

VALIANT



Compte Rendu

Laurent Raphaël x Beaujard Traïan

Sommaire :

Sommaire :	2
I/Introduction :.....	4
II/Analyse du Mini-Projet :.....	4
A/Cahier des charges.....	4
B/Brainstorming du projet.....	4
1/Diagramme de déploiement.....	6
2/Diagramme des cas d'utilisations.....	7
3/Diagramme d'exigences.....	8
4/Diagramme de séquences.....	9
5/Diagramme des états-transitions.....	10
6/Répartitions des tâches.....	11
7/Outils utilisés.....	12
III/Journal de L.Raphaël :.....	13
1. Planification du Jeu.....	13
2. Configuration de l'Environnement de Développement.....	13
3. Création des Structures de Données.....	13
4. Logique de Placement des Navires.....	13
5. Développement du Moteur de Jeu.....	14
6. Gestion des Tours.....	14
7. Fin de Jeu et Victoire.....	14
8. Tests et Débogage.....	14
7) Création de l'infrastructure:.....	17
8) L'implémentation d'une "IA":.....	19
9) Le placement des bateaux:.....	21
Analyse de la Méthode.....	24
10) La boucle du jeu:.....	25
11) Actualiser la grille du jeu:.....	27
12) Saisie du joueur:.....	29
13) gérer les attaques:.....	31
14) Historique des problèmes rencontré:.....	33
15) La sérialisation:.....	36
Analyse de la Méthode.....	40
Conclusion.....	41
Réalisation de test unitaire Catch2 :	41
16) Mise en place de la fonctionnalité multi Joueurs:.....	43
Méthode CoreGame::envoyerAttaque(int ligne, int colonne).....	49
Méthode TCPWinsocksMaster::sendMessage(std::string msg).....	49
Conclusion.....	49
Méthode CoreGame::recevoirAttaque().....	51
Méthode TCPWinsocksMaster::receiveMessage().....	51

Conclusion.....	51
17) La base de données:.....	52
Bataille Navale avec Extension Web.....	55
Bataille Navale en C++ avec Surcouche Graphique SFML.....	58
Introduction.....	58
Surcouche Graphique SFML.....	58
Mode Multijoueur.....	58
Système de Comptes et Statistiques.....	58
Boutique d'Items.....	58
Expérience Utilisateur Améliorée.....	58
Exemple de la création d'une image pour le SFML :.....	59
Création de l'images d'accueil :.....	66
Affichage des items :.....	66
Structure et Organisation.....	69
Logique de Placement.....	69
Conclusion.....	70
Conclusion.....	75
IV/Journal de B.Traian :.....	75
A/Création d'une classe chargement de Json.....	76
B/Création de la base réseau.....	77
1/Structure réseau.....	77
2/Fonctionnement de l'acceptation d'un client.....	78
3/Fonctionnement du matchmaking.....	80
4/Code pour les messages.....	82
5/Fonctionnement d'une session de jeux.....	82
C/Création base graphique.....	85
1/Éviter le hard coding et les nombres magiques.....	85
2/Les fenêtres.....	86
3/Les éléments graphiques.....	87
D/Intégration Jeu au graphique.....	88
E/Ajouts de features graphiques.....	88
F/Données de joueurs.....	89
H/Chat de discussion.....	89
I/Partie Physique : Fonctionnement d'un thread au niveau CPU.....	90
V/Conclusion :.....	91

I/Introduction :

Dans le cadre de notre BTS SNIR, nous devons réaliser un mini-projet entre le lundi 20 novembre et le 22 décembre 2023. Nous avons choisi de réaliser un jeu de bataille navale en équipe de 2.

Dans ce rapport, le cahier des charges sera analysé et complété. Ensuite il y aura le journal de bord de Laurent Raphaël suivi du journal de bord de Beaujard Traïan.

→ Le code du projet est disponible sur [GitHub](#).

→ Le jeu peut être téléchargé depuis [Itch.io](#) ou via le site web officiel [stein-ind](#).

II/Analyse du Mini-Projet :

A/Cahier des charges

Lors de la réalisation d'un projet, il est important de comprendre le cahier des charges. Ici, il nous a été donné. Nous devons délivrer un produit reprenant les points suivants :

- Un jeu de bataille navale multijoueur en 1 contre 1.
- Chaque client doit dialoguer avec l'autre client via un serveur.
- Le client et le serveur auront une classe commune.
- Le serveur doit être un serveur concurrent, il devra écouter sur le port 12345.
- Le serveur utilise les threads pour gérer les parties entre les joueurs.
- Boucle de jeu de bataille navale fonctionnel

B/Brainstorming du projet

Avant de commencer, nous devons étudier comment répondre au cahier des charges. De plus, le client nous a autorisé à améliorer ce cahier des charges tant que ce dernier est complètement répondu. Nous avons donc réfléchi aux nouvelles fonctionnalités que nous voulions.

