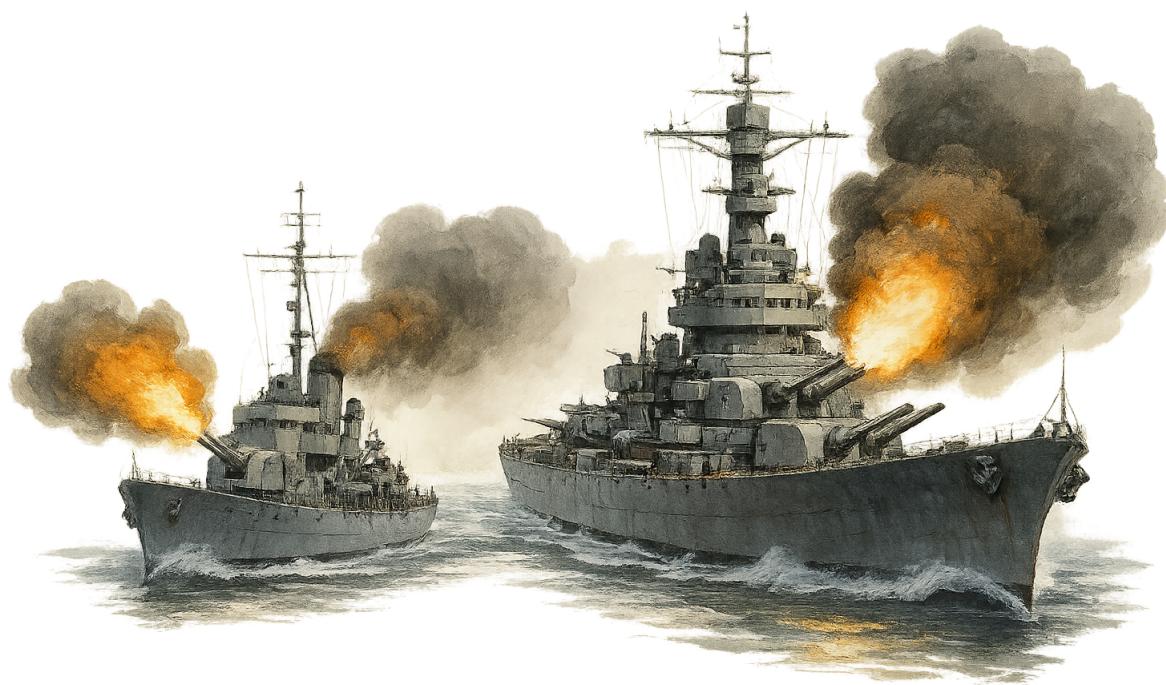


Projet Informatique
Polytech Clermont-Ferrand
présenté par Cindy et Rémi
Janvier 2026

BattleShip



Enseignant référent : Christophe de Vaulx

Table des matières

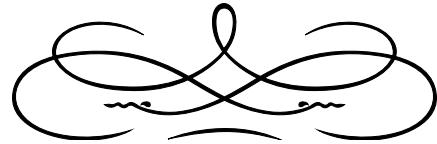
1	Organisation	3
1.1	GitHub	3
2	Version 1	5
2.1	Présentation du jeu	5
2.2	Mise en œuvre du projet	5
2.3	Structures de données	6
2.3.1	Vue d'ensemble	6
2.3.2	Grille d'attaque	6
2.3.3	Coordonnées des bateaux	6
2.3.4	Métadonnées des bateaux	6
2.4	Optimisation	7
2.5	Placement des bateaux	7
2.6	Robustesse du code	10
3	Version 2	12
3.1	Présentation du jeu	12
3.2	Mise en œuvre du projet	13
3.3	Structures de donnée	13
3.4	Intelligence Artificielle de l'Adversaire	14
3.4.1	Mode Facile : Attaque Aléatoire	14
3.4.2	Mode Difficile : Stratégie Adaptative	14
3.4.3	Comparaison des Performances	15
3.5	Fonctionnalités Additionnelles	15
3.6	Optimisation	16
3.7	Robustesse du code	16
4	Version 3	18
4.1	Mise en œuvre du projet	19
4.2	Fonctionnalités importante de Pygame	20
4.3	Difficultés rencontrées	21
4.4	Ressources	22
5	Conclusion	22
6	Références	23

1 Organisation

1.1 GitHub

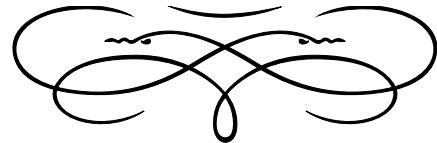
Afin d'assurer une organisation efficace du travail, ainsi qu'un suivi clair des évolutions du projet, l'ensemble du développement a été centralisé sur un dépôt GitHub.

Ce dépôt contient le code source du jeu *BattleShip*, les différentes versions, ainsi que les instructions et conditions nécessaires pour lancer le jeu. Il constitue ainsi le support principal du travail collaboratif réalisé.



Cliez Ici!

github.com/Battleship.git



VERSION I

2 Version 1

2.1 Présentation du jeu

Dans cette partie, le joueur joue seul et doit retrouver puis couler les 5 bateaux de longueur 5, 4, 3, 2 et 1 placés aléatoirement par l'ordinateur. La surface de jeu, montrée ci-dessous, est constituée d'une grille sur laquelle le joueur tire, un espace pour afficher un message relatif à la dernière action du joueur ainsi que son score (le nombre de tirs) et un espace pour entrer les coordonnées de sa prochaine attaque.



Légende :

- * : Coup dans l'eau
- + : Touché un bateau
- X : Coulé un bateau

2.2 Mise en œuvre du projet

Fichiers Relatifs

```
Battleship_Project/  
└── Battleship_v1.py  
└── weightmap_boats.npy
```

Dépendances logicielles

Le jeu nécessite l'installation de python avec une version entre 3.11 et 3.14

La seule bibliothèque à installer au préalable nécessaire pour lancer cette version est **numpy** en version 2.3.4 ou plus.

