Analyse et et optimisation d’un projet

Bataille de Boules

Chamsedine AMOUCHE & Elias LAHLOUH

Résumé

Ce rapport présente l'application des techniques d'optimisation et de gestion de la mémoire sur un projet développé dans le cadre de la formation en BUT informatique. Le projet optimisé est le jeu Bataille des boules, réalisé en première année. L'objectif principal de ce travail était d'améliorer l'efficacité du code tout en assurant une gestion optimale de la mémoire, en particulier à travers l'utilisation des outils de profilage comme cProfile et Memray.

Bien que l'optimisation des performances soit au cœur du projet, certaines étapes, telles que la refactorisation du code, ont également été abordées dans une optique de lisibilité et de maintenabilité, sans impact direct sur les performances mais en facilitant les améliorations futures. En outre, ce rapport fournit une vue d'ensemble des modifications réalisées et des résultats obtenus, tout en omettant volontairement certains détails techniques spécifiques aux différentes étapes de résolution des problèmes, qui seront approfondis dans le rapport final à soumettre.

Table des matières

1. Contexte

1.1 Le projet

Le projet Bataille des Boules est un jeu développé dans le cadre du BUT Informatique en première année. Il s'agit d'un jeu compétitif dans lequel deux joueurs s'affrontent en tour par tour. Le but du jeu est de recouvrir le plus grand nombre de pixels possibles à l'aide de boules de couleur, chaque joueur contrôlant une couleur différente. Chaque joueur place des boules sur un espace de jeu, et l'objectif est d'avoir plus de pixels recouverts que l'adversaire à la fin de la partie.

Règles de base du jeu

- Non-intersection des boules : Les boules de couleurs différentes ne peuvent pas s'intersecter. Si un joueur tente de placer une boule qui entre en collision avec une boule adverse, il perd son tour et la boule n'est pas placée.
- Diviser les boules adverses : En cliquant sur une boule adverse, celle-ci se divise en deux boules plus petites.
- Limites du jeu : Un joueur ne peut pas placer une boule si celle-ci dépasse les limites de l'espace de jeu. En d'autres termes, aucune boule ne peut sortir de la surface jouable.

Variantes implémentées

- Sablier : Chaque tour est limité dans le temps. Un compte à rebours s'affiche à chaque tour, obligeant les joueurs à effectuer leur mouvement rapidement.
- Scores : Si un joueur appuie sur la touche 's', le score du jeu est affiché en temps réel pendant la partie.
- Terminaison : Cette fonctionnalité, utilisable une seule fois par partie, permet à un joueur de terminer la partie immédiatement après avoir appuyé sur la touche 't'. La partie se termine dans 5 tours après l'activation de cette fonctionnalité.
- Obstacles : Des obstacles (entre 4 et 7) apparaissent de manière aléatoire sur la surface jouable. Les boules ne peuvent pas toucher ces obstacles, ce qui ajoute un défi supplémentaire.
- Taille des boules : Un budget est alloué au début de chaque partie, permettant aux joueurs de définir le rayon des boules qu'ils placent. Les joueurs doivent gérer leur budget pour éviter de finir avec un budget nul.
- Dynamique des boules : Les boules déjà placées sur le terrain augmentent en taille à la fin de chaque tour, rendant le terrain de jeu de plus en plus occupé.

Bonus

- Pause et Sauvegarde : Le joueur peut mettre en pause la partie à tout moment et la reprendre plus tard, tout en sauvegardant sa progression.
- Classement : Une page d'accueil affiche un classement des trois meilleurs joueurs de tous les temps, permettant de suivre les performances historiques des joueurs.

1.2 Conditions de travail

Pour l'analyse des performances, nous avons activé toutes les variantes du jeu sauf 'Taille boules' et 'Terminaison', car Taille boules nécessite un temps supplémentaire pour remplir les valeurs et il faut se rappeler du budget alloué à chaque pour chaque test. Les variantes sélectionnées sont donc : Sablier, Obstacle, Score et Dynamique. Cela permet de tester le jeu dans une configuration complète pour obtenir un aperçu des performances avec le plius de fonctionnalités activées.

Le nombre de tours est fixé à 10 tours au total, répartis en 5 tours par joueur. Ce nombre est suffisant pour observer les performances du jeu tout en étant suffisamment court pour éviter des tests trop longs.

Il faut donc modifier cette ligne avec le nombre de tours souhaités :

Ensuite, nous lançons le jeu, puis utilisons la souris pour cliquer sur "Jouer". Le jeu se déroule de manière automatique et aléatoire : nous essayons de gérer diverses situations comme cliquer sur une boule adverse, cliquer à côté d'une boule adverse ou à côté d'une bordure, afin de simuler un gameplay varié. Cela permet de tester les performances du jeu dans des conditions proches de l'utilisation réelle.