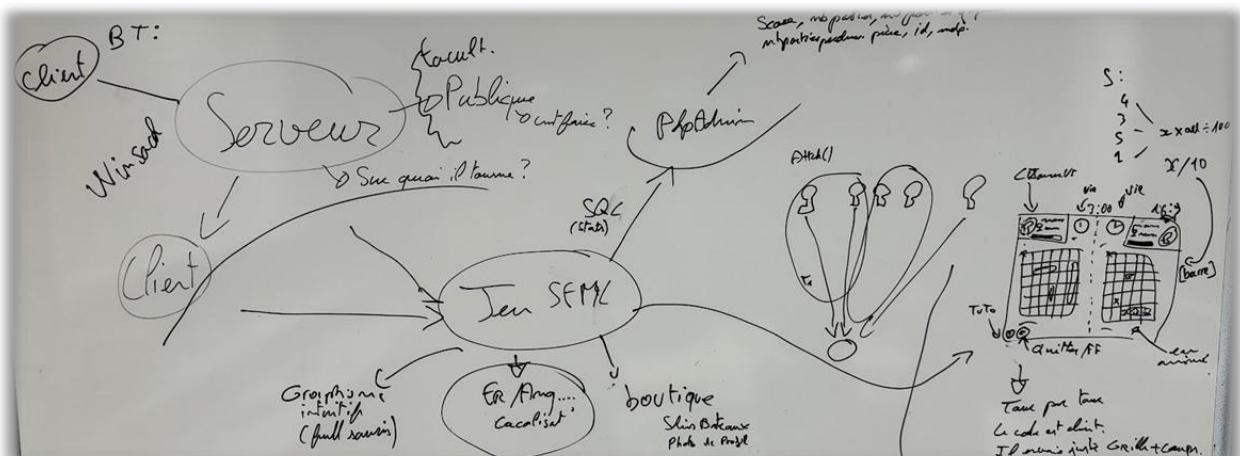


Figure 1 : Idées de fonctionnalités

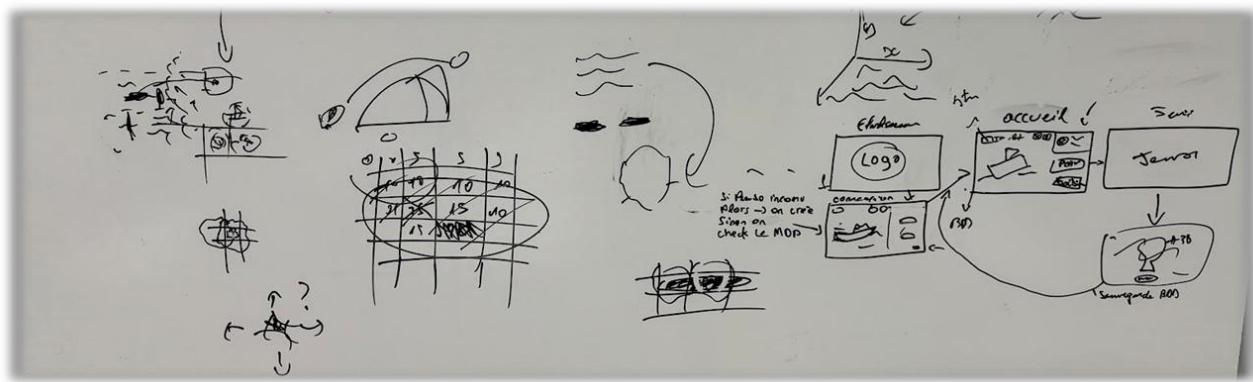


Figure 2 : Idées graphiques

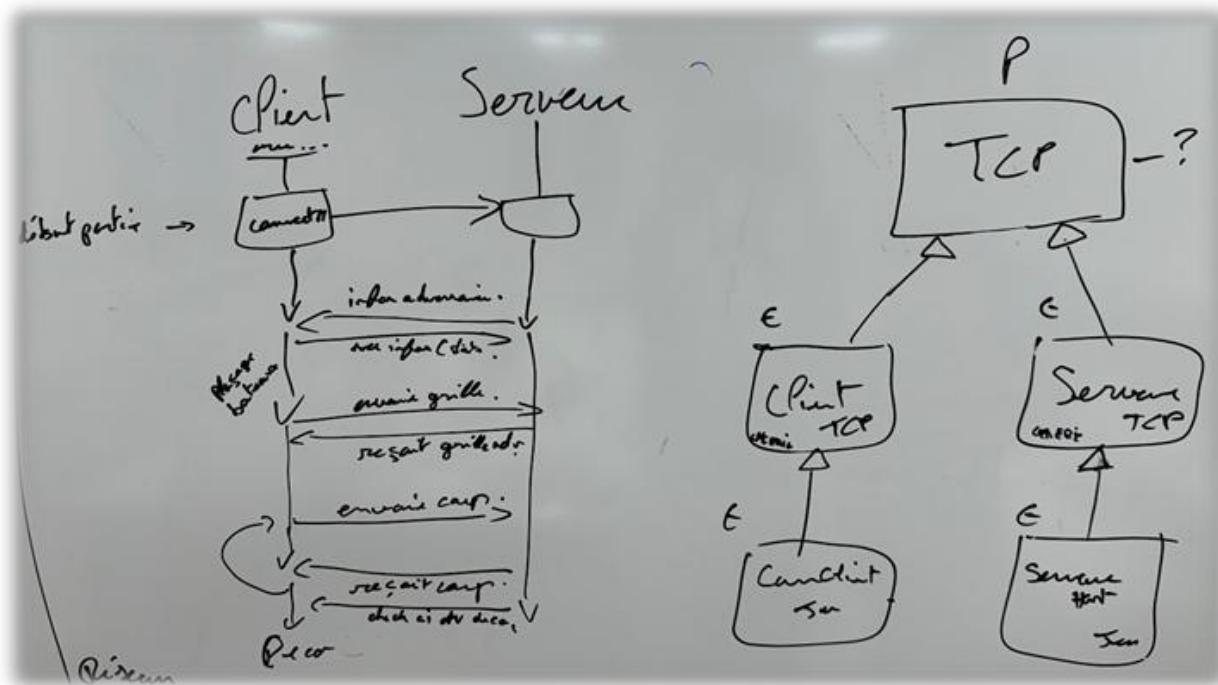


Figure 3 : Idées graphiques

C/Cahier des charges++

Après cette réflexion, nous avons décidé d'ajouter de nouvelles obligations dans le cahier des charges :

- Le joueur pourra se connecter à une base de données afin de sauvegarder ses statistiques de jeux et son score.
- Le joueur pourra aller sur un site web pour regarder ses données ou celles d'un autre joueur via son pseudo.
- Le jeu aura une couche graphique complète. C'est à dire qu'il y doit y avoir :
 - Un Menu d'accueil pour lancer le matchmaking.
 - Un Menu de jeu pour placer les bateaux et attaquer la cible.
- La partie en jeu intégrera un chat de discussion entre les 2 joueurs pour communiquer.
- Le joueur aura accès à un magasin pour personnaliser son avatar et/ou ses bateaux.