Lancement du jeu

Pour lancer le jeu, il suffit d'exécuter la commande suivante depuis le répertoire *Project Battleship* contenant les fichiers ci-dessus :

Commande de lancement Linux / macOS

```
python3 Battleship_v1.py
```

Commande de lancement Windows

```
py Battleship_v1.py
```

2.3 Structures de données

La conception du jeu repose sur plusieurs structures de données complémentaires. Le choix de ces structures a été guidé par des considérations de performance (voir 2.3 Optimisation), de lisibilité du code et de facilité de manipulation.

2.3.1 Vue d'ensemble

Le jeu utilise principalement deux types de structures :

- **Matrices (NumPy ndarray)** : pour la représentation visuelle et l'état de la grille de jeu
- **Dictionnaires Python** : pour le stockage optimisé des informations sur les bateaux

2.3.2 Grille d'attaque

La grille d'attaque est implémentée comme un tableau NumPy de dimensions 10×10 , initialisé avec des espaces vides, puis stocke l'état de chaque case selon la convention suivante :

Symbol	Signification	Description
" "	Case vierge	Le joueur n'a pas encore attaqué cette position
"*"	Tir manqué	Le joueur a attaqué cette case mais n'a touché aucun bateau
"+"	Bateau touché	Le joueur a touché un bateau, mais celui-ci n'est pas encore coulé
"X"	Bateau coulé	Toutes les cases du bateau ont été touchées, il est coulé

TABLE 1 – Convention des symboles dans la grille d'attaque

L'accès aux éléments de la matrice se fait par indexation bidimensionnelle :

`attack_grid[ligne][colonne]`, où les indices varient de 0 à 9. Le joueur, quant à lui, entre des coordonnées sous forme de lettres (A-J) pour les lignes et de nombres (1-10) pour les colonnes.

2.3.3 Coordonnées des bateaux

Contrairement à la grille d'attaque, les positions des bateaux ne sont pas stockées dans une matrice. Nous utilisons un dictionnaire où :

- **Clés** : symboles des bateaux ("P", "C", "S", "T", "B")
- **Valeurs** : listes de tuples représentant les coordonnées (*ligne, colonne*)

Exemple de structure :

```
1 boat_coords = {
2     "P": [(3, 2), (3, 3), (3, 4), (3, 5), (3, 6)],    # Porte-avions
3     "C": [(7, 1), (7, 2), (7, 3), (7, 4)],          # Croiseur
4     "S": [(0, 5), (1, 5), (2, 5)],                  # Sous-marin
5     "T": [(5, 8), (6, 8)],                          # Torpilleur
6     "B": [(9, 0)]                                    # Barque
7 }
```

2.3.4 Métadonnées des bateaux

Deux dictionnaires supplémentaires stockent les informations statiques sur les bateaux :

1. **Tailles des bateaux** : {"P": 5, "C": 4, "S": 3, "T": 2, "B": 1}
2. **Noms complets** : utilisé pour l'affichage des messages au joueur

```

1 boat_names = {
2     "P": "le Porte-Avion",
3     "C": "le Croiseur",
4     "S": "le Sous-marin",
5     "T": "le Torpilleur",
6     "B": "la Barque"
7 }

```

2.4 Optimisation

L'optimisation du code et de l'algorithme a fait partie intégrante du projet. Tout d'abord, nous nous sommes intéressés à l'optimisation temporelle (temps de calcul), spatiale (place de stockage) ainsi qu'à l'équilibre entre les deux.

Au début de notre projet, nous utilisions uniquement des matrices pour stocker des informations. Notre modification principale fut alors de transformer les matrices statiques contenant peu d'information en dictionnaire avec les coordonnées souhaitées de manière à obtenir les structures de données présentées ci-dessus (voir 2.2 Structures de données). Cela permet de réduire l'espace de stockage en supprimant les cases vides inutilisées de la matrice initiale, ainsi que le temps de calcul puisque nous n'avons plus besoin de parcourir tout le tableau de 100 cases pour trouver les bateaux.

TABLE 2 – Analyse comparative des performances : matrice statique vs. dictionnaire de coordonnées

Critère	Matrice 10×10	Dictionnaire	Amélioration
<i>Complexité temporelle</i>			
Recherche d'un bateau	$O(n^2) = O(100)$	$O(1)$	$\times 100$
Vérification collision	$O(n^2) = O(100)$	$O(k) = O(15)$	$\times 6,7$
Parcours des bateaux	$O(n^2) = O(100)$	$O(k) = O(15)$	$\times 6,7$
<i>Complexité spatiale</i>			
Espace alloué	100 cases	5 clés + 15 tuples	-70%
Stockage (valeurs)	100	30 + 5 clés	-65%
Cases utilisées	15/100 (85% vide)	15/15 (100% utilisé)	+85% efficacité

Note : n = dimension de la grille (10), k = nombre total de cases occupées (15).

Les bateaux ont des longueurs respectives de 5, 4, 3, 2 et 1 cases.

Nous avons cependant conservé le stockage en matrice pour les éléments pouvant être mis à jour durant la partie et étant utilisés pour l'affichage. Nous avons ensuite cherché à optimiser notre algorithme, même si cela pouvait être aux dépens de nos efforts d'optimisation spatiale et temporelle.

2.5 Placement des bateaux

L'ensemble de cette optimisation repose sur les expérimentations et analyses réalisées dans le notebook *Boat_placement_testing.ipynb*, disponible sur GitHub. Vous pourrez y trouver les algorithmes et leurs descriptions respectives, qui ne seront pas aussi détaillés ci-dessous, pour chaque version.

