



ÉCOLE CENTRALE LYON

INF TC2
TD/BE5
RAPPORT

Programmation objet et interfaces

Élève :
Flavio SESTU (B1b)

Enseignant :
René CHALON

16 février 2022

Table des matières

1	Introduction	2
2	Implémentation du retour en arrière	2
3	Base de données et classement	3
4	Personnalisation de l'interface	5
4.1	Personnalisation simple : modification des couleurs et fonds	5
4.2	Personnalisation du personnage pendu : requêtes API et skin Minecraft	5
5	Diagramme de classe UML et conclusion	6

1 Introduction

L'objectif de ce rapport est de rendre compte de l'amélioration d'un jeu du pendu : personnalisation de l'interface, implémentation d'un retour en arrière et mise en place d'un système de sauvegarde des parties pour pouvoir afficher un classement.

De plus, j'ai fait le choix de modifier l'apparence du personnage "pendu" pour améliorer la personnalisation et pouvoir utiliser des requêtes API et manipuler des images dans mon projet, et ainsi amorcer de nouvelles compétences.

Je présenterai donc succinctement :

- L'implémentation du retour en arrière et la logique utilisée
- Le système de classement, et la façon d'identifier et présenter le classement pour permettre une expérience utilisateur un minimum optimisée.
- La personnalisation de l'interface, principalement l'affichage du personnage pendu.

2 Implémentation du retour en arrière

Pour pouvoir revenir en arrière, il faut avoir stocké les coups précédents, ce qui n'était pas fait jusque là. Ainsi, une liste `self.__coups` contenant les dernières lettres jouées sera tenue à jour tout au long de la partie. De même, une variable `self.__fini` sera initialisée à **False** en début de partie et modifiée pour **True** lorsque la partie se termine (par une victoire ou par une défaite) afin de pouvoir gérer cette situation différemment qu'un retour en arrière en milieu de partie (le texte affiché a changé et il faut supprimer l'enregistrement de la base de données).

```

290 # Ex 8 : ajout Undo
291 def triche(self):
292     lettre=self.__coups.pop()
293     for k in self.__clavierListe:# On dégrise la dernière case
294         if k['text']==lettre:
295             k.config(state='normal')
296     if lettre in self.__motSecret:# Si le dernier coup était un bon coup, on remet les *
297         mottemp=self.__mot.get()
298         actuel=mottemp[::-1][len(self.__motSecret):-1]
299         for i in range(len(self.__motSecret)):
300             if self.__motSecret[i]==lettre:
301                 actuel=actuel[:i]+'*'+actuel[i+1:]
302             self.__mot.set("Mot : "+actuel)
303             if self.__fini:
304                 self.__bdd.del_last()
305     else:# Si c'était un mauvais coup
306         if self.__fini:# Et que c'était fini, on remet le message tel quel
307             self.__mot.set(self.__perdu)
308             self.__bdd.del_last()
309             self.__fini=False
310         # Dans tous les cas, il faut retirer le dernier dessin et une erreur comptabilisée.
311         self.__canva.etatForme(self.__rates,'hidden')
312         self.__rates-=1
313
314     if self.__boutonsAllumes!=[]:# Si la partie était finie (gagnée/perdue), on remet les cases telles qu'elles étaient
315         for k in self.__boutonsAllumes:
316             k.config(state='normal')
317         self.__boutonsAllumes=[]

```

FIGURE 1 – Méthode triche() qui effectue le retour en arrière

Étapes suivies pour restaurer l'état précédent (Voir Figure 1) :

- On dégrise la dernière lettre sur le clavier
- Si le dernier coup était un bon coup : on récupère dans le texte du Label la partie représentant le mot secret, qui peut donc être le mot ou contenir des étoiles. On remplace la lettre par des étoiles, comme si elle n'avait pas été trouvée. Si de plus la partie est finie, on avait gagné, et on supprime la dernière ligne de la table **parties** de la base de données (cf. partie suivante).
- Si le coup était mauvais, la partie du label représentant le mot n'a pas changé. On cache la dernière partie du pendu qui a été affichée. Si la partie était finie (perdue), on change le texte du Label (qui contient un message qui explique que la partie est perdue) et on supprime la dernière ligne de la table **parties** comme en cas de victoire.
- Si la partie était finie, peu importe l'issue, on restaure les boutons qui n'avaient pas été utilisés (stockés dans une liste en fin de partie).