2. Réduction du nombre d’appels de fonctions

```
Wed Jan  1 16:56:31 2025    profile_output.prof

         2921604 function calls (2912954 primitive calls) in 21.115 seconds

   Ordered by: call count
   List reduced from 2080 to 50 due to restriction <50>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   373021    0.023    0.000    0.023    0.000 {method 'append' of 'list' objects}
331572/331200    0.023    0.000    0.023    0.000 {built-in method builtins.len}
   314124   18.802    0.000   18.831    0.000 {method 'call' of '_tkinter.tkapp' objects}
   313517    0.023    0.000    0.023    0.000 upemtk.py:568(type_evenement)
   313517    0.081    0.000    0.103    0.000 upemtk.py:554(donne_evenement)
   313515    0.074    0.000   18.737    0.000 __init__.py:1465(update)
   313514    0.074    0.000   18.757    0.000 upemtk.py:71(update)
   313514    0.075    0.000   18.832    0.000 upemtk.py:156(mise_a_jour)
    52307    0.004    0.000    0.006    0.000 {built-in method builtins.isinstance}
    20440    0.003    0.000    0.005    0.000 __init__.py:1730(<genexpr>)
    13521    0.009    0.000    0.009    0.000 {method 'getint' of '_tkinter.tkapp' objects}
    12936    0.002    0.000    0.002    0.000 {built-in method builtins.getattr}
    11242    0.003    0.000    0.009    0.000 __init__.py:1723(getint_event)
     7398    0.001    0.000    0.001    0.000 typing.py:1799(<genexpr>)
     7240    0.001    0.000    0.001    0.000 {method 'rstrip' of 'str' objects}
     6713    0.001    0.000    0.001    0.000 {built-in method builtins.hasattr}
     6469    0.001    0.000    0.001    0.000 {method 'startswith' of 'str' objects}
     3750    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
     3257    0.000    0.000    0.000    0.000 {method 'find' of 'str' objects}
3141/3134    0.001    0.000    0.001    0.000 {method 'join' of 'str' objects}
     3095    0.000    0.000    0.001    0.000 {built-in method builtins.setattr}
     2952    0.000    0.000    0.000    0.000 _parser.py:239(__next)
     2833    0.000    0.000    0.000    0.000 {method 'strip' of 'str' objects}
2828/2398    0.002    0.000    0.005    0.000 {built-in method __new__ of type object at 0x101383cf0}
     2772    0.000    0.000    0.000    0.000 typing.py:1427(<genexpr>)
     2765    0.000    0.000    0.000    0.000 {method 'get' of 'mappingproxy' objects}
     2729    0.001    0.000    0.001    0.000 typing.py:1294(_is_dunder)
     2686    0.000    0.000    0.000    0.000 util.py:200(__call__)
     2586    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
     2509    0.000    0.000    0.001    0.000 _parser.py:167(__getitem__)
     2500    0.000    0.000    0.000    0.000 {method 'endswith' of 'str' objects}
     2477    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:491(_verbose_message)
     2476    0.001    0.000    0.001    0.000 typing.py:1122(_is_unpacked_typevartuple)
     2406    0.000    0.000    0.001    0.000 _parser.py:260(get)
     2313    0.001    0.000    0.002    0.000 <frozen importlib._bootstrap_external>:131(_path_join)
     2269    0.000    0.000    0.000    0.000 {built-in method builtins.min}
     2179    0.000    0.000    0.000    0.000 {built-in method builtins.chr}
     2123    0.000    0.000    0.000    0.000 {method 'lstrip' of 'str' objects}
     2044    0.001    0.000    0.001    0.000 {method 'getboolean' of '_tkinter.tkapp' objects}
     1966    0.000    0.000    0.000    0.000 {method 'pop' of 'list' objects}
     1871    0.000    0.000    0.000    0.000 {built-in method builtins.issubclass}
     1726    0.000    0.000    0.000    0.000 typing.py:840(<genexpr>)
     1694    0.000    0.000    0.000    0.000 unicode.py:63(<genexpr>)
     1690    0.001    0.000    0.001    0.000 typing.py:1368(__setattr__)
 1665/178    0.000    0.000    0.001    0.000 <frozen abc>:121(__subclasscheck__)
 1665/178    0.001    0.000    0.001    0.000 {built-in method _abc._abc_subclasscheck}
     1626    0.001    0.000    0.001    0.000 cbook.py:468(_strip_comment)
     1566    0.000    0.000    0.000    0.000 {method 'rpartition' of 'str' objects}
     1544    0.000    0.000    0.000    0.000 {method 'split' of 'str' objects}
     1472    0.000    0.000    0.000    0.000 {built-in method builtins.callable}
```

Figure 1 – Les méthodes ou fonctions les plus appelées lors de l’exécution du programme dans les conditions expliquées en sous-section 1.2 (TOP 50).

Voici un aperçu de ce que nous allons faire dans la suite, ainsi que de ce que nous allons ignorer et pourquoi, en examinant les fonctions de la figure 1 :

1. list.append sera traité en sous-section 2.1
2. len sera traité en sous section 2.2
3. les méthodes et fonctions de _tkinter.tkapp (_tkinter.tkapp.call, _tkinter.tkapp.getint etc.) ne seront pas traitées, car tous les appels à ces méthodes proviennent de fichiers inaccessibles (bibliothèque tkinter).
4. Ces trois fonctions, présentes dans upemtk.py, sont appelées exclusivement par la fonction attente_touche. Il n'est donc pas possible de réduire leurs appels sans modifier le fonctionnement même de attente_touche, ce qui pourrait perturber le déroulement du jeu. Ces fonctions sont utilisées pour gérer les événements de l'interface graphique et mettre à jour l'état du jeu en fonction des actions de l'utilisateur.
5. Il en va de même pour la fonction upemtk.update, qui est uniquement appelée par mise_a_jour. Comme mise_a_jour est utilisée pour rafraîchir l'état du jeu à chaque itération, réduire ses appels risquerait de nuire à l'expérience utilisateur et à la fluidité du jeu.
6. isinstance(), getattr(), hasattr(), getint(), getboolean() sont des fonctions utilisées pour vérifier des types ou obtenir des attributs d'objets. Elles sont rapides et nécessaires dans le fonctionnement du programme, donc aucun ajustement n'est requis.
7. Les fonctions appliquées sur les chaînes de caractères (comme strip(), rstrip(), join(), find() etc.) sont des opérations standard et essentielles en Python, donc aucune optimisation n'est nécessaire.
8. __next__, __new__, et callable() sont des fonctions utilisées pour l'itération sur des objets et la création d'objets, elles sont rapides et essentielles dans leur contexte, donc aucune optimisation n'est nécessaire.
9. Les appels aux fonctions de typing.py sont liés au système de typage de Python et ne peuvent pas être modifiés sans perturber l'intégrité du programme.
10. Les appels liés à <genexpr> sont principalement des générateurs ou des compréhensions utilisées dans les itérations. Ces appels sont déjà optimisés pour leur fonctionnement et ne nécessitent pas de modifications. Il n'y a rien à faire ici.
11. dict.get() on ne peut rien y faire (appelants non modifiables)
12. list.pop() sera traité en sous section 2.3
13. La fonction _strip_comment dans cbook.py est utilisée pour supprimer les commentaires dans les chaînes de texte et n'est pas modifiable, car elle est rapide et spécifique au traitement des données dans ce contexte, sans impact direct sur les performances du jeu.
14. {frozen importlib._bootstrap>:491(_verbose_message) provient de l'importation des modules Python et de l'affichage d'informations sur l'importation. Il est généré par l'importation des bibliothèques standard. Il n'est pas possible de réduire ces appels sans affecter le comportement de l'importation des modules, donc nous ne touchons pas à cela.
15. Les fonctions liées à _parser.py sont des fonctions spécifiques au traitement de texte ou de fichiers et sont nécessaires pour le fonctionnement interne du programme. Elles sont exécutées rapidement et n'ont pas d'impact sur les performances globales du jeu, donc aucun ajustement n'est requis.