Note de lecture

Les heatmaps présentées ci-après utilisent une échelle de colorimétrie **relative**, normalisée individuellement pour chaque figure en fonction de ses valeurs minimales et maximales. Les couleurs ne sont donc pas directement comparables entre les différentes visualisations.

La première version de notre placement de bateaux était assez rudimentaire. Elle consistait à générer des positions au hasard jusqu'à trouver une configuration valide, avec comme orientation possible la droite et le bas, et une boucle qui s'arrêtait uniquement après avoir trouvé une position valide pour l'ensemble des bateaux. Au vu de la taille de la grille, et sachant que nous commençons par placer les bateaux les plus longs, le risque d'avoir un programme qui boucle sans fin était presque nul mais pas impossible.

Nous avons donc décidé de simuler 100 000 placement de bateaux avec cette première version et de compter combien de fois un bateau apparaissait sur chaque case de la grille, et nous avons obtenu la heatmap ci-dessous.

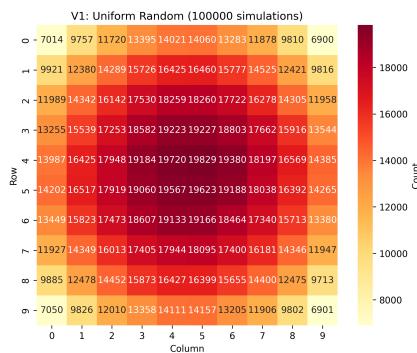


FIGURE 1 – Heatmap illustrant la distribution des bateaux avec la version 1

En voyant ce résultat traduisant une répartition loin d'être uniforme, nous avons cherché des méthodes pour l'améliorer. Notre première idée fut de créer une carte de poids à partir de la figure 1, et de l'utiliser lors de la sélection de la case de départ du bateau en générant un hasard avec biais. Nous avons ensuite réalisé la même simulation avec 100 000 itérations.

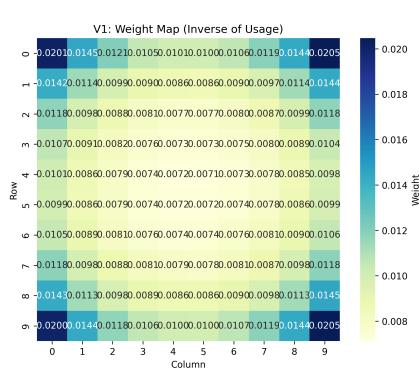


FIGURE 2 – Carte de poids de la version 1

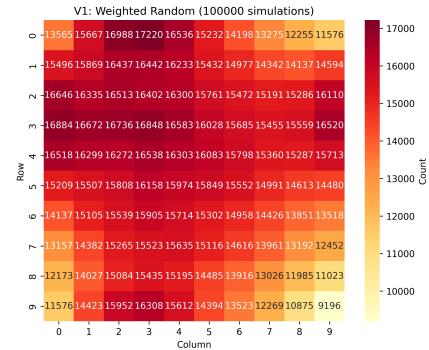


FIGURE 3 – Heatmap de la distribution des bateaux de la version 1 avec carte de poids

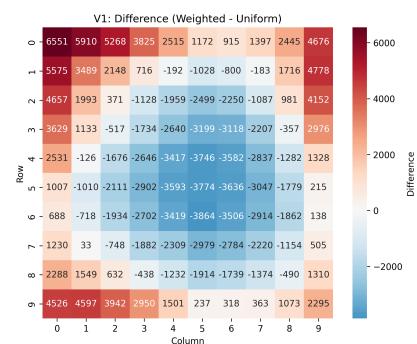


FIGURE 4 – Heatmap illustrant la différence de distribution entre les méthodes avec et sans poids

N'étant pas satisfait du résultat malgré l'amélioration, nous avons changé d'approche en regardant l'algorithme directement. Nous l'avons donc modifié pour lui permettre d'évoluer dans les 4 directions et pour introduire une logique adaptative : au lieu d'abandonner immédiatement une position invalide, l'algorithme cherche à placer le bateau autrement dans la même orientation et avec la même case de départ en cas de chevauchement ou dépassement de la grille. Le but de cette amélioration était d'augmenter l'utilisation des cases proches du bord de la grille qui était facilement rejetée dans la première version (par exemple : la case d'index (9,9) en bas à droite était uniquement sélectionnée pour le porte-avion de 5 cases si le hasard choisissait (9,5) horizontal ou (5,9) vertical). Nous avons ensuite simulé avec et sans la carte de poids.

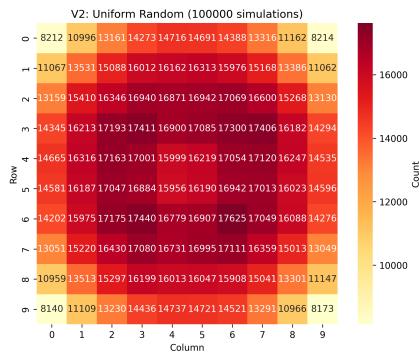


FIGURE 5 – Distribution sans poids de la version 2

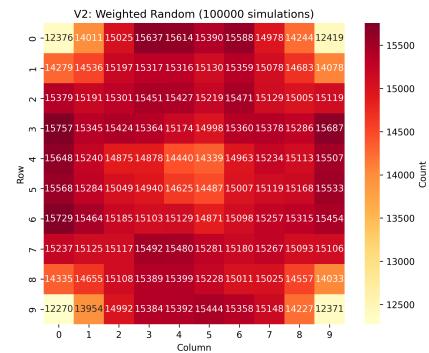


FIGURE 6 – Distribution avec poids de la version 2

Malgré l'amélioration importante que la version 2 nous apportait, nous avons décidé de développer d'autres idées qui nous étaient venues. Premièrement, nous avons développé un algorithme ressemblant, mais qui au lieu de choisir un sens au hasard, utilise le point sélectionné comme le centre du bateau, tout en conservant la logique adaptative de la version précédente.

