap**.**get(cv2**.**CAP\_PROP\_FRAME\_HEIGHT))

        self**.**ratio = min(self**.**canvas\_width / self**.**video\_width,

self**.**canvas\_height / self**.**video\_height)

        custom\_Button(self,

                        command=self**.**master**.**startGame,

                        text="J O U E R", font=font**.**Font(size=30,

weight="bold"),

                        image=paths / "../assets/Button1.png",

                        height=90, width=412,

                        bg=self**.**master**.**color\_background,

fg=self**.**master**.**color\_text,

                        column=0, row=1, pady=(10,0), ipadx=5, ipady=2)

        custom\_Button(self,

                        command=self**.**master**.**quit,

                        text="Q U I T T E R", font=font**.**Font(size=15,

weight="bold"),

                        image=paths / "../assets/Button1.png",

                        height=50, width=230,

                        bg=self**.**master**.**color\_background,

fg=self**.**master**.**color\_text,

                        column=0, row=2, pady=(0, 50), ipadx=5, ipady=2)

        self**.**update\_frame()

Ces composants ajoutés à l'écran principal sont essentiels pour l'expérience utilisateur. L'image d'arrière-plan crée une ambiance visuelle attrayante, tandis que le canevas vidéo et les boutons offrent des fonctionnalités interactives pour démarrer le jeu ou quitter l'application. La méthode update\_frame() permet de maintenir la fluidité de l'animation vidéo en actualisant régulièrement les images affichées.

    def update\_frame(self):

        ret, frame = self**.**cap**.**read()

        if ret:

            new\_width = int(self**.**video\_width \* self**.**ratio) +

self**.**canvas\_border \* 2

            new\_height = int(self**.**video\_height \* self**.**ratio) +

self**.**canvas\_border \* 2 + 1

            resized\_frame = cv2**.**resize(frame, (new\_width, new\_height))

            frame\_rgb = cv2**.**cvtColor(resized\_frame, cv2**.**COLOR\_BGR2RGB)

            self**.**photo = ImageTk**.**PhotoImage(image=Image**.**fromarray(frame\_rgb))

            self**.**canvas**.**delete("all")

            self**.**canvas**.**create\_image((self**.**canvas\_width - new\_width) // 2,

(self**.**canvas\_height - new\_height) // 2, anchor=NW,

image=self**.**photo)

            self**.**after(33, self**.**update\_frame)

        else:

            self**.**cap**.**release()

            self**.**cap = cv2**.**VideoCapture(random**.**choice(self**.**video\_source))

            self**.**update\_frame()

Cette méthode update\_frame(self) est responsable de la mise à jour de l'image vidéo frame par frame. Elle lit une nouvelle image du flux vidéo à chaque appel, redimensionne cette image pour l'adapter au canevas, la convertit en format compatible avec Tkinter, puis l'affiche sur le canevas. Elle répète ce processus toutes les 33 millisecondes pour maintenir un affichage fluide de la vidéo. Si la lecture du flux vidéo s'arrête, elle redémarre la lecture en choisissant aléatoirement une nouvelle source vidéo parmi celles spécifiées.

**Menu**

from tkinter import \*

from pathlib import Path

from widgets**.**Button import custom\_Button

from widgets**.**Image import custom\_Image

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ce bloc de code importe les modules nécessaires pour créer une interface graphique avec Tkinter et gérer les chemins de fichiers avec `Path` de la bibliothèque standard de Python. De plus, il importe deux widgets personnalisés, `custom\_Button` et `custom\_Image`, à partir de fichiers externes. Enfin, il définit le chemin absolu du répertoire parent du fichier en cours d'exécution à l'aide de `Path`.

class displayMenu(Frame):

    def \_\_init\_\_(self, master):

        super()**.**\_\_init\_\_(master)

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**master**.**color\_background = "#BFEA7C"

        self**.**grid\_rowconfigure(0, weight=1)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=1)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**grid\_columnconfigure(1, weight=1)

        self**.**addComponents()

La classe displayMenu(Frame) définit l'écran du menu principal de l'application, où l'utilisateur peut choisir parmi les modes de jeu solo, duo ou portail. Dans son constructeur \_\_init\_\_(self, master), elle hérite de Frame de Tkinter et configure le style de fond en fonction du thème de l'application. Elle organise également les composants graphiques dans une grille pour un agencement approprié. Enfin, elle appelle la méthode addComponents() pour ajouter les composants nécessaires à l'écran.

   def addComponents(self):

        custom\_Image(self, image=paths / "../assets/Background.png",

bg=self**.**master**.**color\_background, width=700, height=700,

column=0, columnspan=2, row=0, rowspan=3)

        self**.**navbar = Frame(self)

        self**.**navbar**.**grid(column=0, columnspan=2, row=0)

        custom\_Image(self**.**navbar, image=paths / "../assets/Logo.png",

                     bg=self**.**master**.**color\_background,

                     width=571, height=82,

                     column=0, columnspan=2, row=0)

        custom\_Button(self**.**navbar, image=paths / "../assets/menu/Return.png",

                      command=self**.**master**.**home,

                      bg=self**.**master**.**color\_second,

                      width=55, height=55,

                      column=1, row=0, sticky=E,

                      padx=(0, 20))

        custom\_Button(self,

                        command=self**.**master**.**menuSolo,

                        image=paths / "../assets/menu/Solo.png",

                        height=280, width=268,

                        bg=self**.**master**.**color\_background,

                        column=0, row=1, padx=(30, 0), ipadx=5, ipady=2)

        custom\_Button(self,

                        command=self**.**master**.**menuDuo,

                        image=paths / "../assets/menu/Duo.png",

                        height=280, width=268,

                        bg=self**.**master**.**color\_background,

                        column=1, row=1, padx=(0, 30), ipadx=5, ipady=2)

        custom\_Button(self,

                        command=self**.**master**.**menuPortail,

                        image=paths / "../assets/menu/Portail.png",

                        height=168, width=571,

                        bg=self**.**master**.**color\_background,

                        column=0, row=2, ipadx=5, ipady=2, columnspan=2)

Ces composants ajoutés à l'écran du menu principal permettent à l'utilisateur de naviguer facilement entre les différentes options de jeu. L'image d'arrière-plan crée une ambiance visuelle cohérente, tandis que la barre de navigation offre une interface familière avec le logo de l'application et un bouton de retour. Les boutons du menu principal permettent à l'utilisateur de choisir entre les modes de jeu solo, duo ou portail, en offrant une expérience interactive et intuitive.

**Menu Solo**

from tkinter import \*

from tkinter import font

from pathlib import Path

from widgets**.**Button import custom\_Button

from widgets**.**Image import custom\_Image

from other**.**json**.**JsonFile import readJsonFile

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ce bloc de code importe les modules nécessaires pour créer une interface graphique avec Tkinter, gérer les polices avec `font`, gérer les chemins de fichiers avec `Path` de la bibliothèque standard de Python, et lire des fichiers JSON à l'aide de `readJsonFile` à partir d'un fichier externe. De plus, il importe deux widgets personnalisés, `custom\_Button` et `custom\_Image`, à partir de fichiers externes. Enfin, il définit le chemin absolu du répertoire parent du fichier en cours d'exécution à l'aide de `Path`.

class displaySolo(Frame):

    def \_\_init\_\_(self, master):

        super()**.**\_\_init\_\_(master)

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**grid\_rowconfigure(0, weight=2)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=3)

        self**.**grid\_columnconfigure(0, weight=1)

        fileData = readJsonFile(paths / "../data/data.json")**.**get()

        try:

            if len(fileData["score"]) > 0:

                bestScore = 0

                for i in range(len(fileData["score"])):

                    if bestScore <=  fileData["score"][i]["score"]:

                        bestScore = fileData["score"][i]["score"]

                        try:

                            self**.**userName = fileData["score"][i]["name"]

                        except:

                            self**.**userName = "Inconnue"

                        self**.**userScore = fileData["score"][i]["score"]

            else:

                self**.**userName = "Inconnue"

                self**.**userScore = 0

        except:

            self**.**userName = "Inconnue"

            self**.**userScore = 0

        self**.**addComponents()

La classe `displaySolo(Frame)` définit l'écran du menu pour jouer seul (Solo) dans l'application. Dans son constructeur `\_\_init\_\_(self, master)`, elle hérite de `Frame` de Tkinter et configure le style de fond en fonction du thème de l'application. Elle organise également les composants graphiques dans une grille pour un agencement approprié. Enfin, elle charge les données du meilleur score à partir d'un fichier JSON et initialise les variables correspondantes.