D/Études en Diagrammes

1/Diagramme de déploiement

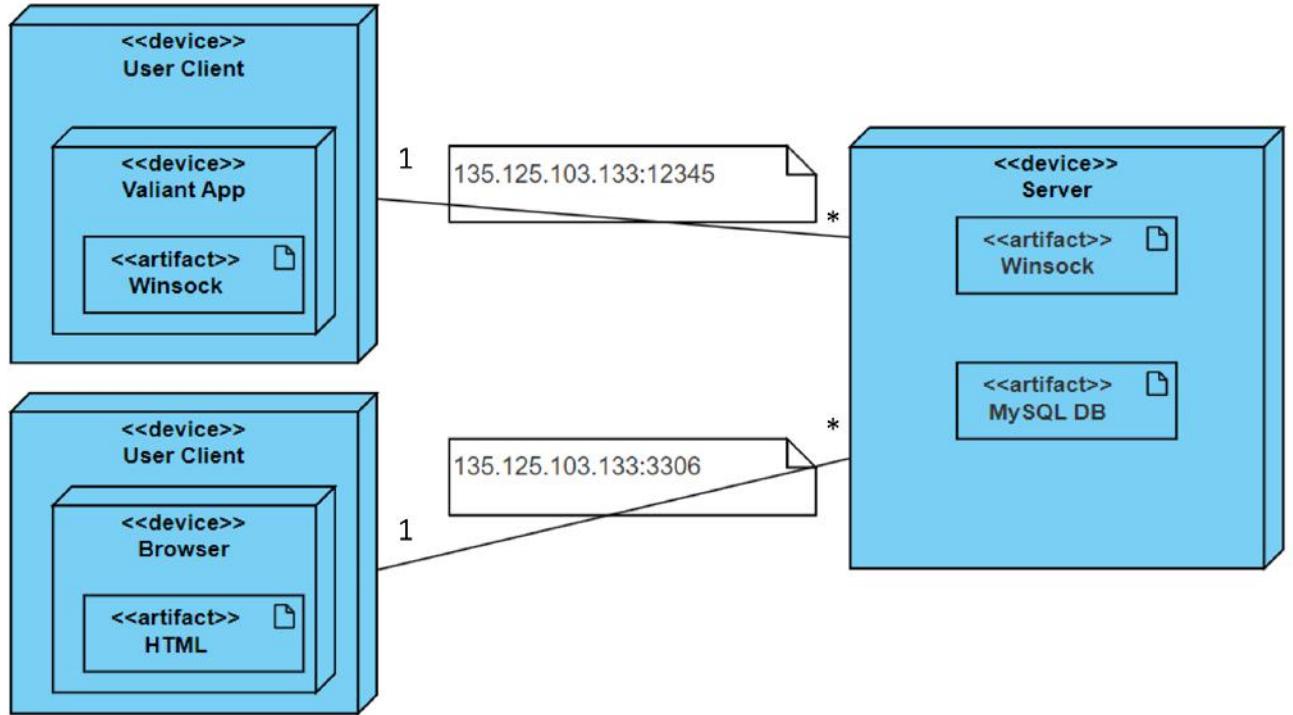


Figure 4 : Diagramme de déploiement

Le client (l'utilisateur) peut se connecter au serveur via l'application Valiant qui utilise Winsocks pour établir une connexion TCP au serveur. Le serveur écoute les clients du jeu sur son port 12345 comme demandé dans le cahier des charges. Le client peut également se connecter à la base de données SQL du serveur via une interface web qui se connectera au port 3306 du serveur.

2/Diagramme des cas d'utilisations

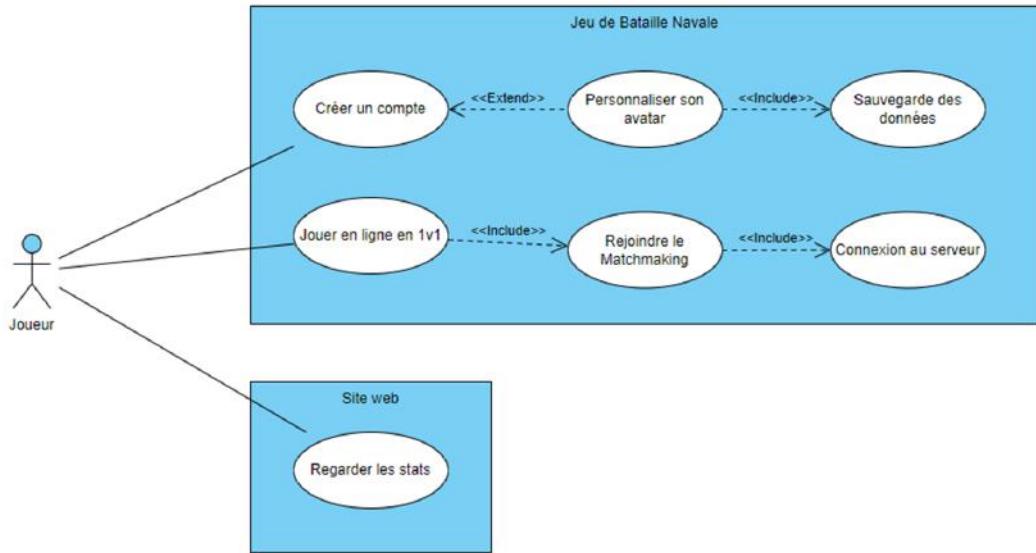


Figure 5 : Diagramme des cas d'utilisations

Il y a 1 acteur qui sera le joueur. Le système ne possède aucune option d'administration directement implantée. Le système est découpé entre le jeu et le site web.

Dans le jeu, le joueur a le choix de créer un compte, auquel cas il pourra sauvegarder ses changements d'avatar et ses données de jeu.

Il pourra évidemment jouer en ligne en rejoignant un matchmaking géré par le serveur.

Sur le site web, le joueur peut consulter ses statistiques.