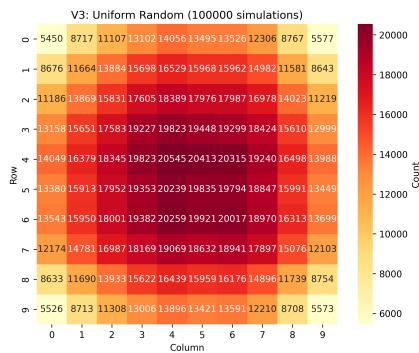


FIGURE 7 – Distribution sans poids de la version 3

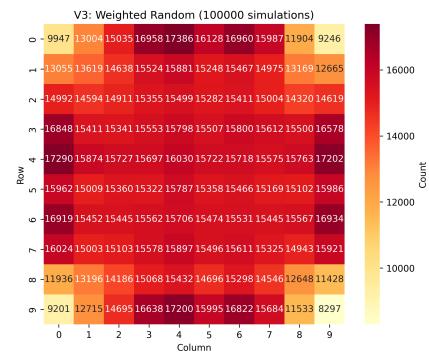


FIGURE 8 – Distribution avec poids de la version 3

Deuxièmement, nous avons développé un algorithme basé sur une stratégie de répulsion. L'idée centrale était de calculer dynamiquement des poids en fonction de la densité des bateaux déjà placés : une carte de densité est générée autour de chaque case occupée, puis inversée pour favoriser les zones moins peuplées lors du placement du prochain bateau. Cette approche permet de répartir naturellement les bateaux sur l'ensemble de la grille en évitant les zones déjà occupées. L'algorithme conserve également la logique adaptative de la version 2 et inclut un mécanisme de placement forcé en cas d'échec après un nombre maximal de tentatives, garantissant ainsi qu'une configuration valide soit toujours trouvée.

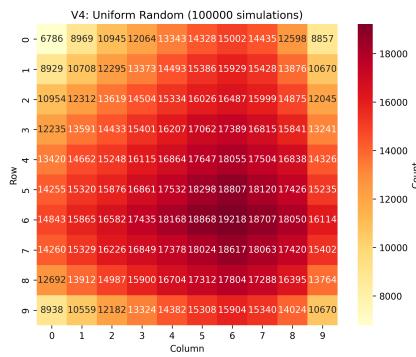


FIGURE 9 – Distribution sans poids de la version 4

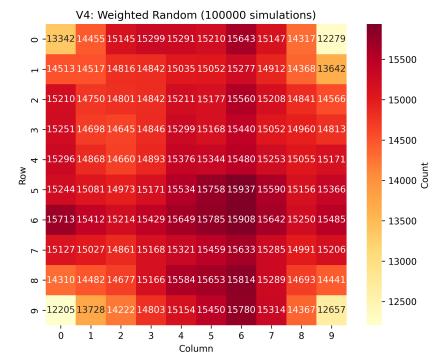


FIGURE 10 – Distribution avec poids de la version 4

TABLE 3 – Analyse statistique comparative : placement uniforme vs. pondéré

Version	Statistiques de placement (100 000 tentatives)								Amélioration
	Min	Max	σ	Var.	Min	Max	σ	Var.	
	Méthode Classique				Méthode Pondérée				
V1	6 900	19 829	3 223,95	10 393 866	9 196	17 220	1 519,88	2 310 027	-77,78
V2	8 140	17 625	2 252,14	5 072 139	12 270	15 757	668,85	447 355	-91,18
V3	5 450	20 545	3 812,98	14 538 789	8 297	17 386	1 690,17	2 856 668	-80,35
V4	6 786	19 218	2 572,78	6 619 173	12 205	15 937	656,96	431 590	-93,48

En observant les résultats présentés dans le tableau 3, nous avons sélectionné les algorithmes pondérés V2 et V4 comme candidats finaux potentiels. En effet, leurs écarts-types sont similaires et nettement inférieurs à ceux des autres versions, témoignant d'une meilleure stabilité dans la répartition. Cependant, ayant constaté un temps de simulation important lors des tests de V4, nous avons mesuré les temps de calcul moyens respectifs sur une grille de 10 par 10 : $2,7 \times 10^{-7}$ s pour V2 et $4,25 \times 10^{-7}$ s pour V4. Bien que V2 soit environ deux fois plus rapide, le temps de calcul de V4 reste suffisamment faible pour être intégré sans impact sur l'expérience utilisateur, d'autant plus que le placement des bateaux n'est effectué qu'une seule fois par partie. Nous avons donc retenu la version 4 pour l'implémentation finale.

2.6 Robustesse du code

Dès le début du projet, assurer la robustesse du code a été une de nos priorités. Nous nous sommes donc basés sur le principe de "Never Trust User Input", qui consiste à anticiper le fait que l'utilisateur puisse entrer des données non conformes aux attentes du programme.

Pour ce faire, nous avons implémenté un système de validation en plusieurs étapes. Tout d'abord, le type de chaque valeur entrée est vérifié. Si celui-ci est incorrect, le jeu affiche un message d'erreur. Dans le cas contraire, le code vérifie que le format correspond aux attentes et le normalise automatiquement si nécessaire (retirer les espaces indésirables avec *strip* ou encore transformer en majuscule avec *upper*). Concernant la saisie des coordonnées, nous avons adopté la notation classique de la bataille navale (ex : A9) afin de rendre le jeu plus fluide et intuitif pour les joueurs familiers avec ce format.