16. Les appels à __subclasscheck__, issubclass et _abc_subclasscheck sont liés à la gestion des sous-classes dans Python et font partie du mécanisme interne de la gestion des classes. Ces appels sont nécessaires et ne peuvent pas être optimisés sans altérer la logique du système de types de Python. Il n'y a donc pas de modification à apporter.
17. Les fonctions chr() et min() sont des fonctions internes de Python utilisées pour des opérations standard, Il n'y a donc rien à modifier.

2.1 list.append()

Dans le code, pour ajouter les boules des joueurs dans la liste boule et les listes spécifiques à chaque joueur (Boules_bleu et Boules_rouge), chaque boule était ajoutée individuellement dans ces listes via append dans des boucles for.

```
903             if len(EmplacementsJoueur1) > 0:
904                 for i in EmplacementsJoueur1:
905                     boule1 = cercle(i[0], i[1], i[2], couleur=couleur_joueur1, remplissage=couleur_joueur1)
906                     boule.append(boule1)
907                     Boules_bleu.append(boule1)
908             if len(EmplacementsJoueur2) > 0:
909                 for i in EmplacementsJoueur2:
910                     boule2 = cercle(i[0], i[1], i[2], couleur=couleur_joueur2, remplissage=couleur_joueur2)
911                     boule.append(boule2)
912                     Boules_rouge.append(boule2)
```

Dans cet extrait, à chaque itération des boucles for, append est appelé à deux reprises pour chaque élément de EmplacementsJoueur1 et EmplacementsJoueur2. Cela conduit à de nombreux appels de la méthode append qui peuvent être coûteux en ressources lorsque le nombre d'éléments est élevé.

Les appels à append ont été optimisés en utilisant la compréhension de liste et la méthode extend. Voici les modifications :

Au lieu de créer chaque boule individuellement et de l'ajouter via append, une compréhension de liste a été utilisée pour créer toutes les boules en une seule fois. Ensuite, la méthode 'extend' est utilisée pour ajouter toutes les boules à la 'liste boule'.

```
903             if len(EmplacementsJoueur1) > 0:
904                 Boules_bleu = [cercle(i[0], i[1], i[2], couleur=couleur_joueur1, remplissage=couleur_joueur1) for i in EmplacementsJoueur1]
905                 boule.extend(Boules_bleu)
906             if len(EmplacementsJoueur2) > 0:
907                 Boules_rouge = [cercle(i[0], i[1], i[2], couleur=couleur_joueur2, remplissage=couleur_joueur2) for i in EmplacementsJoueur2]
908                 boule.extend(Boules_rouge)
```

Dans la version initiale de la fonction verifier_clic_boule, bien que la méthode append ne soit pas appelée explicitement, elle est indirectement utilisée lorsque nous ajoutons des éléments à la liste lst dans chaque itération de la boucle. Chaque appel à append provoque l'extension de la liste en ajoutant un nouvel élément.

```
def verifier_clic_boule(emplacementClic):
    """Permet de vérifier si l'utilisateur clique à l'intérieur d'une boule adverse.

    Arguments :
        emplacementClic : variable contenant l'abscisse, l'ordonnée et le type du clic de l'utilisateur.
    """
    x = emplacementClic[0]
    y = emplacementClic[1]
    lst = []
    EmplacementsJoueurAdversaire = []

    if tourJoueur == 1:
        EmplacementsJoueurAdversaire = EmplacementsJoueur2
    else:
        EmplacementsJoueurAdversaire = EmplacementsJoueur1

    f = True

    for Emplacement in EmplacementsJoueurAdversaire:
        if sqrt((Emplacement[0] - x) ** 2 + (Emplacement[1] - y) ** 2) <= Emplacement[2]:
            lst = [False, Emplacement[0], Emplacement[1]]
            return lst
        else:
            lst = [True, Emplacement[0], Emplacement[1]]
    return lst
```

Dans la version optimisée, nous avons remplacé l'appel implicite à append par une réaffectation directe de la variable lst. Au lieu d'ajouter des éléments à la liste, nous mettons directement à jour la variable avec une nouvelle valeur à chaque condition. Cela élimine l'utilisation d'append tout en obtenant le même résultat, mais de manière plus efficace.

```
def verifier_clic_boule(emplacementClic):
    """Permet de vérifier si l'utilisateur clique à l'intérieur d'une boule adverse.

    Arguments :
        emplacementClic : variable contenant l'abscisse, l'ordonnée et le type du clic de l'utilisateur.
    """
    lst = [True, None, None]
    EmplacementsJoueurAdversaire = EmplacementsJoueur2 if tourJoueur == 1 else EmplacementsJoueur1

    for Emplacement in EmplacementsJoueurAdversaire:
        if sqrt((Emplacement[0] - emplacementClic[0]) ** 2 + (Emplacement[1] - emplacementClic[1]) ** 2) <= Emplacement[2]:
            lst = [False, Emplacement[0], Emplacement[1]]
            return lst

    return lst
```