def addComponents(self):

        custom\_Image(self, image=paths / "../assets/Background.png",

                     bg=self**.**master**.**color\_background,

                     width=700, height=700,

                     column=0, row=0, rowspan=3)

        self**.**navbar = Frame(self)

        self**.**navbar**.**grid(column=0, row=0)

        custom\_Image(self**.**navbar, image=paths /

"../assets/solo/Header\_Solo.png",

                     bg=self**.**master**.**color\_background,

                     width=571, height=82,

                     column=0, columnspan=2, row=0)

        custom\_Button(self**.**navbar, image=paths / "../assets/Home.png",

                      command=self**.**master**.**startGame,

                      bg=self**.**master**.**color\_second,

                      width=55, height=55,

                      column=1, row=0, sticky=E,

                      padx=(0, 20))

        self**.**score = Frame(self)

        self**.**score**.**grid(column=0, row=1, sticky=S)

        self**.**score**.**grid\_columnconfigure(0, weight=1)

        self**.**score**.**grid\_columnconfigure(1, weight=1)

        self**.**score**.**grid\_columnconfigure(2, weight=15)

        custom\_Image(self**.**score, image=paths / "../assets/Frame1.png",

                     bg=self**.**master**.**color\_background,

                     height=84, width=571,

                     column=0, row=0, columnspan=3)

        custom\_Image(self**.**score, image=paths / "../assets/solo/Trophy.png",

                     bg=self**.**master**.**color\_second,

                     width=50, height=50,

                     column=0, row=0,

                     sticky=W, padx=(20, 0))

        fontStyle = font**.**Font(size=30, weight="bold")

        self**.**name = Label(self**.**score, text=self**.**userName, font=fontStyle,

                          bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text)

        self**.**name**.**grid(column=1, row=0)

        fontStyle = font**.**Font(size=30, weight="bold")

        self**.**percentage = Label(self**.**score, text=str(self**.**userScore) + "%",

font=fontStyle, bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text)

        self**.**percentage**.**grid(column=2, row=0, sticky=E, padx=(0, 20))

        self**.**quiz = Frame(self)

        self**.**quiz**.**grid(column=0, row=2)

        custom\_Image(self**.**quiz, image=paths / "../assets/Frame2.png",

                     bg=self**.**master**.**color\_background,

                     height=320, width=571,

                     column=0, row=0, rowspan=2, columnspan=3)

        self**.**quiz**.**grid\_rowconfigure(0, weight=1)

        self**.**quiz**.**grid\_rowconfigure(1, weight=1)

        self**.**quiz**.**grid\_columnconfigure(0, weight=1)

        self**.**quiz**.**grid\_columnconfigure(1, weight=1)

        self**.**quiz**.**grid\_columnconfigure(2, weight=1)

        fontStyle = font**.**Font(size=25, weight="bold")

        self**.**text = Label(self**.**quiz, text="Choisis le nombre de questions et

réalise ton meilleur score", wraplength=525,

                          bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text, font=fontStyle)

        self**.**text**.**grid(column=0, columnspan=3, row=0, pady=0)

        fontStyle = font**.**Font(size=35, weight="bold")

        custom\_Button(self**.**quiz, image=paths / "../assets/Button2.png",

                      command=lambda: self**.**master**.**startQuizSolo(5),

                      text="5", font=fontStyle,

                      width=120, height=120,

                      bg=self**.**master**.**color\_second, fg=self**.**master**.**color\_text,

                      column=0, row=1)

        custom\_Button(self**.**quiz, image=paths / "../assets/Button2.png",

                      command=lambda: self**.**master**.**startQuizSolo(10),

                      text="10", font=fontStyle,

                      width=120, height=120,

                      bg=self**.**master**.**color\_second, fg=self**.**master**.**color\_text,

                      column=1, row=1)

        custom\_Button(self**.**quiz, image=paths / "../assets/Button2.png",

                      command=lambda: self**.**master**.**startQuizSolo(20),

                      text="20", font=fontStyle,

                      width=120, height=120,

                      bg=self**.**master**.**color\_second, fg=self**.**master**.**color\_text,

                      column=2, row=1)

Ces composants ajoutés à l'écran du menu Solo offrent à l'utilisateur une expérience complète et conviviale. La barre de navigation assure une navigation facile entre les différentes sections de l'écran, tandis que la visualisation du meilleur score personnel permet à l'utilisateur de suivre ses progrès. Les options de sélection du nombre de questions offrent une personnalisation du jeu, ce qui permet à l'utilisateur de choisir entre une expérience plus courte ou plus longue en fonction de ses préférences.

**Menu Duo**

from tkinter import \*

from tkinter import font, messagebox

from pathlib import Path

import random

from PIL import Image, ImageTk

from widgets**.**Image import custom\_Image

from widgets**.**Button import custom\_Button

from other**.**firebase**.**firestore import createGroup, joinGroup, storageQuestion, loadQuestion

from other**.**json**.**JsonFile import readJsonFileSchema

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ce bloc de code importe les modules nécessaires pour créer une interface graphique avec Tkinter, gérer les polices avec font, gérer les chemins de fichiers avec Path de la bibliothèque standard de Python, manipuler des images avec Image et ImageTk de PIL (Pillow), et afficher des boîtes de dialogue avec messagebox. De plus, il importe deux widgets personnalisés, custom\_Image et custom\_Button, à partir de fichiers externes. Enfin, il importe des fonctions pour interagir avec une base de données Firebase Firestore et lire des schémas JSON à partir d'un fichier externe.

class displayDuo(Frame):

    def \_\_init\_\_(self, master):

        super()**.**\_\_init\_\_(master)

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**grid\_rowconfigure(0, weight=1)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=1)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**grid\_columnconfigure(1, weight=1)

        self**.**addComponents()

La classe `displayDuo(Frame)` définit l'écran du menu pour jouer à deux (Duo) dans l'application. Dans son constructeur `\_\_init\_\_(self, master)`, elle hérite de `Frame` de Tkinter et configure le style de fond en fonction du thème de l'application. Elle organise également les composants graphiques dans une grille pour un agencement approprié. Enfin, elle appelle la méthode `addComponents()` pour ajouter les composants nécessaires à l'écran.

def addComponents(self):

        custom\_Image(self, image=paths / "../assets/Background.png",

                     bg=self**.**master**.**color\_background,

                     width=700, height=700,

                     column=0, columnspan=2, row=0, rowspan=3)

        self**.**navbar = Frame(self)

        self**.**navbar**.**grid(column=0, columnspan=2, row=0)

        custom\_Image(self**.**navbar, image=paths /

"../assets/duo/Header\_Duo.png",

                     bg=self**.**master**.**color\_background,

                     width=571, height=82,

                     column=0, columnspan=2, row=0)

        custom\_Button(self**.**navbar, image=paths / "../assets/Home.png",

                      command=self**.**master**.**startGame,

                      bg=self**.**master**.**color\_second,

                      width=55, height=55,

                      column=1, row=0, sticky=E,

                      padx=(0, 20))

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Join\_Group2.png")**.**resize((275,

204), Image**.**LANCZOS)

        )

        self**.**button\_join = Button(self, command=self**.**join\_group, image=photo,

                                bg=self**.**master**.**color\_background,

                                cursor="hand2", compound=CENTER,

                                bd=0, highlightthickness=0,

highlightbackground="white",

                                activebackground=self**.**master**.**color\_background)

        self**.**button\_join**.**image = photo

        self**.**button\_join**.**grid(column=0, row=1, padx=(55, 0), ipadx=5, ipady=2)

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Create\_Group1.png")**.**resize((275,

204), Image**.**LANCZOS)

        )

        self**.**button\_create = Button(self, command=self**.**create\_group,

image=photo,

                                bg=self**.**master**.**color\_background,

                                cursor="hand2", compound=CENTER,

                                bd=0, highlightthickness=0,

highlightbackground="white",

                                activebackground=self**.**master**.**color\_background)

        self**.**button\_create**.**image = photo

        self**.**button\_create**.**grid(column=1, row=1, padx=(0, 55), ipadx=5,

ipady=2)

        self**.**frame = Frame(self)

        self**.**frame**.**grid(column=0, columnspan=2, row=2)

        self**.**join\_group()

La méthode `addComponents(self)` de la classe `displayDuo` ajoute les composants graphiques à l'écran du menu pour jouer à deux (Duo). Cela inclut une barre de navigation avec un bouton de retour à l'écran d'accueil, ainsi que des boutons pour rejoindre ou créer un groupe de jeu.

    def join\_group(self):

        self**.**loopCreate = True

        photo\_join = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Join\_Group2.png")**.**resize((275,

204), Image**.**LANCZOS)

        )

        self**.**button\_join**.**config(image=photo\_join)

        self**.**button\_join**.**image = photo\_join

        photo\_create = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Create\_Group1.png")**.**resize((275,

204), Image**.**LANCZOS)

        )

        self**.**button\_create**.**config(image=photo\_create)

        self**.**button\_create**.**image = photo\_create

        for content in self**.**frame**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**frame**.**grid\_rowconfigure(0, weight=1)

        self**.**frame**.**grid\_rowconfigure(1, weight=1)

        self**.**frame**.**grid\_rowconfigure(2, weight=1)

        custom\_Image(self**.**frame, image=paths / "../assets/Frame3.png",

                     bg=self**.**master**.**color\_background,

                     height=274, width=571,

                     column=0, row=0, rowspan=3)

        fontStyle = font**.**Font(size=25, weight="bold")

        self**.**text = Label(self**.**frame, text="Saisis le code de la salle pour la

rejoindre", wraplength=487,

                          bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text, font=fontStyle)

        self**.**text**.**grid(column=0, row=0, pady=0)

        fontStyle = font**.**Font(size=27, weight="bold")

        self**.**entry = Text(self**.**frame, bg=self**.**master**.**color\_fourth,

                          width=20, height=1,

                          font=fontStyle, fg=self**.**master**.**color\_text,

insertbackground=self**.**master**.**color\_text,

                          highlightthickness=4, highlightbackground="white",

highlightcolor="white")

        self**.**entry**.**grid(column=0, row=1, ipadx=5)

        self**.**center\_text(None)

        self**.**entry**.**bind("<KeyRelease>", self**.**center\_text)