Ensuite, pour le placement des bateaux, le programme à besoin du fichier *weight_map.npy* dont le chemin est actuellement codé en dur, ce qui peut soulever des erreurs si le jeu n'est pas lancé depuis le bon endroit. Nous avons donc placé l'import dans un *try/except* et avons rendu le fichier externe facultatif pour placer les bateaux. Finalement, les seules erreurs que nous rencontrions encore étaient liées aux bibliothèques python. C'est pourquoi nous avons encapsulé l'appel de la fonction *main* dans un bloc *try/except* qui affiche le message "*An error has occurred, please check game requirements*". Bien que fonctionnelle, cette approche reste générique et pourrait être améliorée. La version 2 du jeu propose d'ailleurs une gestion d'erreur plus complète.

VERSION II

3 Version 2

3.1 Présentation du jeu

Dans cette version, le joueur et l'ordinateur s'affrontent l'un contre l'autre. Tout d'abord, le joueur rentre son pseudo ainsi que le niveau difficulté souhaité (**easy** ou **hard**) correspondant à l'intelligence de son adversaire. Il place ensuite ses bateaux sur une grille, puis l'ordinateur fait de même, à l'abri des yeux du joueur. Ensuite, chacun attaque à tour de rôle. Bien entendu, toutes les règles et mécanismes établis dans la version 1 sont conservés afin d'assurer la cohérence et la fluidité du jeu. Les niveaux de difficulté **easy** et **hard** correspondent respectivement aux versions 2 et 3 décrites dans les consignes du devoir à rendre.

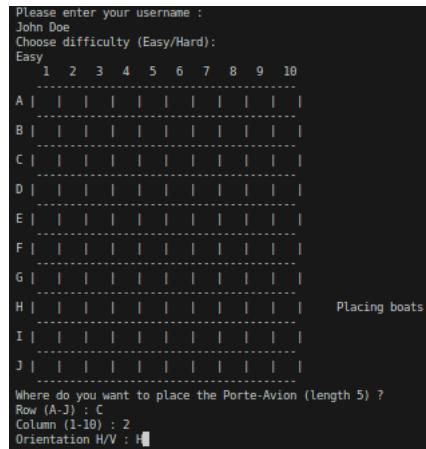


Image 1 : Placement des bateaux par l'utilisateur

```

1 2 3 4 5 6 7 8 9 10
A | | | | | | | | | B |
B | | | | | | | | | |
C | | P | P | P | P | | |
D | | | | | | | | | |
E | | | | | | | | | C |
F | | | | T | T | | | C |
G | | | | | | | | | C |
H | | S | | | | | | C |
I | | S | | | | | | |
J | | S | | | | | | |

Do you want to change a boat's position ? (Y/N) : 
  
```

Légende :
 P : Porte-avions
 C : Croiseur
 S : Sous-marin
 T : Torpilleur
 B : Barque

```

1 2 3 4 5 6 7 8 9 10
A | | | | | | | | | B |
B | | | | | | | | | |
C | | P | P | P | P | | |
D | | | | | | | | | |
E | | | | | | | | | C |
F | | | | T | T | | | C |
G | * | | | | | | | C |
H | | S | | | | | | C |
I | | S | | | | | | |
J | | S | | | | | | |

Enter the row to attack (A-J) : F
Enter the column to attack (1-10) : 
  
```

```

1 2 3 4 5 6 7 8 9 10
A | | | | | | | | | |
B | | | | | | | | | |
C | | | | | | | | | |
D | | | | | | | | | |
E | | | | * | | | | | |
F | | | | | | | | | |
G | | | | | | | | | |
H | | | | | | | | | |
I | | | | | | | | | |
J | | | | | | | | | |

Nombre de tirs : 1
Miss !
Terminator missed !
  
```

Image 3 : Placement des bateaux par l'utilisateur

3.2 Mise en œuvre du projet

Fichiers Relatifs

```
Battleship_Project/  
|   weightmap_boats.npy  
|  
+-- version_2/  
|   main.py  
|  
|   players.py  
|  
|   interface.py  
|  
|   database.py  
|  
|   exceptions.py  
|  
+-- scoreboard_battleship.csv
```

Dépendances logicielles

Le jeu nécessite l'installation de python avec une version entre 3.11 et 3.14

La seule bibliothèque à installer au préalable nécessaire pour lancer cette version est **numpy** en version 2.3.4 ou plus.

Lancement du jeu

Pour lancer le jeu, il suffit d'exécuter la commande suivante depuis le répertoire *Project Battleship* contenant les fichiers ci-dessus :

Commande de lancement Linux / macOS

```
python3 version_2/main.py
```

Commande de lancement Windows

```
py version_2/main.py
```

3.3 Structures de donnée

Dans cette partie, les structures expliquées à la Section 2.3 sont conservées.

Dans cette version, nous avons utilisé des classes Python pour représenter les différents types de joueurs et gérer les informations de jeu. Les classes permettent d'organiser les données et les comportements associés, et facilitent l'extension du programme.

Nous avons défini une classe parent **BasePlayer** et deux classes enfants : **Player** pour le joueur humain et **Bot** pour l'ordinateur.

- **Classe parent**¹ : **BasePlayer** contient les attributs et méthodes communs à tous les joueurs, tels que :
 - name : nom du joueur
 - boats_left : nombre de bateaux encore à flot
 - grid : grille de jeu (10x10)
 - boat_coords : dictionnaire des coordonnées des bateaux

1. Une classe parent (ou super-classe) est une classe dont héritent d'autres classes, appelées classes enfants. Elle contient des attributs et méthodes communs aux classes enfants.