Dans l'ancienne version de la fonction remplacer_boule, nous utilisons plusieurs appels successifs à pop() (voir section 2.3) et append() pour modifier la liste emplacement et la liste boule. Cela provoque plusieurs manipulations coûteuses des listes pour chaque boule supprimée et ajoutée, ce qui augmente le nombre d'opérations et ralentit l'exécution du programme.

```
def remplacer_boule(emplacement, boule, lst, Clic, couleurBoule, Liste_des_pixels):
    """Permet d'effacer une boule si l'utilisateur adverse clique dessus, pour laisser de la place afin de la remplacer par 2 boules plus petites.…
    # Etape1 supprimer l'ancienne boule :
    for i in range(len(emplacement)):
        # Sert à savoir quelle boule supprimer
        if emplacement[i][0] == lst[1] and emplacement[i][1] == lst[2]:
            compteur = i
            x = emplacement[i][0]
            y = emplacement[i][1]
            r = emplacement[i][2]
            a = Clic[0]
            b = Clic[1]
            if (sqrt(((a - x) ** 2) + ((b - y) ** 2))) == 0:
                break
            else:
                efface(boule[compteur])
                # Etape2 Placer les 2 boules :
                # rayon du plus petit cercle (1er cercle)
                r1 = r - (sqrt((((x - a) ** 2) + ((y - b) ** 2))))
                r2 = r - r1  # rayon du deuxième cercle
                s = r - (2 * r1)
                t = s - r2
                va = (a - x) / (sqrt(((a - x) ** 2) + ((b - y) ** 2)))
                vb = (b - y) / (sqrt(((a - x) ** 2) + ((b - y) ** 2)))
                c = x + (t * va)
                d = y + (t * vb)

                boule1 = cercle(a, b, r1, couleur=couleurBoule, remplissage=couleurBoule, epaisseur=1)
                boule2 = cercle(c, d, r2, couleur=couleurBoule, remplissage=couleurBoule, epaisseur=1)
                # Etape 3 supprimer dans la liste d'emplacement l'ancien cercle, et y ajouter les 2 nouveaux cercles.
                emplacement.pop(i)
                emplacement.append([a, b, r1])
                emplacement.append([c, d, r2])
                # Etape 4 supprimer dans la liste des boules, l'ancienne boule et y ajouter les 2 nouvelles boules.
                boule.pop(i)
                boule.append(boule1)
                boule.append(boule2)
                #Etape 5 supprimer les pixels qui ne sont plus présent et affecter les nouveaux pixels.
                Liste_des_pixels=supprimer_pixel(Liste_des_pixels, x, y, r)
                Liste_des_pixels=affecter_pixel(Liste_des_pixels, a, b, r1, couleurBoule)
                Liste_des_pixels=affecter_pixel(Liste_des_pixels, c, d, r2, couleurBoule)
```

La modification remplace les appels successifs à pop() et append() par une opération plus concise utilisant des slices pour réaffecter les éléments dans les listes :

- emplacement[i:i+1] = [[a, b, r1], [c, d, r2]] remplace les éléments à l'indice i par deux nouveaux éléments dans la liste emplacement.
- boule[i:i+1] = [boule1, boule2] remplace de manière similaire dans la liste boule.

```python
def remplacer_boule(emplacement, boule, lst, Clic, couleurBoule, Liste_des_pixels):
    """Permet d'effacer une boule si l'utilisateur adverse clique dessus, pour laisser de la place afin de la remplacer par 2 boules plus petites.…
    # Etape1 supprimer l'ancienne boule :
    for i in range(len(emplacement)):
        # Sert à savoir quelle boule supprimer
        if emplacement[i][0] == lst[1] and emplacement[i][1] == lst[2]:
            compteur = i
            x = emplacement[i][0]
            y = emplacement[i][1]
            r = emplacement[i][2]
            a = Clic[0]
            b = Clic[1]
            if (sqrt(((a - x) ** 2) + ((b - y) ** 2))) == 0:
                break
            else:
                efface(boule[compteur])
                # Etape2 Placer les 2 boules :
                # rayon du plus petit cercle (1er cercle)
                r1 = r - (sqrt((((x - a) ** 2) + ((y - b) ** 2))))
                r2 = r - r1  # rayon du deuxième cercle
                s = r - (2 * r1)
                t = s - r2
                va = (a - x) / (sqrt(((a - x) ** 2) + ((b - y) ** 2)))
                vb = (b - y) / (sqrt(((a - x) ** 2) + ((b - y) ** 2)))
                c = x + (t * va)
                d = y + (t * vb)

                boule1 = cercle(a, b, r1, couleur=couleurBoule, remplissage=couleurBoule, epaisseur=1)
                boule2 = cercle(c, d, r2, couleur=couleurBoule, remplissage=couleurBoule, epaisseur=1)

                # Etape 3 supprimer dans la liste d'emplacement l'ancien cercle, et y ajouter les 2 nouveaux cercles.
                emplacement[i:i+1] = [[a, b, r1], [c, d, r2]]  # Remplacer directement avec une slice
                # Etape 4 supprimer dans la liste des boules, l'ancienne boule et y ajouter les 2 nouvelles boules.
                boule[i:i+1] = [boule1, boule2]  # Remplacer directement avec une slice
                #Etape 5 supprimer les pixels qui ne sont plus présent et affecter les nouveaux pixels.
                Liste_des_pixels=supprimer_pixel(Liste_des_pixels, x, y, r)
                Liste_des_pixels=affecter_pixel(Liste_des_pixels, a, b, r1, couleurBoule)
                Liste_des_pixels=affecter_pixel(Liste_des_pixels, c, d, r2, couleurBoule)
```

Dans la version initiale, une boucle for imbriquée était utilisée pour créer les éléments de Liste_des_pixels. À chaque itération de la boucle interne, l'élément (i, j, None) était ajouté à la liste Liste_des_pixels via la méthode append(). Bien que cette approche soit fonctionnelle, elle implique de multiples appels à append(), ce qui peut entraîner un léger ralentissement, surtout si le nombre d'itérations est élevé, comme c'est le cas ici avec les plages 200, 801.

```python
Liste_des_pixels = []
for i in range(200,801):
    for j in range(200,801):
        Liste_des_pixels.append((i,j,None))
```