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Join\_Button.png")**.**resize((303,

51), Image**.**LANCZOS)

        )

        self**.**button\_joined = Button(self**.**frame, command=lambda: self**.**join(self**.**entry**.**get(1.0, END)**.**replace('\n', '')), image=photo,

                                    bg=self**.**master**.**color\_second,

                                    cursor="hand2", compound=CENTER,

                                    bd=0, highlightthickness=0,

highlightbackground="white",

                                    activebackground=self**.**master**.**color\_second)

        self**.**button\_joined**.**image = photo

        self**.**button\_joined**.**grid(column=0, row=2, ipadx=5, ipady=2)

Cette partie de la classe displayDuo est dédiée à l'interface pour rejoindre un groupe de jeu à deux. Elle modifie l'apparence des boutons pour indiquer l'action actuelle (dans ce cas, rejoindre un groupe), et elle affiche une interface pour saisir le code de la salle à rejoindre. Une fois le code saisi, l'utilisateur peut appuyer sur le bouton pour rejoindre le groupe.

    def join(self, token):

        self**.**join\_group\_connexion = joinGroup(token)

        if self**.**join\_group\_connexion**.**report:

            self**.**loadQuestion = loadQuestion(token)

            self**.**master**.**question = self**.**loadQuestion**.**question

            self**.**master**.**listQuestion = self**.**loadQuestion**.**listQuestion

            self**.**master**.**startQuizDuo(1, self**.**entry**.**get(1.0, END)**.**replace('\n',

''))

Dans cette partie de la classe displayDuo, la méthode join(token) est responsable de rejoindre un groupe dans la base de données. Elle utilise la fonction joinGroup(token) pour tenter de se connecter au groupe en utilisant le token fourni. Si la connexion réussit, elle charge les questions associées à ce groupe en utilisant la fonction loadQuestion(token). Ensuite, elle utilise la méthode startQuizDuo(1, token) du maître (probablement la fenêtre principale) pour démarrer le quiz en mode duo avec l'utilisateur actuel et le token du groupe.

    def create\_group(self):

        self**.**loopCreate = False

        photo\_join = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Join\_Group1.png")**.**resize((275,

204), Image**.**LANCZOS)

        )

        self**.**button\_join**.**config(image=photo\_join)

        self**.**button\_create**.**image = photo\_join

        photo\_create = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/duo/Create\_Group2.png")**.**resize((275,

204), Image**.**LANCZOS)

        )

        self**.**button\_create**.**config(image=photo\_create)

        self**.**button\_join**.**image = photo\_create

        for content in self**.**frame**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**frame**.**grid\_rowconfigure(0, weight=1)

        self**.**frame**.**grid\_rowconfigure(1, weight=1)

        custom\_Image(self**.**frame, image=paths / "../assets/Frame3.png",

                     bg=self**.**master**.**color\_background,

                     height=274, width=571,

                     column=0, row=0, rowspan=2)

        fontStyle = font**.**Font(size=25, weight="bold")

        self**.**text = Label(self**.**frame, text="Donne le code a ton amis pour

jouer a deux", wraplength=487,

                          bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text, font=fontStyle)

        self**.**text**.**grid(column=0, row=0, pady=0)

        self**.**create\_group\_connexion = createGroup()

        self**.**readFile = readJsonFileSchema(paths /

"../data/question.json")**.**get()

        if self**.**readFile == []:

            self**.**error()

        self**.**listeType = []

        for data in self**.**readFile:

            self**.**listeType**.**append(data["type"])

        self**.**randomList = random**.**sample(range(0, len(self**.**listeType)), min(20,

len(self**.**listeType)))

        storageQuestion(self**.**create\_group\_connexion**.**id, self**.**readFile,

self**.**randomList)

        fontStyle = font**.**Font(size=27, weight="bold")

        self**.**code = custom\_Image(self**.**frame, image=paths /

"../assets/Frame4.png",

                                 font=fontStyle,

text=self**.**create\_group\_connexion**.**id,

                                 width=499, height=56,

                                 bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text,

                                 row=1, column=0)

        self**.**check\_report()

Dans la partie "Interface pour créer un groupe", une fois que l'utilisateur a déclenché l'action de création de groupe, l'interface est mise à jour pour afficher un message indiquant à l'utilisateur de transmettre le code de groupe à son partenaire de jeu. Le code unique du groupe est généré et affiché à l'utilisateur. Les questions sont sélectionnées aléatoirement à partir d'un fichier JSON contenant une liste de questions, et elles sont stockées dans la base de données associée au groupe nouvellement créé. Enfin, la méthode check\_report() est appelée pour vérifier si le stockage des questions s'est bien déroulé.

    def check\_report(self):

        if self**.**create\_group\_connexion**.**report:

            self**.**master**.**question = self**.**readFile

            self**.**master**.**listQuestion = self**.**randomList

            self**.**create\_group\_connexion**.**stop\_listening()

            self**.**master**.**startQuizDuo(2, self**.**create\_group\_connexion**.**id)

        elif not self**.**loopCreate:

            self**.**master**.**after(1000, self**.**check\_report)

Dans cette section, la méthode check\_report() vérifie le rapport de création du groupe. Si le rapport indique que le groupe a été créé avec succès, alors le jeu est démarré en mode duo avec les questions stockées dans le groupe nouvellement créé. Si le groupe n'a pas été créé avec succès et que l'utilisateur a choisi de créer un groupe (et non de rejoindre un groupe existant), la méthode réessaye de vérifier le rapport après une courte pause. Cela permet de gérer les cas où la création du groupe échoue temporairement en raison de problèmes de connexion ou d'autres erreurs temporaires.

    def center\_text(self, event):

        self**.**entry**.**tag\_configure("center", justify='center')

        self**.**entry**.**tag\_add("center", "1.0", "end")

La méthode center\_text permet de centrer le texte dans le widget Text. Elle configure le tag "center" pour justifier le texte au centre et l'applique à tout le contenu du widget Text. Cela assure que le texte saisi sera toujours centré verticalement et horizontalement dans le widget Text, quelles que soient les modifications apportées.

    def error(self):

        self**.**master**.**home()

        messagebox**.**showwarning("Erreur 40", "Une erreur s'est produite lors de

la lecture du fichier de données des questions. Le fichier est peut

être mal écrit, contient des erreurs ou est vide.")

La méthode error est appelée en cas d'erreur lors de la lecture du fichier de données des questions. Elle ramène l'utilisateur à l'écran d'accueil et affiche une boîte de dialogue d'avertissement avec le message d'erreur. Cela permet d'informer l'utilisateur de l'erreur survenue et de lui donner des indications sur la manière de la résoudre.

**Menu Portail**

from tkinter import \*

from tkinter import font

from pathlib import Path

from PIL import Image, ImageTk

from widgets**.**Image import custom\_Image

from widgets**.**Button import custom\_Button

from other**.**firebase**.**firestore import connexionPortail

from other**.**json**.**JsonFile import readJsonFile

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ces lignes importent les modules essentiels pour l'interface graphique avec Tkinter, notamment pour les polices de caractères et les images avec PIL. Elles importent également des classes personnalisées pour les widgets d'image et de bouton, ainsi que des modules pour la gestion de Firebase et de fichiers JSON. Enfin, elles définissent le chemin absolu du répertoire parent du fichier en cours d'exécution.

class displayPortail(Frame):

    def \_\_init\_\_(self, master):

        super()**.**\_\_init\_\_(master)

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**grid\_rowconfigure(0, weight=1)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=4)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**grid\_columnconfigure(1, weight=1)

        fileData = readJsonFile(paths / "../data/data.json")**.**get()

        try:

            if len(fileData["score"]) > 0:

                bestScore = 0

                for i in range(len(fileData["score"])):

                    if bestScore <=  fileData["score"][i]["score"]:

                        bestScore = fileData["score"][i]["score"]

                        try:

                            self**.**userName = fileData["score"][i]["name"]

                        except:

                            self**.**userName = "Inconnue"

                        self**.**userScore = fileData["score"][i]["score"]

            else:

                self**.**userName = "Inconnue"

                self**.**userScore = 0

        except:

            self**.**userName = "Inconnue"

            self**.**userScore = 0

        self**.**addComponents()

Cette classe displayPortail hérite de Frame de Tkinter pour créer l'écran du menu permettant de jouer avec le portail. Dans son constructeur \_\_init\_\_(), elle configure le fond de l'écran et les options de la grille pour l'organisation des composants. Elle charge également les données du meilleur score à partir d'un fichier JSON. Si aucun score n'est disponible, elle définit un score et un nom d'utilisateur par défaut. Ensuite, elle appelle la méthode addComponents() pour ajouter les composants à l'écran du menu.