- `register_attack()` : méthode pour enregistrer une attaque
- `check_sink()` : vérifie si un bateau est coulé
- **Classes enfants²** : **Player** et **Bot** héritent de **BasePlayer** et ajoutent des fonctionnalités spécifiques :
 - **Player** :
 - `score` : nombre de tentatives valides
 - `place_boat()` : placer un bateau sur la grille
 - **Bot** :
 - `difficulty` : niveau de difficulté du bot
 - `hit_coords` : coordonnées des bateaux touchés mais non coulés
 - Placement et attaque automatiques avec algorithmes aléatoires et intelligents

3.4 Intelligence Artificielle de l'Adversaire

Le jeu propose deux niveaux de difficulté pour l'ordinateur, se distinguant par leur stratégie d'attaque.

3.4.1 Mode Facile : Attaque Aléatoire

Le bot sélectionne aléatoirement une case parmi toutes les cases non encore attaquées. Chaque tir est indépendant des précédents, sans mémorisation ni exploitation des informations obtenues.

3.4.2 Mode Difficile : Stratégie Adaptative

Le mode difficile implémente une intelligence capable de mémoriser et d'exploiter les coups réussis pour cibler méthodiquement les bateaux.

Phase de recherche Lorsqu'aucun bateau n'est actuellement ciblé, le bot attaque aléatoirement comme en mode facile. Dès qu'un bateau est touché, le bot mémorise cette coordonnée et active sa stratégie d'élimination.

Détermination de l'orientation Une fois un bateau touché, le bot doit identifier son orientation (horizontale ou verticale) pour l'attaquer efficacement. Deux cas se présentent :

Cas 1 : Une seule case touchée. Le bot ne connaît pas encore l'orientation du bateau. Il analyse alors les cases adjacentes (haut, bas, gauche, droite) pour estimer quelle direction est la plus prometteuse :

- Si le bateau est en bordure de grille, certaines directions sont impossibles et sont éliminées.
- Le bot compte le nombre de cases disponibles (non encore attaquées) dans chaque direction.
- Il choisit l'orientation offrant le plus de possibilités. Par exemple, si les cases à gauche et droite (horizontal) sont disponibles mais que pour la verticale, seule le bas est disponible, le bot privilégie l'orientation horizontale.

Cas 2 : Plusieurs cases touchées. L'orientation devient évidente par l'alignement des coups réussis. Si deux cases touchées ont la même ligne, le bateau est horizontal; si elles ont la même colonne, il est vertical

Élimination systématique Une fois l'orientation déterminée, le bot reconstruit mentalement la ligne formée par toutes les cases touchées du bateau. Il identifie ensuite les deux extrémités de cette ligne et tente d'attaquer ces positions :

1. **Attaque de l'extrémité inférieure/gauche** : Le bot vérifie si la case située juste avant le début de la ligne est disponible (non attaquée et dans la grille). Si oui, il l'attaque en priorité.
2. **Attaque de l'extrémité supérieure/droite** : Si la première extrémité n'est pas disponible (déjà attaquée, en bordure, ou coup raté), le bot cible l'autre extrémité.

Cette stratégie garantit que chaque coup porte soit sur une case du bateau, soit révèle sa fin, permettant au bot de passer rapidement à l'autre extrémité.

2. Une classe enfant (ou sous-classe) hérite d'une classe parent et peut ajouter ou modifier des attributs et méthodes spécifiques.

Gestion des cas particuliers Si aucune des deux extrémités n'est disponible dans l'orientation choisie (situation rare où le bot s'est trompé d'orientation initiale, par exemple si plusieurs bateaux sont collés), le bot change automatiquement d'orientation et réessaie. Si l'échec persiste, il abandonne temporairement ce bateau et reprend une recherche aléatoire.

Mémoire et réinitialisation Le bot maintient en mémoire toutes les coordonnées des cases touchées mais dont le bateau n'est pas encore coulé. Lorsqu'un bateau est entièrement coulé (marqué par des "X"), le bot supprime automatiquement ses coordonnées de la mémoire et reprend sa phase de recherche pour le bateau suivant.

3.4.3 Comparaison des Performances

Pour comparer les niveaux de difficultés, nous avons simulé 5000 parties où le bot joue seul. Vous pouvez retrouver ces analyses dans le notebook *bot_AI_testing.ipynb*.

TABLE 4 – Comparaison des deux niveaux de difficulté (5 000 simulations)

Métrique	Mode Facile	Mode Difficile	Amélioration
Coups moyens	94,7	69,1	-27,0%
Minimum	52	24	-53,8%
Maximum	100	100	0%

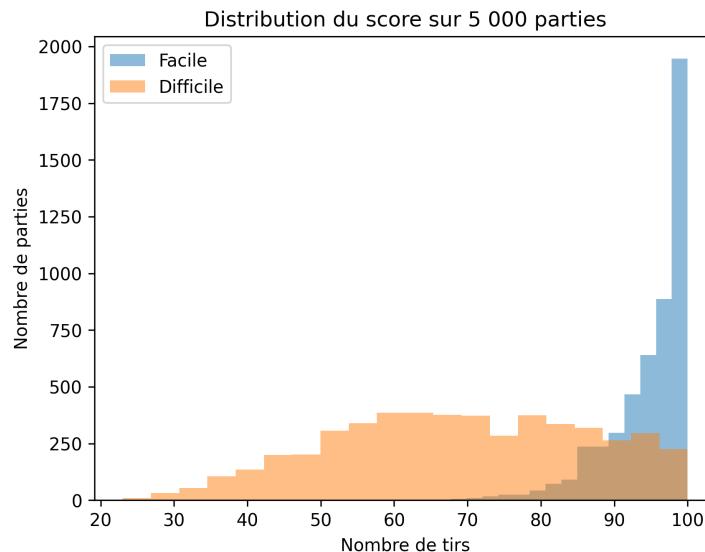


FIGURE 11 – Histogramme comparatif du nombre de coups moyens pour terminer une partie

Comme nous pouvions l'imaginer, le mode difficile a de bien meilleurs résultats, mais nécessitent tout de même parfois 100 coups pour trouver tous les bateaux, sans doute à cause de la barque qui ne fait qu'une case de long.