La modification remplace la boucle classique par une compréhension de liste. Cette approche permet de créer directement la liste en une seule expression, ce qui est non seulement plus concis, mais aussi plus rapide en termes d'exécution, car elle est optimisée en interne par Python. La compréhension de liste génère tous les éléments et les ajoute à Liste_des_pixels en une seule opération, sans avoir besoin de faire appel à append() à chaque itération. C'est donc plus efficace et plus lisible.

```python
Liste_des_pixels = [(i, j, None) for i in range(200, 801) for j in range(200, 801)]
```

Dans l'ancienne version de la fonction Obstacles(), nous utilisons une boucle for pour générer une paire de coordonnées aléatoires (a, b) pour chaque obstacle. Chaque paire de coordonnées était ensuite ajoutée à deux listes séparées, obst_a et obst_b, via des appels successifs à append(). Cela entraînait plusieurs appels à append(), ce qui est généralement moins performant, surtout si le nombre d'obstacles (nb_obstacle) est élevé.

```
def Obstacles(nb_obstacle):
    """Fonction permettant de créer des obstacles.

    Arguments:
        nb_obstacle : variable contenant le nombre d'obstacles à placer.
    """
    obst_a = []
    obst_b = []

    for i in range(nb_obstacle):
        a = randint(200 + 50, 800 - 50)
        b = randint(200 + 50, 800 - 50)
        cercle(a, b, 30, 'black', 'black', 1, "obstacles")
        obst_a.append(a)
        obst_b.append(b)

    return obst_a, obst_b
```

La nouvelle version utilise une compréhension de liste pour générer directement la liste de paires de coordonnées (obstacles). Les coordonnées (a, b) pour chaque obstacle sont créées sous la forme d'une paire et stockées dans une liste obst en une seule opération, sans avoir à utiliser append().

Ensuite, nous affichons les obstacles via une boucle for, puis nous extrayons les listes obst_a et obst_b à l'aide de compréhensions de liste, en séparant les coordonnées en deux listes distinctes.

```
def Obstacles(nb_obstacle):
    """Fonction permettant de créer des obstacles.

    Arguments:
        nb_obstacle : variable contenant le nombre d'obstacles à placer.
    """
    obst = [(randint(200 + 50, 800 - 50), randint(200 + 50, 800 - 50)) for _ in range(nb_obstacle)]

    for a, b in obst:
        cercle(a, b, 30, 'black', 'black', 1, "obstacles")
        cercle(a, b, 30, 'black', 'black', 1, "obstacles")

    obst_a = [x[0] for x in obst]
    obst_b = [x[1] for x in obst]

    return obst_a, obst_b
```

Les modifications réalisées ont permis de réduire efficacement les appels à append(), en particulier en utilisant des compréhensions de liste et des remplacements directs dans les listes. Toutefois, des optimisations supplémentaires dans le main ou la gestion des **éléments** seraient possibles, mais elles impliqueraient des changements complexes et risqueraient de perturber la logique du code. Dans le cas des éléments, l'optimisation n'est pas pertinente étant donné le faible nombre de boutons à gérer.

2.2 len

Dans cette section, nous avons principalement cherché à optimiser les appels répétitifs à la fonction len(). L'un des principaux changements effectués a été de remplacer les expressions suivantes :

for i in range(len(quelquechose))

par une meilleure version :

n = len(quelquechose)
for i in range(n)

En calculant une seule fois la longueur de la liste et en l'utilisant pour l'itération, nous réduisons ainsi les appels redondants à len(). Cela permet d'améliorer la lisibilité du code tout en économisant quelques microsecondes, en particulier lorsque cette opération est effectuée dans une boucle appelée fréquemment.

Exemples :

avant :

```
def remplacer_boule(emplacement, boule, lst, Clic, couleurBoule, Liste_des_pixels):
    """Permet d'effacer une boule si l'utilisateur adverse clique dessus, pour laisser de la place afin de la remplacer par 2 boules plus petites.…
    # Etape1 supprimer l'ancienne boule :
    for i in range(len(emplacement)):
```

après :

```
def remplacer_boule(emplacement, boule, lst, Clic, couleurBoule, Liste_des_pixels):
    """Permet d'effacer une boule si l'utilisateur adverse clique dessus, pour laisser de la place afin de la remplacer par 2 boules plus petites.…
    # Etape1 supprimer l'ancienne boule :
    n = len(emplacement)
    for i in range(n):
```

avant :

```
def affecter_pixel(Liste_des_pixels,x,y,r,couleurJoueur):
    """Fonction permettant d'affecter un/des pixel à un joueur.

    Arguments:
        Liste_des_pixels : Liste de couples où tous les pixels du jeu sont enregistrés.
        x : abscisse de la boule posée.
        y : ordonnée de la boule posée.
        r : rayon de la boule posée.
        couleurJoueur : couleur du joueur qui se voit affecter un pixel.
    """
    for i in range(len(Liste_des_pixels)):
        x2=Liste_des_pixels[i][0]
        y2=Liste_des_pixels[i][1]
        if ((x2-x)<=r and (x2-x)>=0) and ((y2-y)<=r and (y2-y)>=0):
            Liste_des_pixels[i]=(x2,y2,couleurJoueur)

    return Liste_des_pixels
```

après :

```
def affecter_pixel(Liste_des_pixels,x,y,r,couleurJoueur):
    """Fonction permettant d'affecter un/des pixel à un joueur.

    Arguments:
        Liste_des_pixels : Liste de couples où tous les pixels du jeu sont enregistrés.
        x : abscisse de la boule posée.
        y : ordonnée de la boule posée.
        r : rayon de la boule posée.
        couleurJoueur : couleur du joueur qui se voit affecter un pixel.
    """
    n = len(Liste_des_pixels)
    for i in range(n):
```