Enfin, la méthode permettant le retour en arrière a été liée à un bouton dans la barre de menu afin que le joueur la découvre de façon simple, ainsi qu'au Control-Z pour être fidèle à l'habitude des joueurs plus expérimentés.

3 Base de données et classement

Dans le but de créer un classement, les joueurs s'identifieront avec un pseudo. Le choix a été fait de simplement taper son pseudo, sans authentification par mot de passe pour éviter d'alourdir l'expérience utilisateur (UX) au vu du fonctionnement local du jeu. Après réflexion, pour conserver une bonne UX, le pseudo sera demandé en début de partie, pourra être changé via un bouton dans la barre de menu et sera demandé lors de l'affichage du classement s'il n'a pas encore été choisi (l'utilité de s'authentifier pour voir le classement sera vue ci-dessous).

Ce pseudo sera stocké dans une base de données et chaque partie sera stockée (dans une autre table) avec l'identifiant du joueur qui l'a terminée. La base de données **pendu.db** est composée de deux tables, et est structurée de la façon suivante :

- joueurs(idjoueur,pseudo)
- parties(idpartie,idjoueur,mot,score)

Comme conseillé, le score de chaque partie est égal au nombre de lettres trouvées divisé par la taille du mot. Le score total d'un joueur est donc la somme des scores de ses parties. Afin de retirer un certain biais, j'ai trouvé raisonnable d'afficher deux classements : un en fonction du score total, et l'autre en fonction du score moyen par partie.

Lorsque le joueur entre son pseudo, on exécute cette requête :

```
self.__curseur.execute("SELECT idjoueur FROM joueurs WHERE pseudo=(?)",(pseudo,))
```

Si le retour est vide, alors une exception déclenchera la requête suivante pour ajouter un nouveau joueur à la table joueurs :

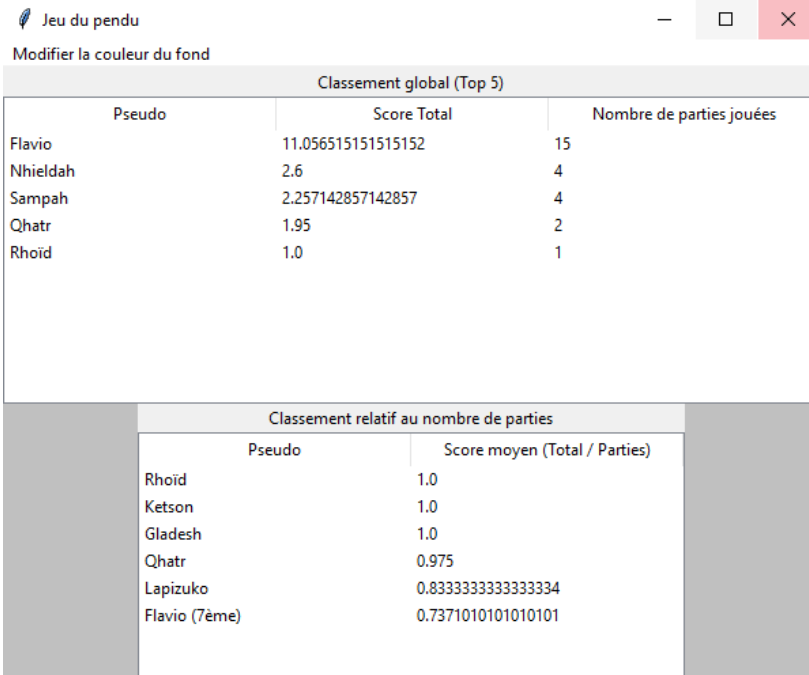
```
self.__curseur.execute("INSERT INTO joueurs (pseudo) VALUES (?)",(pseudo,))
```

Le procédé est semblable pour ajouter une partie en envoyant idjoueur, mot et score. Pour les tables **joueurs** et **parties** respectivement, les clés primaires sont idjoueur et idpartie qui s'auto-incrémentent et n'ont pas à être envoyées.

Ensuite, deux requêtes permettent d'obtenir les classements désirés. La première :

```
SELECT joueurs.pseudo,sum(score),count(score)
FROM parties JOIN joueurs ON joueurs.idjoueur=parties.idjoueur
GROUP BY joueurs.pseudo ORDER BY sum(score) DESC
```

Celle-ci classe les joueurs par score total. La seconde requête est très semblable, la seule différence est **ORDER BY sum(score)/count(score)** qui permet d'obtenir la même chose classée par score moyen par partie.