3/Diagramme d'exigences

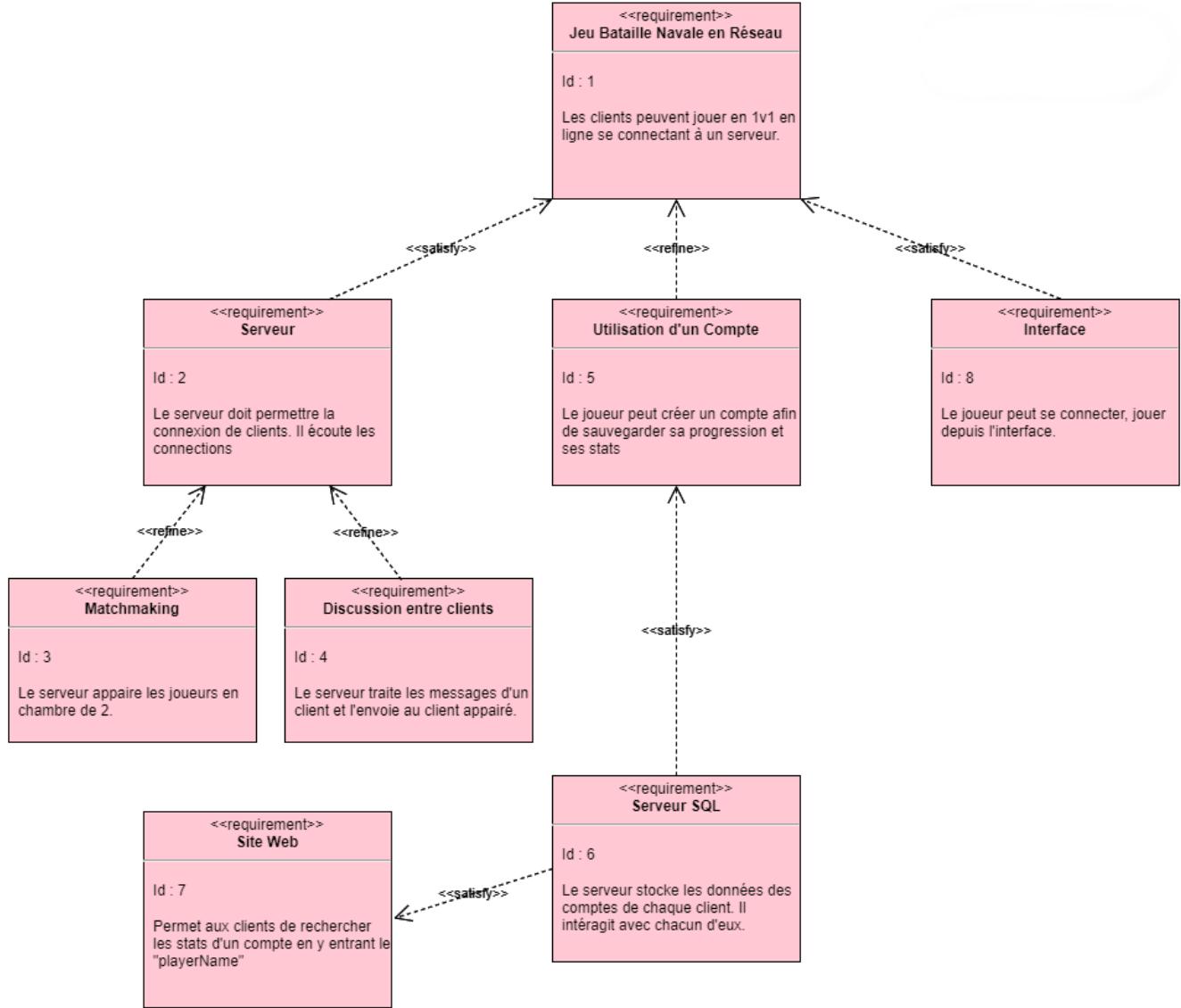


Figure 6 : Diagramme d'exigences

Le jeu doit compter un serveur qui gère le matchmaking et la discussion entre les clients. L'utilisateur peut avoir un compte sur la base de données SQL et doit pouvoir utiliser un interface pour se connecter et jouer.