Concernant les modifications spécifiques, la fonction appelait len(lst_col) plusieurs fois dans le même bloc de code, ce qui n'était pas optimal. En remplaçant chaque appel par une seule variable n qui contient la longueur de lst_col, nous avons évité plusieurs appels à len().

avant :

```
def verifier_indices(i, j):
    """Corrige les indices pour qu'ils ne soient pas identiques."""
    if i == j:
        if i == 0:
            j = len(lst_col) - 1
        elif i == len(lst_col) - 1:
            j = len(lst_col) - 2
        else:
            i += 1
    return i, j
```

après :

```
def verifier_indices(i, j):
    """Corrige les indices pour qu'ils ne soient pas identiques."""
    n = len(lst_col)  # Calculer la longueur une seule fois
    if i == j:
        if i == 0:
            j = n - 1
        elif i == n - 1:
            j = n - 2
        else:
            i += 1
    return i, j
```

La fonction mettre_a_jour_indices(), qui mettait à jour les indices en fonction des clics de l'utilisateur, était également optimisée de la même manière en stockant la longueur de lst_col dans une variable n.

```
def mettre_a_jour_indices(x, y, i, j):
    """Met à jour les indices de couleur en fonction des clics."""
    n = len(lst_col) # Calculer la longueur une seule fois
    if verifierEmplacementClic((x, y), 0, 400, 480, 425, 520):
        i = (i + 1) % n
    elif verifierEmplacementClic((x, y), 0, 85, 480, 110, 520):
        i = (i - 1) % n
    elif verifierEmplacementClic((x, y), 0, 565, 480, 590, 520):
        j = (j + 1) % n
    elif verifierEmplacementClic((x, y), 0, 895, 480, 920, 520):
        j = (j - 1) % n
    return i, j
```

Bien que des optimisations supplémentaires puissent être envisagées, notamment en utilisant des structures de données plus adaptées comme des dictionnaires pour éviter de parcourir les listes à chaque fois, ces changements ne sont pas réalisables à ce stade, car ils demanderaient des modifications complexes du code, ainsi qu'une refactorisation importante de certaines parties. En effet, ces optimisations impliqueraient un changement de la manière dont les données sont organisées et accédées dans le programme. Voici quelques exemples de fonctions qui pourraient bénéficier de telles optimisations :

- remplacer_boule() : Plutôt que de parcourir toute la liste à la recherche de la boule à supprimer ou à modifier, un dictionnaire pourrait être utilisé pour stocker les boules avec une clé unique, comme l'identifiant ou les coordonnées de chaque boule. Cela permettrait un accès direct à l'élément voulu, sans avoir à parcourir toute la liste.
- verifierIntersection() : Une structure de données plus adaptée, comme un dictionnaire, pourrait être utilisée pour limiter les boules à vérifier en fonction de la zone d'interaction.
- Obstacles() : Ici aussi, un dictionnaire pourrait être utilisé.

2.3 list.pop()

Dans le cadre de l'optimisation du code, nous avons particulièrement réduit le nombre d'appels à la méthode pop() des objets list. Une modification clé a été apportée dans la fonction remplacer_boule(), où nous avons remplacé l'utilisation de pop() par une slice pour remplacer directement un élément de la liste. Plutôt que d'utiliser pop() pour retirer un élément et réajuster la liste, nous avons utilisé la technique de slice pour remplacer l'élément à la position donnée par une nouvelle valeur.

avant :

```
# Etape 3 supprimer dans la liste d'emplacement l'ancien cercle, et y ajouter les 2 nouveaux cercles.
emplacement.pop(i)
emplacement.append([a, b, r1])
emplacement.append([c, d, r2])
# Etape 4 supprimer dans la liste des boules, l'ancienne boule et y ajouter les 2 nouvelles boules.
boule.pop(i)
boule.append(boule1)
boule.append(boule2)
```

après :

```
# Etape 3 supprimer dans la liste d'emplacement l'ancien cercle, et y ajouter les 2 nouveaux cercles.
emplacement[i:i+1] = [[a, b, r1], [c, d, r2]]  # Remplacer directement avec une slice
# Etape 4 supprimer dans la liste des boules, l'ancienne boule et y ajouter les 2 nouvelles boules.
boule[i:i+1] = [boule1, boule2]  # Remplacer directement avec une slice
```

Cette optimisation semble être la principale à effectuer dans ce contexte. Nous avons décidé de ne pas appliquer de modification à d'autres fonctions, telles que afficher_classement(), car l'appel à pop() dans ces fonctions ne constitue pas un goulot d'étranglement en termes de performances dans le cadre des conditions de travail actuelles.

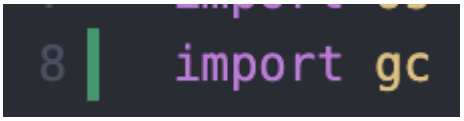
3. Réduction de la consommation mémoire

Afin de nous assurer qu'il n'y ait pas de fuite de mémoire dans notre programme Bataille des Boules, nous avons effectué plusieurs tests visant à surveiller la consommation mémoire tout au long de la partie et après la fin de la partie également. Nous avons mis en place différentes vérifications et ajouté le module gc pour gérer efficacement la mémoire.

Tests Effectués :

1. **Ajout du module gc :**

Nous avons intégré le module gc (Garbage Collector) dans le code afin de nous assurer de libérer la mémoire correctement à la fin de chaque grande modification. Ce module nous permet de forcer le nettoyage de la mémoire, en particulier lors de la suppression d'objets qui ne sont plus nécessaires.