Classement global (Top 5)		
Pseudo	Score Total	Nombre de parties jouées
Flavio	11.056515151515152	15
Nhieldah	2.6	4
Sampah	2.257142857142857	4
Qhatr	1.95	2
Rhoid	1.0	1

Classement relatif au nombre de parties	
Pseudo	Score moyen (Total / Parties)
Rhoid	1.0
Ketson	1.0
Gladesh	1.0
Qhatr	0.975
Lapizuko	0.8333333333333334
Flavio (7ème)	0.7371010101010101

FIGURE 2 – Fenêtre classement

Mon choix a été de n'afficher que le top 5 des joueurs pour chaque classement, le tout dans une nouvelle fenêtre. Afin que le joueur connecté puisse toujours voir sa place, s'il n'est pas dans le top il est ajouté au classement, avec sa place précisée entre parenthèses. (Voir Figure 2)

4 Personnalisation de l'interface

4.1 Personnalisation simple : modification des couleurs et fonds

Mon choix pour personnaliser l'interface a été de permettre via un menu de modifier la couleur de plusieurs éléments (Voir Figure 3).

Pour le choix de la couleur, le module `tkinter.colorchooser.askcolor` fait parfaitement l'affaire. La couleur est ensuite stockée au format hexadécimal, et une simple méthode `objet.configure(bg=couleur)` pour les éléments de l'interface ou `objet.config(bg=couleur)` pour les éléments du canevas permet d'appliquer le choix.

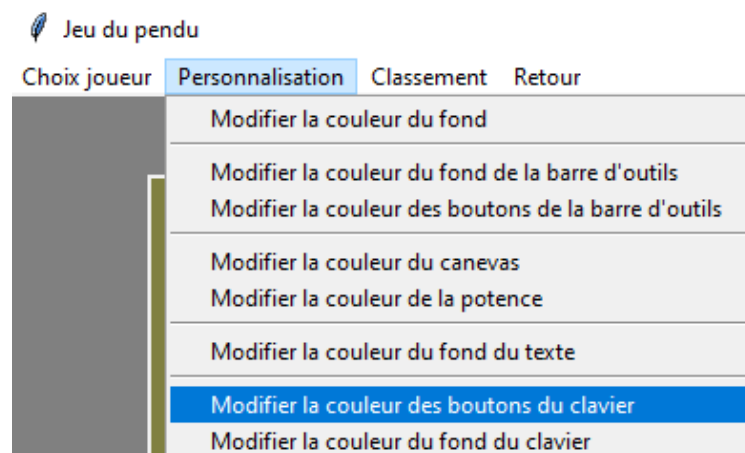


FIGURE 3 – Menu personnalisation

4.2 Personnalisation du personnage pendu : requêtes API et skin Minecraft

Pour personnaliser le jeu, rendre l'expérience utilisateur plus amusante et approfondir mon travail, j'ai fait le choix de remplacer les rectangles représentant le bonhomme pendu par un skin Minecraft (Voir Figure 4). Le "skin" est l'apparence du personnage, personnalisable dans le jeu Minecraft et propre à chaque joueur. L'idée est la suivante :

- Le skin par défaut est le mien (associé au pseudo Minecraft "FlavCraft0834"). Si le pseudo entré par le joueur en début de partie n'est pas associé à un compte Minecraft, le dernier skin affiché sera conservé.
- Si le pseudo entré est associé à un compte Minecraft, alors le skin sera récupéré et découpé, puis remplacera le pendu.

Mojang ne supportant plus son API, deux autres seront utilisées (avec de simples requêtes GET). La première, `playerdb.co` permet de savoir si le pseudo est associé à un compte Minecraft et, le cas échéant, d'obtenir (entre autres) l'ID associé. La seconde API, `visage.surgeplay.com` délivre, en fonction de l'ID donné, un fichier PNG représentant le skin de face, en 2D (à l'origine, un skin minecraft est un patron du skin 3D d'où le choix de cette API).