4/Diagramme de séquences

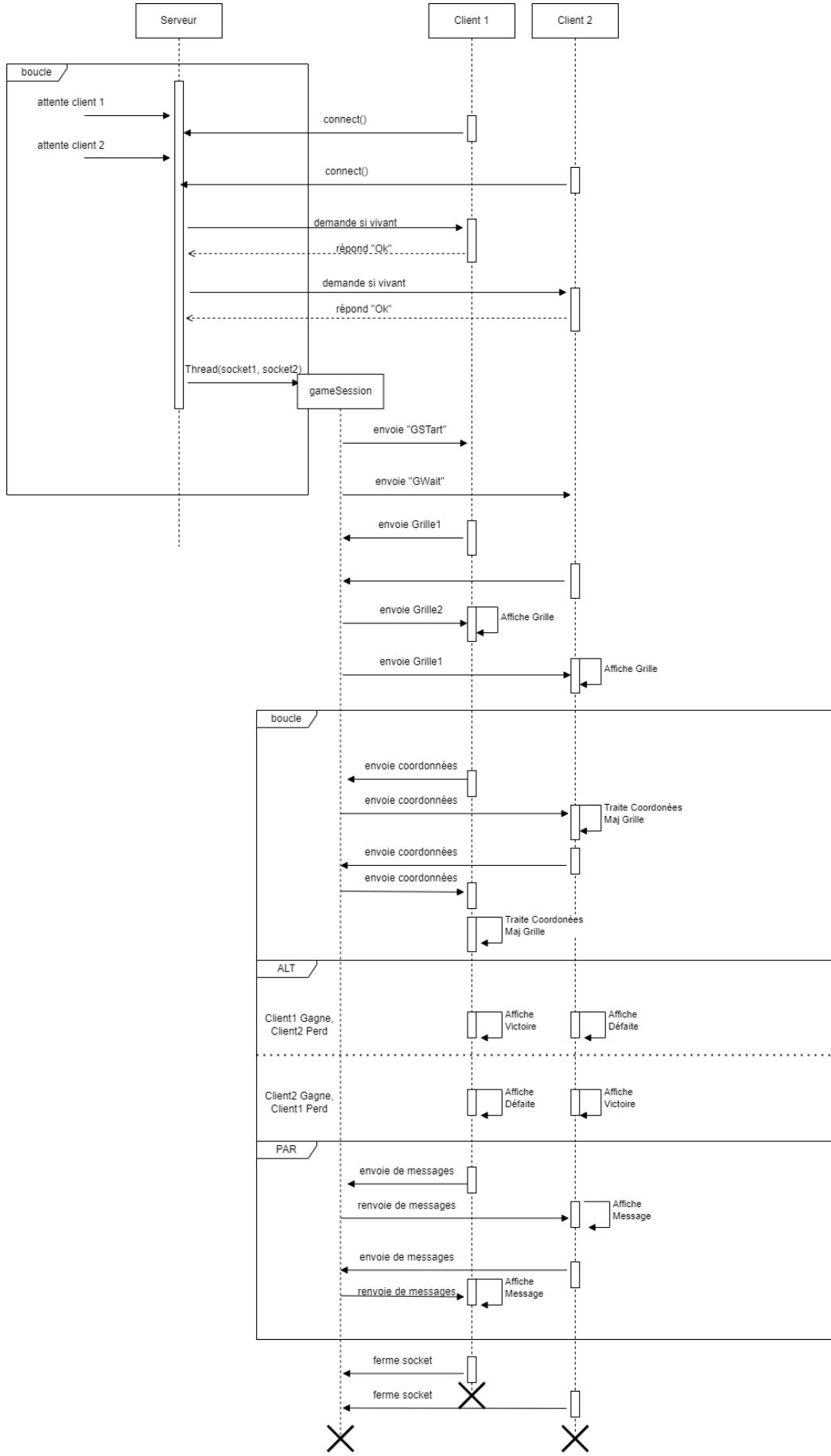


Figure 7: Diagramme de séquences au niveau réseau

5/Diagramme des états-transitions

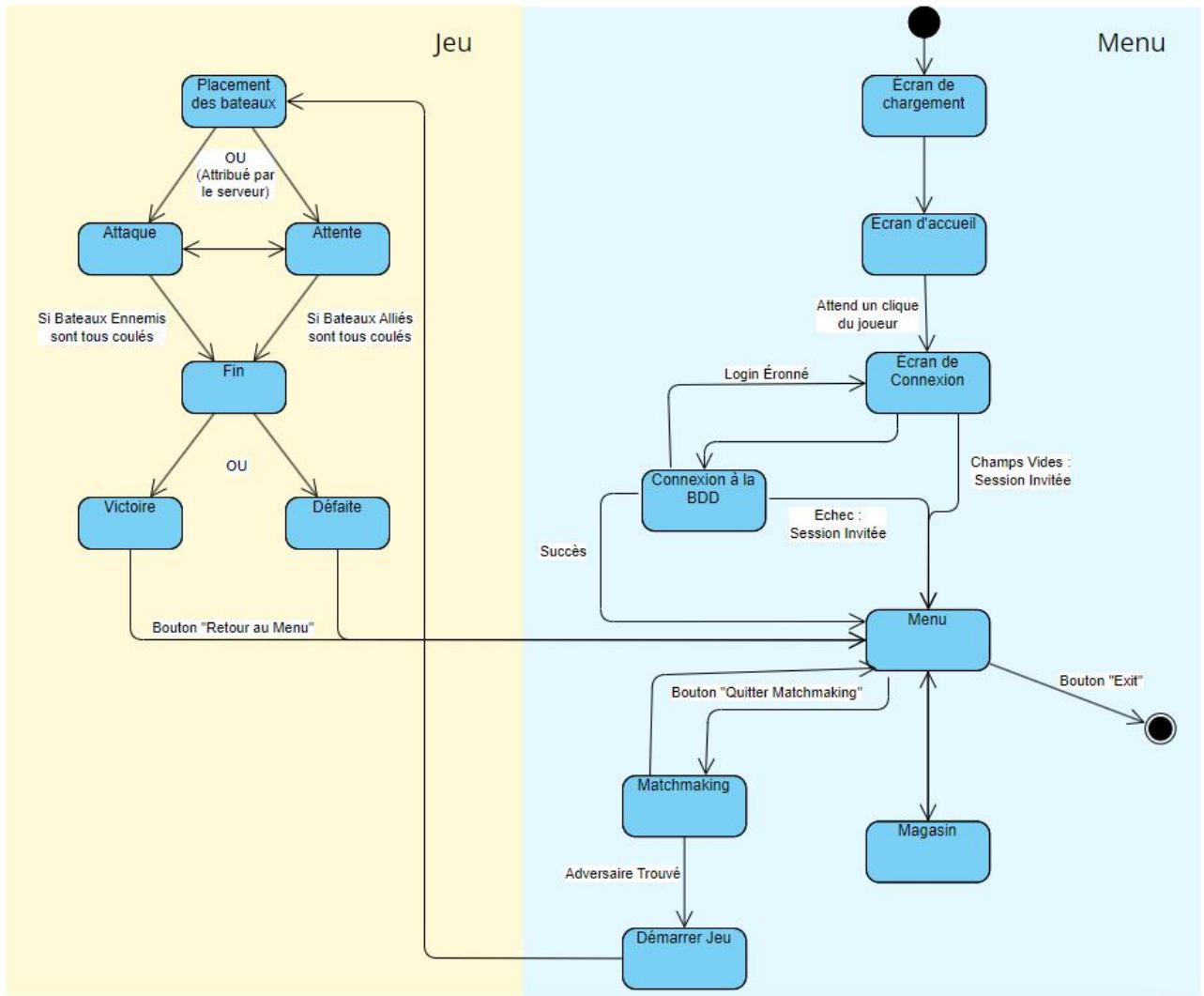


Figure 8 : Diagramme des états-transitions (machine state)

6/Répartitions des tâches