def addComponents(self):

        custom\_Image(self, image=paths / "../assets/Background.png",

                     bg=self**.**master**.**color\_background,

                     width=700, height=700,

                     column=0, columnspan=2, row=0, rowspan=4)

        self**.**navbar = Frame(self)

        self**.**navbar**.**grid(column=0, columnspan=2, row=0)

        custom\_Image(self**.**navbar, image=paths /

"../assets/portail/Header\_Portail.png",

                     bg=self**.**master**.**color\_background,

                     width=571, height=82,

                     column=0, columnspan=2, row=0)

        custom\_Button(self**.**navbar, image=paths / "../assets/Home.png",

                      command=self**.**master**.**startGame,

                      bg=self**.**master**.**color\_second,

                      width=55, height=55,

                      column=1, row=0, sticky=E,

                      padx=(0, 20))

        self**.**score = Frame(self)

        self**.**score**.**grid(column=0, row=1, columnspan=2, sticky=S)

        self**.**score**.**grid\_columnconfigure(0, weight=1)

        self**.**score**.**grid\_columnconfigure(1, weight=1)

        self**.**score**.**grid\_columnconfigure(2, weight=15)

        custom\_Image(self**.**score, image=paths / "../assets/Frame1.png",

                     bg=self**.**master**.**color\_background,

                     height=84, width=571,

                     column=0, row=0, columnspan=3)

        custom\_Image(self**.**score, image=paths / "../assets/solo/Trophy.png",

                     bg=self**.**master**.**color\_second,

                     width=50, height=50,

                     column=0, row=0,

                     sticky=W, padx=(20, 0))

        fontStyle = font**.**Font(size=30, weight="bold")

        self**.**name = Label(self**.**score, text=self**.**userName, font=fontStyle,

                          bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text)

        self**.**name**.**grid(column=1, row=0)

        fontStyle = font**.**Font(size=30, weight="bold")

        self**.**percentage = Label(self**.**score, text=str(self**.**userScore) + "%",

font=fontStyle,

                                bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text)

        self**.**percentage**.**grid(column=2, row=0, sticky=E, padx=(0, 20))

        self**.**frame = Frame(self)

        self**.**frame**.**grid(column=0, columnspan=2, row=2)

        custom\_Image(self**.**frame, image=paths / "../assets/Frame3.png",

                     bg=self**.**master**.**color\_background,

                     height=274, width=571,

                     column=0, row=0, rowspan=3)

        fontStyle = font**.**Font(size=25, weight="bold")

        self**.**text = Label(self**.**frame, text="Saisir le code au démarrage du

portail pour te connecter", wraplength=487,

                          bg=self**.**master**.**color\_second,

fg=self**.**master**.**color\_text, font=fontStyle)

        self**.**text**.**grid(column=0, row=0, pady=0)

        fontStyle = font**.**Font(size=27, weight="bold")

        self**.**entry = Text(self**.**frame, bg=self**.**master**.**color\_fourth,

                          width=20, height=1,

                          font=fontStyle, fg=self**.**master**.**color\_text,

insertbackground=self**.**master**.**color\_text,

                          highlightthickness=4, highlightbackground="white",

highlightcolor="white")

        self**.**entry**.**grid(column=0, row=1, ipadx=5)

        self**.**center\_text(None)

        self**.**entry**.**bind("<KeyRelease>", self**.**center\_text)

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../assets/portail/Connexion.png")**.**resize((303,

51), Image**.**LANCZOS)

        )

        self**.**button\_connexion = Button(self**.**frame, command=lambda:

self**.**connexion(self**.**entry**.**get(1.0,

END)**.**replace('\n', '')), image=photo,

                                bg=self**.**master**.**color\_second,

                                cursor="hand2", compound=CENTER,

                                bd=0, highlightthickness=0,

highlightbackground="white",

                                activebackground=self**.**master**.**color\_second)

        self**.**button\_connexion**.**image = photo

        self**.**button\_connexion**.**grid(column=0, row=2, ipadx=5, ipady=2)

Elle configure également les boutons et les labels avec les chemins d'accès aux images appropriés et les paramètres de style des éléments d'interface. La méthode utilise également la classe `custom\_Image` et `custom\_Button` pour créer des widgets d'image et de bouton personnalisés, respectivement.

    def center\_text(self, event):

        self**.**entry**.**tag\_configure("center", justify='center')

        self**.**entry**.**tag\_add("center", "1.0", "end")

Cette méthode `center\_text(self, event)` est utilisée pour centrer le texte dans le widget de saisie de texte (`Text`). Elle configure un tag appelé "center" pour justifier le texte au centre et l'applique à tout le texte dans le widget, ce qui le centre visuellement.

    def connexion(self, token):

        self**.**join\_group\_connexion = connexionPortail(token)

        if self**.**join\_group\_connexion**.**report:

            self**.**master**.**startQuizPortail(token)

Cette méthode `connexion(self, token)` est utilisée pour gérer la connexion au portail en utilisant le token fourni en paramètre. Elle appelle la fonction `connexionPortail(token)` pour établir la connexion. Si la connexion est réussie (c'est-à-dire si `self.join\_group\_connexion.report` est vrai), elle démarre le quiz du portail en appelant la méthode `startQuizPortail(token)` du maître.

**Jeux Solo**

from tkinter import messagebox

from pathlib import Path

import random

from logzero import logger

from other**.**json**.**JsonFile import readJsonFileSchema

from display**.**quiz**.**choice1 import displayChoice1

from display**.**quiz**.**choice2 import displayChoice2

from display**.**quiz**.**click1 import displayClick1

from display**.**quiz**.**DragAndDrop1 import displayDragAndDrop1

from display**.**quiz**.**DragAndDrop2 import displayDragAndDrop2

from display**.**quiz**.**DragAndDrop3 import displayDragAndDrop3

from display**.**quiz**.**audio1 import displayAudio

from display**.**score import displayScore

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ces lignes importent les modules essentiels pour l'interface graphique avec Tkinter, notamment pour les boîtes de dialogue avec `messagebox`. Elles importent également des modules pour la gestion des chemins de fichiers avec `Path`, la génération de nombres aléatoires avec `random`, et la journalisation avec `logzero`. De plus, elles importent des classes spécifiques pour l'affichage des différentes questions du quiz, ainsi que la classe pour l'affichage du score. Enfin, elles définissent le chemin absolu du répertoire parent du fichier en cours d'exécution.

class solo:

    def \_\_init\_\_(self, master, numberQuestion):

        self**.**numberQuestion = numberQuestion

        self**.**master = master

        self**.**playerScore = 0

        self**.**errorQuestion = []

        self**.**readFile = readJsonFileSchema(paths /

'../../data/question.json')**.**get()

        if self**.**readFile == []:

            self**.**error()

        self**.**start()

        self**.**play()

Cette classe `solo` gère le déroulement de la partie en solo du jeu. Dans son constructeur `\_\_init\_\_()`, elle prend en paramètre le maître (probablement la fenêtre principale de l'application) et le nombre de questions pour la partie. Elle initialise également les scores du joueur et une liste pour enregistrer les questions erronées. Ensuite, elle lit le fichier JSON contenant les questions à partir du chemin spécifié et vérifie s'il est vide. Si c'est le cas, elle appelle la méthode `error()` pour gérer l'erreur. Enfin, elle démarre le jeu en appelant les méthodes `start()` et `play()`.

    def start(self):

        logger**.**info("Play Solo Game")

        self**.**listeType = []

        for data in self**.**readFile:

            self**.**listeType**.**append(data["type"])

        self**.**randomList = random**.**sample(range(0, len(self**.**listeType)),

min(self**.**numberQuestion, len(self**.**listeType)))

        self**.**currentQuestionIndex = 0

Cette méthode start(self) initialise le jeu solo en enregistrant les types de questions disponibles dans une liste. Ensuite, elle crée une liste d'indices aléatoires pour sélectionner les questions à partir du fichier JSON. La taille de cette liste est déterminée par le nombre de questions demandées ou le nombre total de questions disponibles, selon le plus petit des deux. Enfin, elle initialise l'indice de la question actuelle à zéro.

def play(self):

        function = {

            "choice1": self**.**Choice1,

            "choice2": self**.**Choice2,

            "click1": self**.**Click1,

            "draganddrop1": self**.**DragAndDrop1,

            "draganddrop2": self**.**DragAndDrop2,

            "draganddrop3": self**.**DragAndDrop3,

            "audio1": self**.**Audio1

        }

        if self**.**currentQuestionIndex > 0:

            self**.**playerScore += self**.**display**.**get()

            if self**.**display**.**get() <= 0:

                self**.**errorQuestion**.**append(self**.**readFile[self**.**randomList[

self**.**currentQuestionIndex - 1]]["question"])

        if self**.**currentQuestionIndex < len(self**.**randomList):

            question\_type =

self**.**listeType[self**.**randomList[self**.**currentQuestionIndex]]

            if question\_type in function:

                question\_class = function[question\_type]

                question\_class()

            else:

                self**.**error()

            self**.**currentQuestionIndex += 1

        else:

            logger**.**info(f"End quiz, score player: {self**.**playerScore}

({int((self**.**playerScore / len(self**.**randomList)) \* 100)} %)")

            displayScore(self**.**master, int((self**.**playerScore /

len(self**.**randomList)) \* 100), self**.**errorQuestion)