3.5 Fonctionnalités Additionnelles

Tableau de scores

Les scores des joueurs sont enregistrés dans un fichier CSV appelé **score.csv**. Chaque ligne du fichier contient :

- le nom du joueur
- son score (nombre de tentatives valides)

Chaque nouvel ajout de score se fait automatiquement dans ce fichier.

Pseudonyme

Pour rendre le jeu plus "vivant" et nous permettre de mettre à jour le tableau des scores, nous avons permis au joueur de choisir un pseudonyme. Nous avons également permis au bot d'en choisir un parmi une liste que vous pourrez retrouver dans le fichier `database.py`

ASCII Art

Enfin, pour marquer la fin du jeu autrement qu'avec un simple `print`, nous avons utilisé du ASCII art pour faire le bilan de la partie. Vous pouvez les retrouver dans le fichier `interface.py`, mais pour apprécier cette fonctionnalité au maximum il faudra faire une partie entière ;P

3.6 Optimisation

Dans la version 2, nous avons bien évidemment conservé les optimisations présentes dans la version 1 (voir 2.3 Optimisation), mais nous avons écrit plus de code et implémenté de nouveaux algorithmes.

La seule chose que nous avons optimisé est la répartition du code en séparant les fonctions dans des fichiers selon leur utilité. Nous avons également séparé le frontend et le backend : la fonction `main` regroupe toutes les interactions avec l'utilisateur (autre que celles dans la fonction `game_init`), et les fonctions d'affichage sont stockées dans le fichier `interface.py`. Le backend, c'est-à-dire l'algorithme, est stocké dans des fonctions et fichiers à part. Cela nous permet d'isoler les blocs d'algorithmes, nous permettant de pouvoir y accéder facilement et les réutiliser à d'autres endroits.

3.7 Robustesse du code

Cette version plus complète du jeu introduit de nouveaux risques d'erreurs liés aux algorithmes avancés et aux dépendances entre fichiers. Plusieurs mécanismes ont été mis en place pour garantir la stabilité du programme. Tout d'abord, nous avons conservé le principe du "*Never Trust User Input*" établi en version 1, assurant une validation stricte de toutes les entrées utilisateur.

Gestion des fichiers externes

Le code nécessite deux fichiers externes : `weightmap_boats.npy` et `scoreboard_battleship.csv`. Contrairement à la version précédente où les chemins étaient codés en dur, nous avons implémenté un système de recherche dynamique via deux fonctions dans `database.py`. Ces fonctions parcourent l'arborescence depuis le point de lancement du code pour localiser les fichiers nécessaires.

Pour le tableau des scores, si le fichier est introuvable, le code cherche le dossier parent `version_2` et crée le fichier manquant. Si le dossier lui-même n'existe pas, cette fonctionnalité est désactivée sans interrompre le jeu. La carte de poids suit une logique similaire : en cas d'absence, l'algorithme de placement bascule sur un mode aléatoire classique. Cette approche permet de lancer le jeu depuis n'importe quel répertoire parent de *Battleship Project* sans configuration préalable.

Protection contre les boucles infinies

La boucle principale où joueur et bot s'affrontent a été sécurisée par un compteur d'échecs consécutifs. Si ce compteur dépasse un seuil critique, le programme interrompt l'exécution et soulève une erreur personnalisée `BotLoopError`, évitant ainsi tout blocage indéfini. De manière générale, nous avons vérifié l'ensemble des boucles du code pour garantir qu'aucune ne puisse boucler infiniment.

**VERSION III
INTERFACE GRAPHIQUE**



4 Version 3

Pour l'interface graphique, nous avons choisi d'utiliser la bibliothèque Pygame. Bien que nous ayons déjà eu la possibilité d'utiliser Tkinter, que nous maîtrisions davantage, nous avons décidé de nous tourner vers une bibliothèque que nous ne connaissons pas afin d'apprendre à manier de nouveaux outils : Pygame.

La conception de l'interface a commencé par la création des boutons tactiles, suivie de l'intégration des images, du choix des couleurs des bateaux et de l'ajout d'effets sonores. Ces éléments nous ont permis de rendre le jeu plus interactif et visuellement plus attractif.



Page d'accueil de l'interface

4.1 Mise en œuvre du projet

Fichiers Relatifs

```
Battleship_Project/
└── weightmap_boats.npy
└── version_3/
    ├── block.py
    ├── bouton_transparent.py
    ├── button.py
    ├── classes.py
    ├── colors.py
    ├── deuxieme_grille.py
    ├── grid.py
    ├── main.py
    ├── position.py
    ├── scoreboard.py
    ├── settings_menu.py
    ├── username.py
    └── ressources/
        ├── large-underwater-explosion-sfx.mp3
        ├── Logo.png
        ├── ocean waves GIF by weinventyou.gif
        ├── scoreboard_battleship.csv
        ├── Tobi_Candyland.mp3
        ├── water-splash.mp3
        └── weightmap_boats.npy
```

Dépendances logicielles

Le jeu a été développé en **Python** et nécessite une version comprise entre **3.11 et 3.14**. Avant de lancer le programme, certaines bibliothèques doivent être installées afin d'assurer son bon fonctionnement.

- **pygame** : gestion de l'affichage, des événements et des sons, en version 2.5.2 ou plus.
- **pygame-gui** : interface graphique (saisie du nom du joueur), en version 0.6.9 ou plus.
- **pillow** : traitement et affichage des images GIF, en version 10 ou plus.

L'installation des dépendances peut être effectuée à l'aide de la commande suivante :

```
pip install pygame pygame-gui pillow
```

Lancement du jeu

Décompresser le dossier `Battleship_Project.zip`, ouvrir un terminal dans le dossier `Battleship_Project`, puis exécuter la commande correspondant à votre système d'exploitation.