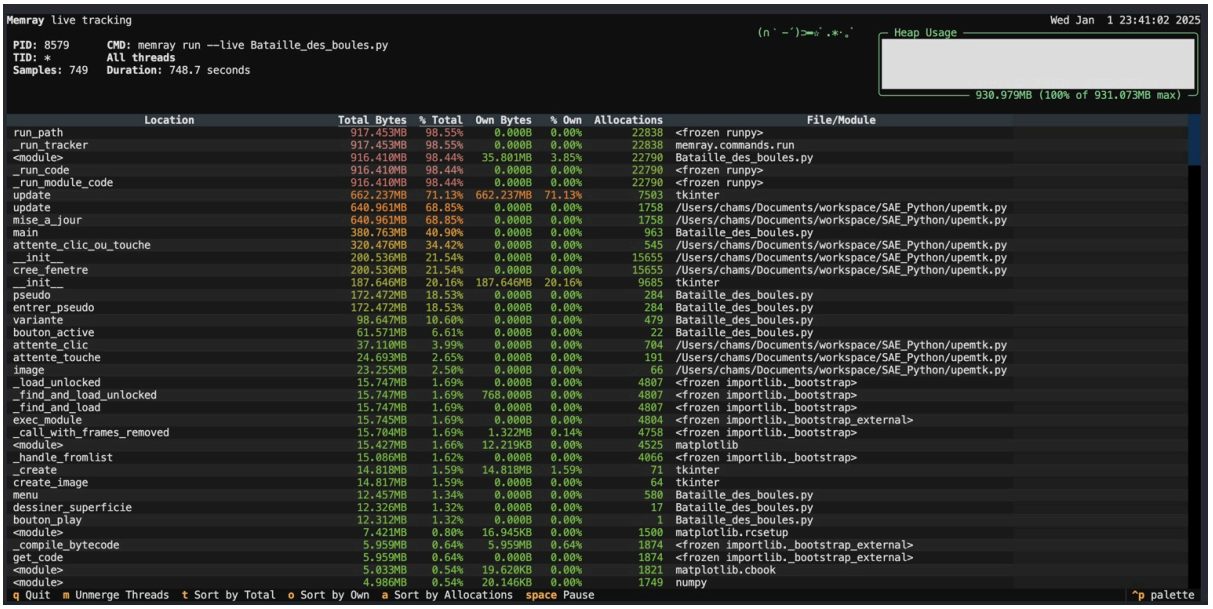


2. **Vider les grandes listes à la fin de la partie :**

Après chaque tour de jeu, nous avons pris soin de vider ou réinitialiser les listes les plus volumineuses, telles que la liste des pixels, ainsi que les listes contenant les boules des joueurs. Cela permet de s'assurer qu'aucune donnée inutile ne persiste en mémoire après la fin du jeu.

3.Test de la stabilité mémoire après 10 minutes :

Pour valider l'absence de fuite mémoire, nous avons laissé le programme tourner pendant 10 minutes après la fin du jeu, sans interaction. Cela nous a permis de constater si la mémoire se stabilisait ou continuait d'augmenter indéfiniment.



Précision sur l'utilisation de del et pop :

Nous avons choisi de ne pas utiliser le code pour gérer les listes, car ces méthodes ne correspondent pas à notre approche. En effet, nous affectons à nouveau directement les indices dans les listes (par exemple, en affectant un nouveau pixel ou une nouvelle position de boules) au lieu de supprimer ou d'extraire des éléments. Cette réaffectation des éléments permet de conserver l'intégrité des indices et des objets nécessaires tout en évitant l'utilisation de méthodes qui ne conviennent pas à la gestion des références dans notre contexte.

En conclusion, après avoir effectué des tests approfondis, nous pouvons confirmer qu'aucune fuite de mémoire n'a été détectée. La consommation mémoire se stabilise correctement à la fin du jeu. Le taux élevé de consommation observé, notamment pour Tkinter et le fichier Bataille_des_boules.py, est normal et dû à la gestion des objets graphiques nécessaires au fonctionnement du jeu. Les méthodes utilisées pour gérer la mémoire, telles que la réaffectation des objets et l'utilisation du module gc pour le nettoyage explicite, s'avèrent efficaces et ne causent pas de perte de ressources.

4. Refactorisation du code

Lors de l'analyse du code source, il est apparu que plusieurs fonctions étaient relativement longues, en particulier la fonction main, qui était en quelque sorte un "fourre-tout" de diverses opérations. Ce code, bien que fonctionnel, manquait de lisibilité et de clarté. Afin d'améliorer cette situation, nous avons procédé à un découpage des fonctions trop longues, comme c'est le cas pour la fonction Accueil(), dans le but de rendre le code plus lisible et plus facile à maintenir.

Cependant, bien que cette réorganisation ait amélioré la lisibilité du code, nous avons constaté que la fonction main reste encore assez longue et désorganisée. Le découpage de cette fonction n'a pas suffi à rendre le code propre et bien structuré. Pour obtenir un code vraiment propre et facile à maintenir, il serait nécessaire de procéder à une refactorisation en profondeur, en réorganisant le flux logique du programme et en décomposant davantage les blocs fonctionnels.

Cela dit, il est important de noter que cette refactorisation du code dépasse le cadre de ce projet (un code bien refactorisé peut être plus facilement optimisé mais ce n'est pas de l'optimisation) et que le temps alloué pour ce travail était limité. En conséquence, bien que nous ayons amélioré la structure du code, une refactorisation complète serait bénéfique pour les projets futurs. Une telle refactorisation améliorerait non seulement la lisibilité du code, mais aussi les possibilités d'optimisation en facilitant l'identification des points de ralentissement et en simplifiant les ajustements futurs.

5. Impacts des modifications

Pour pouvoir comparer de la meilleure des manières, il faudrait comparer avec le même nombre de boules, de tours, etc. Comme nous avons fixé le nombre de tours à 10, nous sommes en bonne position pour effectuer cette comparaison. Il sera cependant important d'éviter d'appeler la fonction terminaison, afin de ne pas rallonger ou raccourcir la partie, et ainsi garantir un environnement aussi similaire que possible pour obtenir la meilleure comparaison possible.

L'analyse de l'impact des optimisations a été réalisée à l'aide de plusieurs outils de profiling et d'analyse de la mémoire, notamment memray en mode live. Ce mode nous a permis de mesurer l'évolution de la consommation de mémoire et d'identifier si les optimisations mises en place avaient un réel impact.