Cette méthode `play(self)` gère le déroulement du jeu solo. Elle utilise un dictionnaire pour mapper chaque type de question à sa méthode correspondante dans la classe. À chaque itération, elle vérifie si une question précédente a été répondue, met à jour le score du joueur et enregistre les questions erronées. Ensuite, elle récupère le type de question et instancie la classe correspondante. Une fois toutes les questions posées, elle enregistre le score final du joueur et affiche l'écran du score avec le pourcentage de réussite et la liste des questions erronées.

    def Choice1(self):

        self**.**display = displayChoice1(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Choice2(self):

        self**.**display = displayChoice2(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Click1(self):

        self**.**display = displayClick1(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        cursorStyle=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]][

"cursor"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def DragAndDrop1(self):

        self**.**display = displayDragAndDrop1(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def DragAndDrop2(self):

        self**.**display = displayDragAndDrop2(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["zones"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def DragAndDrop3(self):

        self**.**display = displayDragAndDrop3(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["zones"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Audio1(self):

        self**.**display = displayAudio(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["sound"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

Ces méthodes `Choice1`, `Choice2`, `Click1`, `DragAndDrop1`, `DragAndDrop2`, `DragAndDrop3` et `Audio1` sont utilisées pour afficher les différentes interfaces du quiz en fonction du type de question. Chaque méthode instancie la classe correspondante pour afficher l'interface de la question spécifique sur la fenêtre principale. Les arguments passés à chaque méthode incluent les informations nécessaires pour afficher la question, telles que le texte de la question, les choix possibles, les réponses correctes, le temps imparti, etc. Les méthodes prennent également en compte le numéro de la question actuelle et le nombre total de questions pour les afficher de manière appropriée.

    def error(self):

        logger**.**error("Erreur 10")

        self**.**master**.**home()

        messagebox**.**showwarning("Erreur de lecture du fichier de données des

questions",

                               "Une erreur s'est produite lors de la lecture du fichier de données des questions. Le fichier est peut être mal écrit, contient des erreurs ou est vide.")

Cette méthode `error(self)` est appelée lorsqu'une erreur se produit lors de la lecture du fichier de données des questions. Elle enregistre un message d'erreur dans les logs, renvoie l'utilisateur à l'écran d'accueil et affiche une boîte de dialogue d'avertissement indiquant que le fichier de données des questions est mal écrit, contient des erreurs ou est vide.

**Jeux Duo**

from tkinter import messagebox

from pathlib import Path

from logzero import logger

from other**.**firebase**.**firestore import userPoints, endGame

from display**.**quiz**.**choice1 import displayChoice1

from display**.**quiz**.**choice2 import displayChoice2

from display**.**quiz**.**click1 import displayClick1

from display**.**quiz**.**DragAndDrop1 import displayDragAndDrop1

from display**.**quiz**.**DragAndDrop2 import displayDragAndDrop2

from display**.**quiz**.**DragAndDrop3 import displayDragAndDrop3

from display**.**quiz**.**audio1 import displayAudio

from display**.**scoreDuo import displayScoreDuo *# type: ignore*

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ces lignes importent les modules essentiels pour l'interface graphique avec Tkinter, ainsi que pour la gestion des chemins de fichiers avec `Path` et la journalisation avec `logzero`. Elles importent également des modules pour l'interaction avec Firebase, en particulier pour la mise à jour des points utilisateur et la fin de la partie. En outre, elles importent des classes spécifiques pour l'affichage des différentes questions du quiz et la classe pour l'affichage du score en mode duo. Enfin, elles définissent le chemin absolu du répertoire parent du fichier en cours d'exécution.

class duo:

    def \_\_init\_\_(self, master, numberQuestion, user, token):

        self**.**numberQuestion = numberQuestion

        self**.**master = master

        self**.**user = user

        self**.**token = token

        self**.**readFile = self**.**master**.**question

        self**.**playerScore = 0

        if self**.**readFile == []:

            self**.**error()

        self**.**userPoints = userPoints(self**.**user, self**.**token)

        self**.**start()

        self**.**play()

Cette classe `duo` gère le déroulement de la partie à deux joueurs. Dans son constructeur `\_\_init\_\_()`, elle prend en paramètre le maître (probablement la fenêtre principale de l'application), le nombre de questions pour la partie, l'utilisateur actuel et le token d'authentification. Elle initialise également les scores du joueur et récupère les questions à partir du maître. Si aucune question n'est disponible, elle appelle la méthode `error()` pour gérer l'erreur. Ensuite, elle initialise l'interaction avec Firebase pour mettre à jour les points de l'utilisateur. Enfin, elle démarre le jeu en appelant les méthodes `start()` et `play()`.

    def start(self):

        logger**.**info("Play Duo Game")

        self**.**listeType = []

        for data in self**.**readFile:

            self**.**listeType**.**append(data["type"])

        self**.**randomList = self**.**master**.**listQuestion

        self**.**currentQuestionIndex = 0

Cette méthode `start(self)` initialise le jeu en duo en enregistrant les types de questions disponibles dans une liste. Ensuite, elle récupère la liste aléatoire des questions à partir du maître et initialise l'indice de la question actuelle à zéro.

def play(self):

        function = {

            "choice1": self**.**Choice1,

            "choice2": self**.**Choice2,

            "click1": self**.**Click1,

            "draganddrop1": self**.**DragAndDrop1,

            "draganddrop2": self**.**DragAndDrop2,

            "draganddrop3": self**.**DragAndDrop3,

            "audio1": self**.**Audio1

        }

        if self**.**currentQuestionIndex > 0:

            self**.**userPoints**.**set(self**.**display**.**get())

            self**.**playerScore += self**.**display**.**get()

        if self**.**currentQuestionIndex < len(self**.**randomList):

            question\_type =

self**.**listeType[self**.**randomList[self**.**currentQuestionIndex]]

            if question\_type in function:

                question\_class = function[question\_type]

                question\_class()

            else:

                self**.**error()

            self**.**currentQuestionIndex += 1

        else:

            endGame(self**.**token, self**.**user)

            logger**.**info(f"End quiz, score player: {self**.**playerScore}

({int((self**.**playerScore / len(self**.**randomList)) \* 100)} %)")

            displayScoreDuo(self**.**master, len(self**.**randomList), style=self**.**user

+ 1, user=self**.**user, token=self**.**token)

Cette méthode `play(self)` gère le déroulement du jeu en duo. Elle met à jour les points de l'utilisateur après chaque question et ajoute les points au score du joueur. Ensuite, elle affiche la prochaine question si elle existe. Une fois toutes les questions posées, elle met fin à la partie et affiche l'écran du score duo.

def Choice1(self):

        self**.**display = displayChoice1(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Choice2(self):

        self**.**display = displayChoice2(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Click1(self):

        self**.**display = displayClick1(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        cursorStyle=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]][

"cursor"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def DragAndDrop1(self):

        self**.**display = displayDragAndDrop1(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def DragAndDrop2(self):

        self**.**display = displayDragAndDrop2(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["zones"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def DragAndDrop3(self):

        self**.**display = displayDragAndDrop3(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["zones"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Audio1(self):

        self**.**display = displayAudio(self**.**master, self**.**play,

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["sound"],

        time=self**.**readFile[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=self**.**user + 1)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

Ces méthodes `Choice1()`, `Choice2()`, `Click1()`, `DragAndDrop1()`, `DragAndDrop2()`, `DragAndDrop3()` et `Audio1()` créent et affichent les différentes interfaces pour les types de questions spécifiques dans le jeu en duo. Chaque méthode utilise les données spécifiques de la question actuelle pour afficher la question correcte et configurer l'interface utilisateur en fonction.

    def error(self):

        logger**.**error("Erreur 20")

        self**.**master**.**home()

        messagebox**.**showwarning("Erreur 20",

                               "Une erreur s'est produite lors de la lecture du fichier de données des questions. Le fichier est peut être mal écrit, contient des erreurs ou est vide.")

Cette méthode error(self) est appelée lorsque survient une erreur lors de la lecture du fichier de données des questions. Elle enregistre l'erreur dans les logs, renvoie l'utilisateur à l'écran d'accueil et affiche une boîte de dialogue indiquant l'erreur.