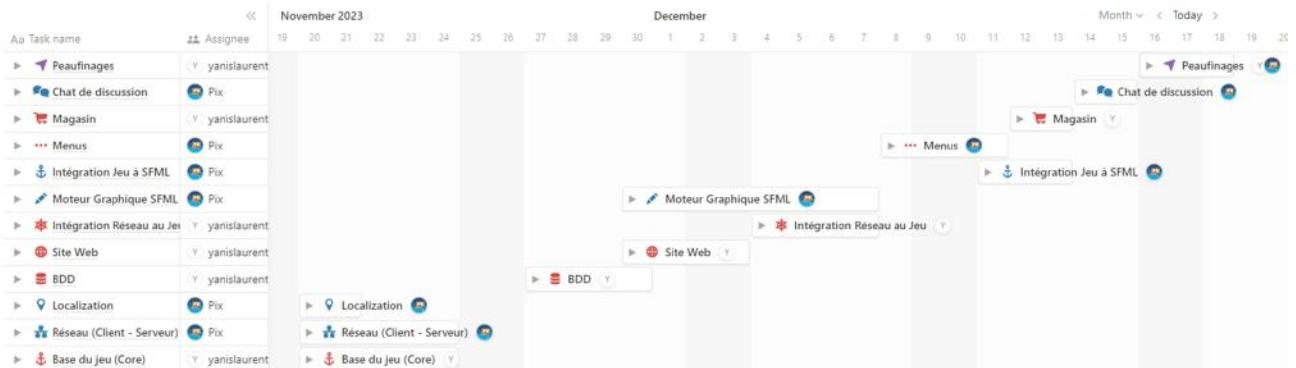


Figure 9 : Diagramme de Gantt (pix = Traïan B., yanis = Raphaël L.)

Nous nous sommes répartis les grandes tâches, c'était surtout important avant la semaine d'absence de Traïan B. Nous devions réaliser le coeur de jeu assez rapidement.