5.1 Impact sur le nombre d'appels de fonctions

Pour évaluer l'impact des optimisations effectuées, nous avons comparé le nombre d'appels des fonctions clés avant et après les améliorations. Afin de garantir une comparaison la plus équitable possible, le nombre de boules et le nombre de tours ont été fixés à 10, et nous avons veillé à ce que les paramètres du jeu restent constants tout au long des tests. En utilisant le module cProfile, nous avons profilé les deux versions du programme (avant et après optimisation) pour analyser les fonctions les plus appelées, ainsi que le nombre d'appels pour chaque fonction :

python3 -m cProfile -o avant.stats -s ncalls Bataille_des_boules.py

python3 -m cProfile -o apres.stats -s ncalls Bataille_des_boules.py

Voici un extrait des résultats :

Fonction	nb appel avant	nb appel après
"<method 'append' of 'list' objects>")	372813	11630
'<built-in method builtins.len>')	276757	242416
"<method 'pop' of 'list' objects>")	1756	1785

Figure 2 – Extrait du tableau comparatif du nombre d'appels avant/après des fonctions que nous avons optimisées.

Le nombre d'appels de la méthode append a diminué de manière significative, passant de 372813 à 11630. Cela suggère que nous avons réussi à limiter les ajouts inutiles d'éléments dans les listes, ce qui a une incidence directe sur la gestion mémoire.

Concernant la fonction len, bien que l'optimisation ait permis une réduction notable du nombre d'appels, nous observons que le nombre d'appels reste encore relativement élevé (de 276757 à 242416). Nous estimons qu'une réduction supplémentaire de ce nombre pourrait être réalisée par une refactorisation du code, en particulier en modifiant certaines structures de données que nous avions choisies lors de la phase initiale du développement. Par exemple, il serait judicieux de remplacer certaines listes par des structures plus adaptées aux besoins spécifiques de notre jeu, afin de limiter l'appel systématique à len.

En ce qui concerne la fonction pop, nous avons observé une légère régression après optimisation, avec un petit nombre d'appels supplémentaires (de 1756 à 1785). Bien que cette régression soit minime, il est important de noter que l'appel à pop() dans notre code a été réduit à une seule occurrence, comme prévu. Cependant, cette légère augmentation pourrait être le résultat de plusieurs facteurs. Il est possible que l'appel à pop dans notre code soit indirectement causé par des appels d'autres parties du programme, ou même par des appels provenant de bibliothèques externes que nous utilisons, comme Tkinter ou d'autres modules. Dans ce cas, la gestion de la mémoire et des objets dans ces bibliothèques pourrait entraîner des appels à cette fonction. Nous concluons donc que cette légère augmentation ne provient pas directement de notre code, mais pourrait être liée à des appels indirects ou externes dans des modules tiers.

En résumé, nos modifications ont eu des effets notables sur le nombre d'appels des fonctions. L'appel à append() a été réduit de manière significative, ce qui a conduit à une réduction drastique de sa consommation mémoire. Quant à len(), bien que l'impact soit plus modéré, nous avons observé une légère amélioration. En revanche, pour pop(), les résultats montrent quasiment aucun changement, malgré nos efforts pour limiter son utilisation dans notre code. Cela suggère que l'augmentation minime observée pourrait être liée à des appels indirects ou externes, provenant possiblement de bibliothèques tierces. Enfin, bien que plusieurs autres fonctions aient vu leur nombre d'appels diminuer, il est difficile de l'expliquer de manière précise, car ces réductions peuvent résulter de facteurs variés, y compris des optimisations indirectes liées à la gestion des structures de données.

5.2 Sur la consommation en mémoire

Les tests effectués avec Memray ont permis de valider l'absence de fuites de mémoire dans notre application. Après avoir laissé tourner le programme pendant un certain temps (10 minutes), nous avons constaté que la consommation mémoire se stabilise à nouveau, ce qui confirme qu'il n'y a pas d'accumulation continue de mémoire.

Les principaux consommateurs de mémoire sont les fonctions liées à Tkinter et à notre fichier principal Bataille_des_boules.py, ce qui est attendu étant donné la gestion des objets graphiques dans l'application. Nous avons utilisé le module gc pour explicitement gérer la mémoire et éviter d'éventuelles fuites, et cette approche a permis de garantir que la mémoire est libérée correctement après chaque cycle.

Ainsi, les optimisations et les mesures mises en place ont prouvé leur efficacité en évitant toute fuite de mémoire, tout en assurant une gestion appropriée des ressources pendant l'exécution du programme.

5.3 Lien entre refactorisation et optimisation

La refactorisation, bien qu'indispensable pour améliorer la lisibilité et la maintenabilité du code, ne fait pas directement partie du processus d'optimisation des performances. Cependant, elle a une forte influence indirecte sur ce dernier. En effet, certaines fonctions de notre projet, comme la fonction main, étaient trop longues et manquaient de clarté, ce qui compliquait leur gestion et leur optimisation. Par exemple, la fonction Accueil aurait pu être découpée en plusieurs sous-fonctions pour rendre le code plus lisible et plus facile à maintenir. Une structure de code propre et bien organisée simplifie l'ajout de nouvelles fonctionnalités et améliore les futures possibilités d'optimisation.

Cependant, bien que la refactorisation permette d'avoir un code plus propre et plus facile à optimiser, elle n'entraîne pas nécessairement un gain de performance immédiat. Dans ce projet, la refactorisation n'a pas été poussée au-delà de la réduction de la taille de certaines fonctions longues, car le temps était limité et le but principal était d'optimiser les performances. Ainsi, bien qu'une refactorisation plus approfondie du main et d'autres parties du code aurait permis de rendre l'application plus modulable et mieux structurée, cela dépasse le cadre de ce projet. Il est important de noter que, même si la refactorisation peut rendre le code plus facile à optimiser à l'avenir, elle n'apporte pas de gains directs en termes de performances dans cette étape spécifique du projet.