Commande de lancement Linux / macOS

```
python3 main.py
```

Commande de lancement Windows

```
py main.py
```

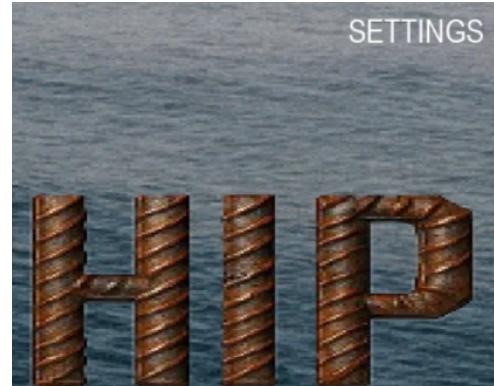
IMPORTANT

Les images, musiques et fichiers Python doivent impérativement rester dans le dossier `battle_ship`. Toute modification de l'arborescence empêchera le bon fonctionnement du jeu.

4.2 Fonctionnalités importante de Pygame

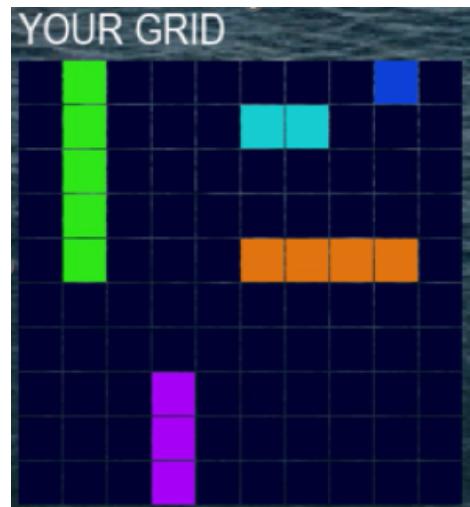
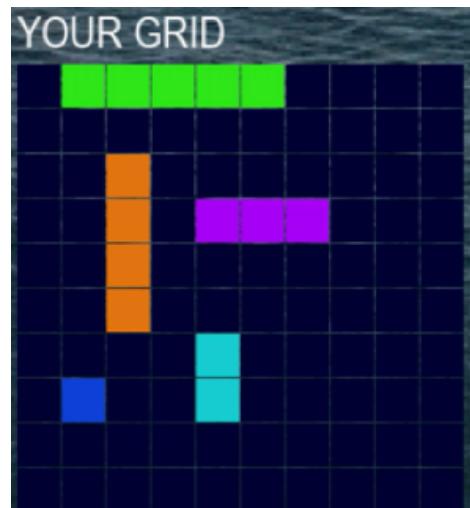
Ce qui rend notre projet unique est l'attention portée à l'expérience utilisateur, avec une gestion intuitive du clavier pour le placement des bateaux et une organisation claire des couleurs garantissant une lecture visuelle immédiate de l'état du jeu.

Dès le lancement du jeu, l'utilisateur accède au menu principal. Un bouton `Settings`, situé en haut à droite de l'écran, est disponible à tout moment, aussi bien dans le menu principal que durant une partie. Ce bouton permet de régler le volume de la musique ainsi que celui des effets sonores, notamment lors des tirs réussis ou manqués. Il offre également la possibilité de modifier la langue du jeu.



Pygame permet de détecter les touches du clavier à l'aide de constantes prédefinies. La touche R, représentée par `pygame.K_r`, est utilisée lors de la phase de placement des bateaux.

Elle permet de changer l'orientation du bateau en alternant entre une position horizontale et verticale. Pour cela, l'utilisateur doit d'abord appuyer sur la touche R, puis cliquer sur la grille afin de placer le bateau dans la nouvelle orientation.



L'interface graphique utilise un code couleur pour distinguer les différents états du jeu : l'eau (bleu foncé), les différents types de bateaux (vert, orange, violet, cyan, bleu), ainsi que les tirs réussis (rouge vif) et ratés (rouge sombre). Ces couleurs sont regroupées dans une classe dédiée pour faciliter la gestion visuelle et assurer la cohérence graphique. Une légende est affichée à l'écran pour permettre au joueur d'identifier rapidement chaque élément.



4.3 Difficultés rencontrées

Nous avons rencontré plusieurs difficultés, notamment concernant l'intégration d'un GIF. Au départ, nous souhaitions insérer une vidéo, mais cela n'a pas fonctionné. Nous avons donc trouvé un site permettant de convertir la vidéo en GIF.

D'autres difficultés concernaient le positionnement des textes et des boutons. Nous avons utilisé l'IA uniquement pour le visuel de l'interface graphique (la position, la taille, etc...), le reste du code a été écrit à la main.

4.4 Ressources

Les ressources extérieures utilisées sont :

- Logo + Image page 1 : Généré par Copilot.
- GIF by weinventyou GIF on Giphy.com
- Effets sonores : La musique Tobu-Candyland qui est libre de droits d'auteur, et les effets sonores des tirs et des ratés.

5 Conclusion

Pour conclure, ce projet a été très enrichissant. Le développement des différentes versions, allant de la plus simple à la plus complète et structurée, nous a permis de découvrir et d'explorer de nombreuses fonctionnalités de Python, tout en approfondissant nos connaissances. Nous remercions M. de Vaulx pour son encadrement tout au long de ce semestre, sa disponibilité et son écoute, qui nous ont permis de mener à bien la réalisation de ce projet.

6 Références

1. https://youtu.be/nF_crEttmpBo?si=YVGBndssELwuOdR-
2. <https://youtu.be/GMBqjxcKogA?si=c08U2XhvJDSwMwGD>
3. <https://youtu.be/kDSDjsdoGOY?si=JsI4dKtW7d16aqKC>
4. <https://pypi.org/project/gif-pygame/>