**Jeux Portail**

from tkinter import messagebox

from pathlib import Path

import random

from logzero import logger

from other**.**json**.**JsonFile import readJsonFileSchema

from other**.**firebase**.**firestore import storageQuestionPortail

from display**.**quiz**.**choice1 import displayChoice1

from display**.**quiz**.**choice2 import displayChoice2

from display**.**score import displayScore

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Il semble que vous importiez plusieurs modules et classes pour la gestion des questions dans votre jeu. Ce bloc de code semble s'occuper de la lecture des questions depuis un fichier JSON, de leur stockage dans Firestore, et de l'affichage des questions dans le jeu. Cela semble être une partie essentielle du fonctionnement de votre application de jeu.

class portail:

    def \_\_init\_\_(self, master, numberQuestion, token):

        self**.**numberQuestion = numberQuestion

        self**.**master = master

        self**.**token = token

        self**.**playerScore = 0

        self**.**errorQuestion = []

        self**.**readFile = readJsonFileSchema(paths /

'../../data/question.json')**.**get()

        if self**.**readFile == []:

            self**.**error()

        self**.**start()

        self**.**play()

Cette classe `portail` semble gérer le déroulement de la partie avec le portail dans votre jeu. Elle initialise les paramètres nécessaires tels que le nombre de questions, le jeton d'authentification, et le score du joueur. Ensuite, elle charge les questions depuis un fichier JSON et démarre le jeu. En cas d'erreur lors de la lecture du fichier, elle gère cette situation en affichant un message d'erreur. C'est une partie cruciale de votre application qui s'assure que le jeu fonctionne correctement avec le portail.

    def start(self):

        logger**.**info("Play Portail Game")

        self**.**listeType, self**.**readQuestion, self**.**question = [], [], []

        for data in self**.**readFile:

            if data["type"] == "choice1" or data["type"] == "choice2":

                self**.**readQuestion**.**append(data)

                self**.**listeType**.**append(data["type"])

        self**.**randomList = random**.**sample(range(0, len(self**.**listeType)),

min(self**.**numberQuestion, len(self**.**listeType)))

        self**.**currentQuestionIndex = 0

        for i in range(len(self**.**randomList)):

            self**.**question**.**append(self**.**readQuestion[self**.**randomList[i]])

        self**.**randomList = random**.**sample(range(0, len(self**.**question)),

min(self**.**numberQuestion, len(self**.**question)))

        storageQuestionPortail(self**.**token, self**.**question, self**.**randomList, [0]

\* min(len(self**.**randomList), 10))

Cette méthode `start(self)` initialise le jeu sur le portail. Elle sélectionne aléatoirement un nombre spécifié de questions parmi celles disponibles dans le fichier de données. Ensuite, elle stocke ces questions dans le portail.

def play(self):

        function = {

            "choice1": self**.**Choice1,

            "choice2": self**.**Choice2

        }

        if self**.**currentQuestionIndex > 0:

            self**.**playerScore += self**.**display**.**get()

            if self**.**display**.**get() <= 0:

                self**.**errorQuestion**.**append(self**.**readFile[self**.**randomList[

self**.**currentQuestionIndex - 1]]["question"])

        if self**.**currentQuestionIndex < len(self**.**randomList):

            question\_type =

self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["type"]

            if question\_type in function:

                question\_class = function[question\_type]

                question\_class()

            else:

                self**.**error()

            self**.**currentQuestionIndex += 1

        else:

            logger**.**info(f"End quiz, score player: {self**.**playerScore}

({int((self**.**playerScore / len(self**.**randomList)) \* 100)} %)")

            displayScore(self**.**master, int((self**.**playerScore /

len(self**.**randomList)) \* 100), self**.**errorQuestion)

Cette fonction `play(self)` gère le déroulement du jeu sur le portail. Elle affiche les différentes questions et leurs types à mesure que le jeu progresse. Une fois que toutes les questions ont été répondues, elle affiche le score final du joueur.

    def Choice1(self):

        self**.**display = displayChoice1(self**.**master, self**.**play,

        self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["choices"],

        self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=4, token=self**.**token)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

    def Choice2(self):

        self**.**display = displayChoice2(self**.**master, self**.**play,

        self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["question"],

        self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["answer"],

        time=self**.**question[self**.**randomList[self**.**currentQuestionIndex]]["time",

        currentQuestion=self**.**currentQuestionIndex + 1,

        maxQuestion=len(self**.**randomList),

        style=4, token=self**.**token)

        self**.**display**.**grid(row=0, column=0, sticky="nsew")

Ces deux méthodes, `Choice1(self)` et `Choice2(self)`, sont utilisées pour afficher les questions de type "choice1" et "choice2" respectivement sur le portail. Elles utilisent les informations stockées dans `self.question` pour récupérer les détails de la question actuelle, telles que la question elle-même, les choix possibles et la réponse correcte, puis affichent la question sur l'interface graphique. Une fois que le joueur a répondu à la question, il est dirigé vers la méthode `self.play()` pour passer à la question suivante.

    def error(self):

        logger**.**error("Erreur 30")

        self**.**master**.**home()

        messagebox**.**showwarning("Erreur 30",

                               "Une erreur s'est produite lors de la lecture du fichier de données des questions. Le fichier est peut être mal écrit, contient des erreurs ou est vide.")

La méthode `error(self)` est appelée en cas d'erreur lors de la lecture du fichier de données des questions sur le portail. Elle enregistre l'erreur dans les logs, renvoie l'utilisateur à l'écran d'accueil et affiche une boîte de dialogue d'avertissement avec un message approprié indiquant qu'une erreur s'est produite lors de la lecture du fichier de données des questions.

**Interface Quiz - Audio**

from tkinter import \*

from tkinter import font, messagebox

import pygame

from logzero import logger

from PIL import Image, ImageTk, ImageDraw

from pathlib import Path

from widgets**.**Button import custom\_Button

from widgets**.**Image import custom\_Image

from other**.**chrono import ChronoApp

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Le code importe plusieurs modules nécessaires à la création de l'interface graphique du jeu. Il utilise Tkinter pour la création des widgets, notamment les boutons et les images, ainsi que pour la gestion des polices. Il importe également les modules pygame et PIL pour le traitement des images et le module logzero pour la journalisation des événements. La classe `ChronoApp` est également importée à partir d'un fichier externe, ce qui semble indiquer qu'elle est utilisée pour gérer le chronomètre du jeu.

class displayAudio(Frame):

    def \_\_init\_\_(self, master, callback, textQuestion, textResponse,

correctResponse, audioFile,

                 style=1, time=60, currentQuestion = 0, maxQuestion=20):

        super()**.**\_\_init\_\_(master)

        self**.**callback = callback

        self**.**style = style

        self**.**time = time

        self**.**textQuestion = textQuestion

        self**.**textResponse = textResponse

        self**.**correctResponse = correctResponse

        self**.**audioFile = paths / "../../data/" / audioFile

        self**.**questionNumber = f"{currentQuestion}/{maxQuestion}"

        self**.**questionNumberSelect = 0

        self**.**points = 0

        for content in self**.**master**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**grid\_rowconfigure(0, weight=3)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=3)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**addComponents()

Cette classe, `displayAudio`, définit l'interface graphique pour la première partie du quiz, qui consiste à retrouver le sens du son. Elle prend en paramètres le maître (master) de la fenêtre, une fonction de rappel (callback), le texte de la question, le texte de la réponse, la réponse correcte, le fichier audio, le style (par défaut à 1), le temps (par défaut à 60 secondes), le numéro de la question actuelle, et le nombre maximum de questions. Elle initialise les attributs de la classe et configure les éléments de l'interface, puis ajoute les composants graphiques nécessaires.

def addComponents(self):

        if self**.**style == 2:

            background\_source = paths / "../../assets/Background-red.png"

            self**.**master**.**color\_background = "#CF6953"

        elif self**.**style == 3:

            background\_source = paths / "../../assets/Background-blue.png"

            self**.**master**.**color\_background = "#53B1CF"

        else:

            background\_source = paths / "../../assets/Background.png"

        custom\_Image(self, image=background\_source,

bg=self**.**master**.**color\_background,

                     width=700, height=700,

                     column=0, row=0, rowspan=3)

        self**.**question = Frame(self, bg=self**.**master**.**color\_background)

        self**.**question**.**grid(column=0, row=0)

        fontStyle = font**.**Font(size=15)

        custom\_Image(self**.**question, image=paths / "../../assets/Frame5.png",

                     text=self**.**textQuestion,

                     fg=self**.**master**.**color\_text, font=fontStyle,

wraplength=600,

                     bg=self**.**master**.**color\_background,

                     width=625, height=100,

                     column=0, row=0)

        fontStyle = font**.**Font(size=15, weight="bold")

        self**.**header = Label(self**.**question, compound="center", font=fontStyle,

fg=self**.**master**.**color\_text, bg=self**.**master**.**color\_background)

        self**.**header**.**grid(column=0, row=1, pady=(7, 0))

        self**.**body = Frame(self, bg=self**.**master**.**color\_background, height=325,

width=620)

        self**.**body**.**grid(column=0, row=1)

        self**.**audio = Frame(self**.**body, bg=self**.**master**.**color\_second,

                           width=622, height=55,

                           highlightthickness=4, highlightbackground="white",

highlightcolor="white")

        self**.**audio**.**grid(column=0, row=0, pady=(10, 20))

        self**.**audio**.**grid\_columnconfigure(0, weight=1)

        self**.**audio**.**grid\_columnconfigure(1, weight=1)

        self**.**audio**.**grid\_columnconfigure(2, weight=1)

        self**.**audio**.**grid\_columnconfigure(3, weight=1)

        fontStyle = font**.**Font(size=15, weight="bold")

        self**.**audioTime = Label(self**.**audio, text="00:00", font=fontStyle,

fg=self**.**master**.**color\_text, bg=self**.**master**.**color\_second)

        self**.**audioTime**.**grid(column=0, row=0, padx=(10, 0))

        image = self**.**timeBar(0, 10)

        photo = ImageTk**.**PhotoImage(image)

        self**.**audioTimeBar = Label(self**.**audio, image=photo,

bg=self**.**master**.**color\_second)

        self**.**audioTimeBar**.**image = photo

        self**.**audioTimeBar**.**grid(column=1, row=0, padx=10, pady=15)

        self**.**buttonPlayStat = "Start"