Ce fichier PNG, après des mesures et dimensionnements de ma part (j'ai d'ailleurs agrandi le

canevas et l'ensemble contenant la potence), sera découpé membre par membre puis enregistré tel quel. Le fichier **recupskin.py** contient la méthode **recupskin(pseudo)** qui enregistre les fichiers PNG des membres du skin associé au pseudo dans le dossier textbfskin/ du répertoire local et retourne True, ou bien retourne seulement False si le pseudo n'a pas de compte associé.

Pour l'affichage, la méthode **Canvas.create_image()** est utilisée. Les manipulations sur les images sont effectuées au format **PIL.Image** de la bibliothèque **Pillow**, puis le format **PIL.TkImage** est utilisé pour rendre les images compatibles avec le canevas Tkinter.

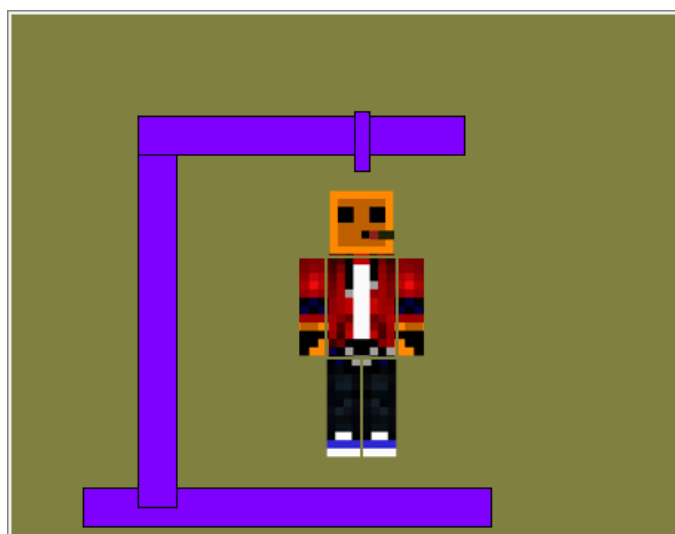


FIGURE 4 – Canevas en fin de partie perdue ou au lancement du jeu

5 Diagramme de classe UML et conclusion

Pour conclure, ce travail a permis d'explorer ou d'aborder les compétences suivantes :

- Programmation orientée objet
- Tkinter, interfaces graphiques et expérience utilisateur
- Base de données
- Requêtes GET API

Différentes pistes seraient à explorer pour améliorer le pendu : interface plus moderne, amélioration de l'expérience utilisateur (en tenant compte de retours!), ...
Le diagramme de classe correspondant au jeu est en Figure 5.

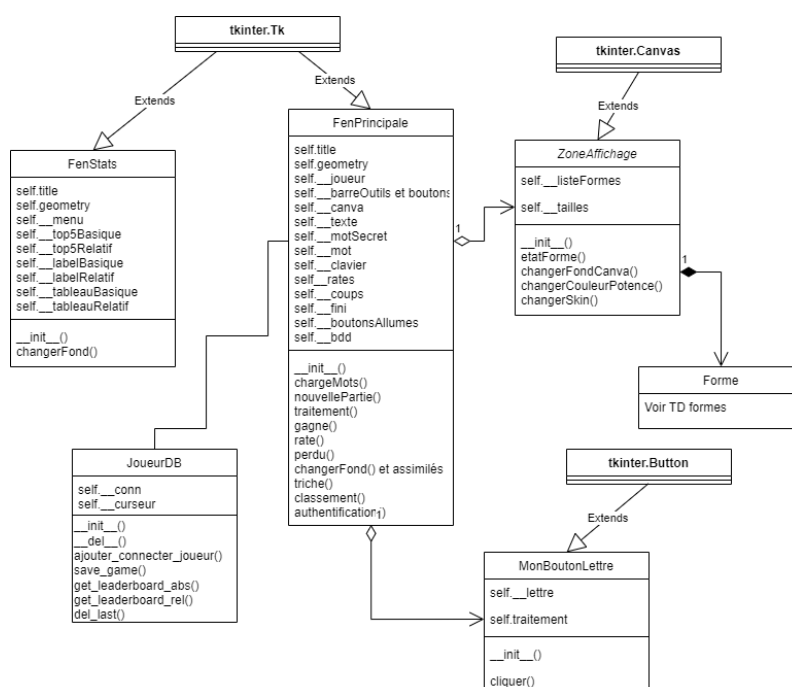


FIGURE 5 – Diagramme de classe UML