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../../assets/quiz/Pause.png")**.**resize((35, 35),

Image**.**LANCZOS)

        )

        self**.**buttonPlay = Button(self**.**audio, command=self**.**play\_pause,

image=photo,

                        bg=self**.**master**.**color\_second,

                        cursor="hand2", bd=0, highlightthickness=0,

highlightbackground="white",

                        activebackground=self**.**master**.**color\_second)

        self**.**buttonPlay**.**image = photo

        self**.**buttonPlay**.**grid(row=0, column=2, padx=10)

        self**.**button\_borders = []

        for i in range(3):

            self**.**createButton(i + 1, self**.**textResponse[i], i + 1)

        custom\_Button(self,

                        command=self**.**validate,

                        image=paths / "../../assets/quiz/Valider.png",

                        height=75, width=343,

                        bg=self**.**master**.**color\_background,

                        column=0, row=2, ipadx=5, ipady=2)

        fontStyle = font**.**Font(size=25, weight="bold")

        self**.**numberQuestion = Label(self, text=self**.**questionNumber,

compound="center", font=fontStyle, fg=self**.**master**.**color\_text2,

bg=self**.**master**.**color\_background)

        self**.**numberQuestion**.**grid(column=0, row=2, sticky=SE, padx=20, pady=20)

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../../assets/Frame6.png")**.**resize((250, 40),

Image**.**LANCZOS)

        )

        self**.**header**.**config(image=photo)

        self**.**header**.**image = photo

        self**.**chrono = ChronoApp(self**.**master, self, self**.**header, self**.**time)

Ce code crée les composants visuels de l'interface utilisateur pour la première partie du quiz, avec comme objectif de retrouver le sens du son. Il configure le style de la fenêtre en fonction d'un paramètre donné, définit la question, le corps de la question, et les éléments audio associés. Enfin, il ajoute des boutons pour les réponses, un bouton de validation, et un numéro de question.

    def createButton(self, row, text, button\_number):

        buttonBorder = Frame(self**.**body, bg="white")

        buttonBorder**.**grid(column=0, row=row, pady=7)

        fontStyle = font**.**Font(size=15, weight="bold")

        button = Button(buttonBorder, text=text, font=fontStyle,

justify='right',

                           width=50, height=2, cursor="hand2",

                           fg=self**.**master**.**color\_text,                            bg=self**.**master**.**color\_fourth, bd=0,

                           highlightthickness=4, highlightbackground="white",

highlightcolor="white",

                           activebackground=self**.**master**.**color\_fourth,

activeforeground=self**.**master**.**color\_text,

                           command=lambda:

self**.**changeBorderColor(buttonBorder, button\_number))

        button**.**grid(column=0, row=0, padx=5, pady=5)

        self**.**button\_borders**.**append(buttonBorder)

    def changeBorderColor(self, selected\_border, button\_number):

        for border in self**.**button\_borders:

            border**.**config(bg="white")

        selected\_border**.**config(bg="#114232")

        self**.**questionNumberSelect = button\_number

Cette partie de code définit deux méthodes pour la classe `displayAudio`. La première, `createButton`, est utilisée pour créer des boutons de réponse dans l'interface graphique. Elle prend en paramètres le numéro de ligne, le texte du bouton, et le numéro du bouton. La deuxième méthode, `changeBorderColor`, est appelée lorsqu'un bouton est cliqué. Elle change la couleur du cadre entourant le bouton cliqué pour indiquer la sélection.

    def play\_pause(self):

        if self**.**buttonPlayStat == "Start":

            self**.**buttonPlayStat = "Play"

            self**.**start\_sound()

            photo = ImageTk**.**PhotoImage(

                Image**.**open(paths / "../../assets/quiz/Play.png")**.**resize((35,

35), Image**.**LANCZOS)

            )

            self**.**buttonPlay**.**config(image=photo)

            self**.**buttonPlay**.**image = photo

            self**.**update\_progress()

        elif self**.**buttonPlayStat == "Play":

            self**.**buttonPlayStat = "Pause"

            pygame**.**mixer**.**music**.**pause()

            photo = ImageTk**.**PhotoImage(

                Image**.**open(paths / "../../assets/quiz/Pause.png")**.**resize((35,

35), Image**.**LANCZOS)

            )

            self**.**buttonPlay**.**config(image=photo)

            self**.**buttonPlay**.**image = photo

        else:

            self**.**buttonPlayStat = "Play"

            pygame**.**mixer**.**music**.**unpause()

            self**.**update\_progress()

            photo = ImageTk**.**PhotoImage(

                Image**.**open(paths / "../../assets/quiz/Play.png")**.**resize((35,

35), Image**.**LANCZOS)

            )

            self**.**buttonPlay**.**config(image=photo)

            self**.**buttonPlay**.**image = photo

    def start\_sound(self):

        try:

            pygame**.**mixer**.**init()

            self**.**totalTimeSound =

pygame**.**mixer**.**Sound(self**.**audioFile)**.**get\_length()

            pygame**.**mixer**.**music**.**load(self**.**audioFile)

            pygame**.**mixer**.**music**.**set\_volume(.5)

            pygame**.**mixer**.**music**.**play(loops=-1)

        except pygame**.**error as \_:

            self**.**error()

    def update\_progress(self):

        try:

            self**.**current\_time = int(pygame**.**mixer**.**music**.**get\_pos() // 1000 %

self**.**totalTimeSound)

            minutes = self**.**current\_time // 60

            seconds = self**.**current\_time % 60

            self**.**audioTime**.**config(text=f"{minutes:02}:{seconds:02}")

            if pygame**.**mixer**.**music**.**get\_busy():

                self**.**master**.**after(1000, self**.**update\_progress)

            image = self**.**timeBar(self**.**current\_time, self**.**totalTimeSound)

            photo = ImageTk**.**PhotoImage(image)

            self**.**audioTimeBar**.**config(image=photo)

            self**.**audioTimeBar**.**image = photo

        except pygame**.**error as e:

            self**.**error()

    def timeBar(self, current, total):

        image = Image**.**new("RGBA", (420, 10), "#ffffff00")

        draw = ImageDraw**.**Draw(image)

        time = (current / total) \* 100

        if time < 10:

            time = 10

        rect2\_coords = (0, 0, (time / 100) \* 420, 10)

        def draw\_rounded\_rectangle(draw, coords, radius, fill):

            x1, y1, x2, y2 = coords

            draw**.**rectangle([x1 + radius, y1, x2 - radius, y2], fill=fill)

            draw**.**rectangle([x1, y1 + radius, x2, y2 - radius], fill=fill)

            draw**.**pieslice([x1, y1, x1 + 2 \* radius, y1 + 2 \* radius], 180,

270, fill=fill)

            draw**.**pieslice([x2 - 2 \* radius, y1, x2, y1 + 2 \* radius], 270,

360, fill=fill)

            draw**.**pieslice([x1, y2 - 2 \* radius, x1 + 2 \* radius, y2], 90, 180,

fill=fill)

            draw**.**pieslice([x2 - 2 \* radius, y2 - 2 \* radius, x2, y2], 0, 90,

fill=fill)

        draw\_rounded\_rectangle(draw, (0, 0, 420, 10), 5, fill="#747977")

        draw\_rounded\_rectangle(draw, rect2\_coords, 5, fill="#FF3D3D")

        return image

Cette partie de code définit plusieurs méthodes pour la classe `displayAudio`. La première, `play\_pause`, permet de mettre en pause ou de reprendre la lecture du son en fonction de l'état actuel du bouton. La méthode `start\_sound` est utilisée pour charger et démarrer le son. La méthode `update\_progress` est appelée périodiquement pour mettre à jour la barre de progression de lecture du son. Enfin, la méthode `timeBar` crée une image représentant la barre de progression de lecture du son.

    def validate(self):

        self**.**chrono**.**stop\_timer()

        if not(pygame**.**mixer**.**get\_init() is None):

            pygame**.**mixer**.**music**.**pause()

        if self**.**questionNumberSelect == self**.**correctResponse:

            self**.**points = 1

        if self**.**callback:

            self**.**callback()

Cette méthode, "validate", est utilisée pour valider et corriger la réponse donnée par l'utilisateur. Elle arrête également le chronomètre en cours. Si l'initialisation de la bibliothèque de sons pygame a été effectuée, elle met en pause la lecture du son en cours. Ensuite, elle vérifie si la réponse sélectionnée par l'utilisateur correspond à la réponse correcte, et attribue un point si c'est le cas. Enfin, elle appelle la fonction de rappel (callback) associée à cette action, le cas échéant.

    def get(self):

        return self**.**points

Cette méthode, "get", retourne le nombre de points accumulés jusqu'à présent dans le jeu. Cela permet d'accéder aux points depuis d'autres parties du programme.

    def error(self):

        logger**.**error("Erreur 50")

        self**.**master**.**home()

        messagebox**.**showwarning("Erreur 50", "Une erreur s'est produite lors de la lecture du fichier audio")

Cette méthode, "error", est appelée en cas d'erreur lors de la lecture du fichier audio. Elle enregistre un message d'erreur dans les logs, puis redirige l'utilisateur vers la page d'accueil de l'application et affiche une boîte de dialogue d'avertissement informant de l'erreur survenue.

**Interface Quiz – Choix 1**

from tkinter import \*

from tkinter import font

from PIL import Image, ImageTk

from pathlib import Path

from widgets**.**Button import custom\_Button

from widgets**.**Image import custom\_Image

from other**.**chrono import ChronoApp

from other**.**firebase**.**firestore import portailNotify

paths = Path(\_\_file\_\_)**.**parent**.**resolve()

Ce code importe les modules nécessaires, tels que Tkinter pour l'interface graphique, PIL pour la manipulation d'images, Path pour la gestion des chemins de fichiers, et des modules personnalisés pour les boutons et les images. Il importe également une classe `ChronoApp` pour la gestion du chronomètre et une fonction `portailNotify` pour les notifications via Firebase Firestore. Enfin, il définit le chemin absolu du fichier en cours d'exécution.

class displayChoice1(Frame):

    def \_\_init\_\_(self, master, callback, textQuestion, textResponse,

correctResponse,

                 style=1, time=60, currentQuestion = 0, maxQuestion=20,

token=""):

        super()**.**\_\_init\_\_(master)

        self**.**callback = callback

        self**.**style = style

        self**.**time = time

        self**.**textQuestion = textQuestion

        self**.**textResponse = textResponse

        self**.**correctResponse = correctResponse

        self**.**token = token

        self**.**currentQuestion = currentQuestion

        self**.**questionNumber = f"{currentQuestion}/{maxQuestion}"

        self**.**questionNumberSelect = 0

        self**.**points = 0

        for content in self**.**master**.**grid\_slaves():

            content**.**grid\_remove()

        self**.**config(bg=self**.**master**.**color\_background)

        self**.**grid(column=0, row=0, sticky="nsew")

        self**.**grid\_rowconfigure(0, weight=3)

        self**.**grid\_rowconfigure(1, weight=1)

        self**.**grid\_rowconfigure(2, weight=3)

        self**.**grid\_columnconfigure(0, weight=1)

        self**.**addComponents()

Cette classe, `displayChoice1`, définit l'interface graphique pour la deuxième partie du quiz, qui consiste à choisir la bonne réponse parmi quatre propositions. Elle prend en paramètres le maître (master) de la fenêtre, une fonction de rappel (callback), le texte de la question, le texte des réponses, la réponse correcte, le style (par défaut à 1), le temps (par défaut à 60 secondes), le numéro de la question actuelle, le nombre maximum de questions, et un token facultatif. Elle initialise les attributs de la classe et configure les éléments de l'interface, puis ajoute les composants graphiques nécessaires.

def addComponents(self):

        if self**.**style == 2:

            background\_source = paths / "../../assets/Background-red.png"

            self**.**master**.**color\_background = "#CF6953"

        elif self**.**style == 3:

            background\_source = paths / "../../assets/Background-blue.png"

            self**.**master**.**color\_background = "#53B1CF"

        else:

            background\_source = paths / "../../assets/Background.png"

        custom\_Image(self, image=background\_source, bg=self**.**master**.**color\_background,

                     width=700, height=700,

                     column=0, row=0, rowspan=3)

        self**.**question = Frame(self, bg=self**.**master**.**color\_background)

        self**.**question**.**grid(column=0, row=0)

        fontStyle = font**.**Font(size=15)

        custom\_Image(self**.**question, image=paths / "../../assets/Frame5.png",

                     text=self**.**textQuestion,

                     fg=self**.**master**.**color\_text, font=fontStyle, wraplength=600,

                     bg=self**.**master**.**color\_background,

                     width=625, height=100,

                     column=0, row=1)

        fontStyle = font**.**Font(size=15, weight="bold")

        self**.**header = Label(self**.**question, compound="center", font=fontStyle, fg=self**.**master**.**color\_text, bg=self**.**master**.**color\_background)

        self**.**header**.**grid(column=0, row=2, pady=(7, 0))

        self**.**body = Frame(self, bg=self**.**master**.**color\_background, height=325, width=620)

        self**.**body**.**grid(column=0, row=1)

        self**.**button\_borders = []

        for i in range(4):

            self**.**createButton(i, self**.**textResponse[i], i + 1)

        if self**.**style != 4:

            custom\_Button(self,

                            command=self**.**validate,

                            image=paths / "../../assets/quiz/Valider.png",

                            height=75, width=343,

                            bg=self**.**master**.**color\_background,

                            column=0, row=2, ipadx=5, ipady=2)

            fontStyle = font**.**Font(size=25, weight="bold")

            self**.**numberQuestion = Label(self, text=self**.**questionNumber, compound="center", font=fontStyle, fg=self**.**master**.**color\_text2, bg=self**.**master**.**color\_background)

            self**.**numberQuestion**.**grid(column=0, row=2, sticky=SE, padx=20, pady=20)

        elif self**.**style == 4:

            self**.**portail\_connexion = portailNotify(self**.**token, self**.**currentQuestion - 1)

            self**.**loopCreate = True

            self**.**selectWithPortail()

        photo = ImageTk**.**PhotoImage(

            Image**.**open(paths / "../../assets/Frame6.png")**.**resize((250, 40), Image**.**LANCZOS)

        )

        self**.**header**.**config(image=photo)

        self**.**header**.**image = photo

        if self**.**style != 4:

            self**.**chrono = ChronoApp(self**.**master, self, self**.**header, self**.**time)

        elif self**.**style == 4:

            self**.**header**.**config(text=self**.**questionNumber)

Cette méthode, `addComponents`, ajoute les composants visuels à l'interface utilisateur. Elle commence par définir le style de la fenêtre en fonction du paramètre donné, en choisissant l'image de fond appropriée et en définissant la couleur de fond. Ensuite, elle crée la zone de la question avec le texte fourni, puis ajoute les éléments de la question, tels que les boutons de réponse. Si le style n'est pas égal à 4, elle ajoute également un bouton de validation et affiche le numéro de la question. Si le style est égal à 4, elle initialise une connexion au portail et passe à l'étape suivante. Enfin, elle lance le chronomètre, sauf si le style est égal à 4.

def createButton(self, row, text, button\_number):

        buttonBorder = Frame(self**.**body, bg="white")

        buttonBorder**.**grid(column=0, row=row, pady=7)

        fontStyle = font**.**Font(size=15, weight="bold")

        if self**.**style != 4:

            button = Button(buttonBorder, text=text, font=fontStyle,

justify='right',

                            width=50, height=2, cursor="hand2",

                            fg=self**.**master**.**color\_text,

bg=self**.**master**.**color\_fourth, bd=0,

                            highlightthickness=4, highlightbackground="white",

highlightcolor="white",

                            activebackground=self**.**master**.**color\_fourth,

activeforeground=self**.**master**.**color\_text,

                            command=lambda:

self**.**changeBorderColor(buttonBorder, button\_number))

        elif self**.**style == 4:

            button = Button(buttonBorder, text=text, font=fontStyle,

justify='right',

                            width=50, height=2, cursor="hand2",

                            fg=self**.**master**.**color\_text,

bg=self**.**master**.**color\_fourth, bd=0,

                            highlightthickness=4, highlightbackground="white",

highlightcolor="white",

                            activebackground=self**.**master**.**color\_fourth,

activeforeground=self**.**master**.**color\_text)

        button**.**grid(column=0, row=0, padx=5, pady=5)

        self**.**button\_borders**.**append(buttonBorder)

    def changeBorderColor(self, selected\_border, button\_number):

        for border in self**.**button\_borders:

            border**.**config(bg="white")

        selected\_border**.**config(bg="#114232")

        self**.**questionNumberSelect = button\_number

Cette méthode, `createButton`, est utilisée pour créer des boutons de réponse dans l'interface graphique. Elle prend en paramètres le numéro de ligne (`row`), le texte du bouton (`text`), et le numéro du bouton (`button\_number`). Selon le style de la fenêtre, elle crée un bouton avec ou sans fonction de changement de couleur lorsqu'il est cliqué. Si le style est différent de 4, un bouton interactif est créé, tandis que si le style est égal à 4, un bouton statique est créé. La méthode `changeBorderColor` est appelée lorsque le bouton est cliqué pour changer la couleur du cadre entourant le bouton sélectionné.