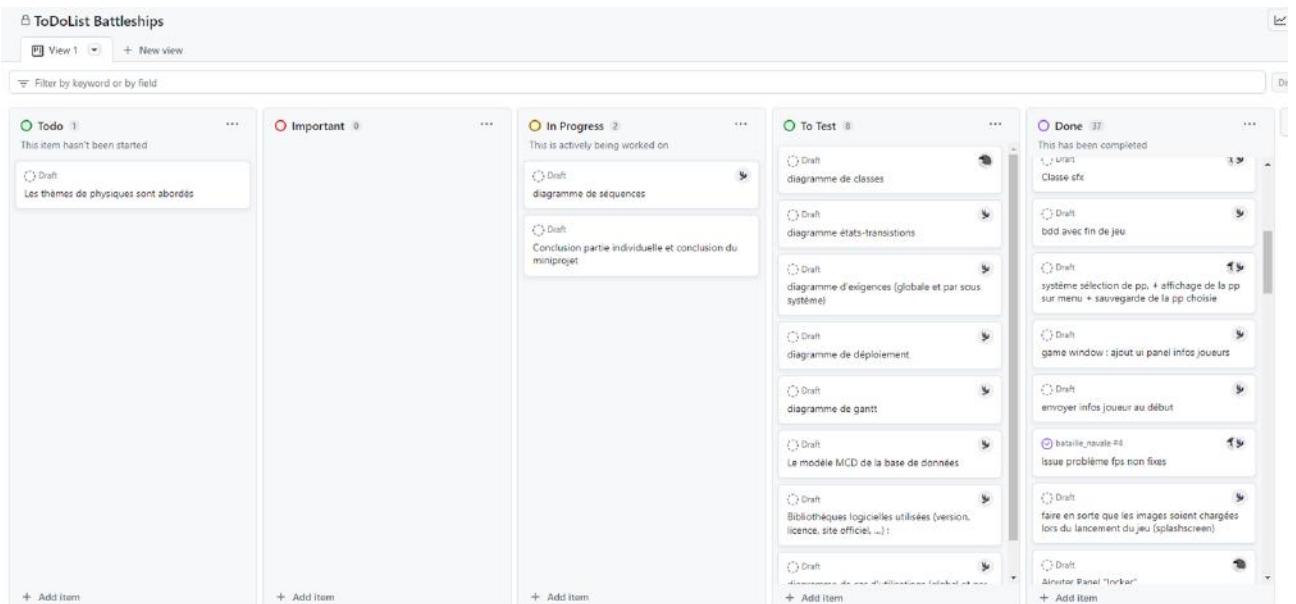


Figure 10 : Exemple de l'outils “projet” de github

Une fois les bases du jeu faites, nous avons utilisé cet outil de github où nous listions les tâches dans “Todo” et “Important”. Puis nous sélectionnions la tâche que nous voulions faire pour avancer le projet en la plaçant dans “In progress”. Quand elle était finie, nous la mettions dans “To test” afin que l'autre personne de l'équipe teste si la tâche était bien réalisée sans bugs.

7/Outils utilisés

- IDE :

- Visual Studio Community 2022 

- Libs :

- SFML 
- Winsocks
- MySql8.0 x Boost

- Organisation :

- Github 

- Autres :

- PhpMyAdmin 
- Adobe Photoshop & Adobe After Effects 
- Blender 
- Draw.io & Visual Paradigm

Figure 11 : Les outils du projet

Nous avons programmé sur l'IDE (interface development environment) Visual studio 2022 en c++.

Pour la partie graphique nous avons choisi la librairie SFML.

Pour la partie réseau nous devions utiliser Winsocks.

Pour la partie sql nous avons utilisé MySql 8.0 qui utilise la librairie Boost

Pour la gestion des tâches, le versionning, la séparation des fonctionnalités du jeu en branches nous utilisons Github.

Nous utilisons phpMyAdmin pour la base de données.

Adobe photoshop et after effects pour les images.

Blender pour créer les images d'eau animée et les explosions.

Draw.io et Visual Paradigm pour les diagrammes.

III/Journal de L.Raphaël :

Partie cœur du jeu :

A l'origine, on m'a attribué la tâche de faire ceci "**la mise en place des bateaux dans la grille**" tout en me rappelant que j'avais le libre arbitre quant à la manière dont j'allais concevoir le jeu.

Voici les étapes et les considérations clés que j'ai décidé de suivre pour créer la base du jeu en c++ :

1. Planification du Jeu

- **Conception des Règles** : Définir les règles de base, la taille de la grille, le nombre et les types de navires.
- **Interface Utilisateur** : Décider si le jeu sera en ligne de commande ou avec une interface graphique. (on a finalement fait les deux)

2. Configuration de l'Environnement de Développement

- **Choix des Outils** : Nous avons décidé de sélectionner Visual Studio comme compilateur C++ et environnement de développement intégré (IDE).

3. Création des Structures de Données

- **Grille de Jeu** : Utiliser un tableau 2D pour représenter la grille.
- **Navires** : Créer une classe `coreGame` avec des attributs spécifique
- **Joueur** : Créer une moyen autonome de riposter pour tester le jeu avant l'intégration en réseau, nous l'appellerions `ia` dans ce contexte précis.

4. Logique de Placement des Navires

- **Placement Manuel/Automatique** : Permettre au joueur de placer ses navires ou générer un placement aléatoire.
- **Vérification de la Validité** : S'assurer que les navires ne se chevauchent pas et restent dans les limites de la grille.

5. Développement du Moteur de Jeu

- **Boucle de Jeu** : Créer une boucle qui gère les tours des joueurs.
- **Gestion des Actions** : Implémenter la logique pour attaquer, vérifier les coups, et déterminer si un navire est coulé.

6. Gestion des Tours

- **Alternance des Joueurs** : Changer de joueur après chaque tour.
- **Feedback** : Informer les joueurs des résultats de leurs actions.

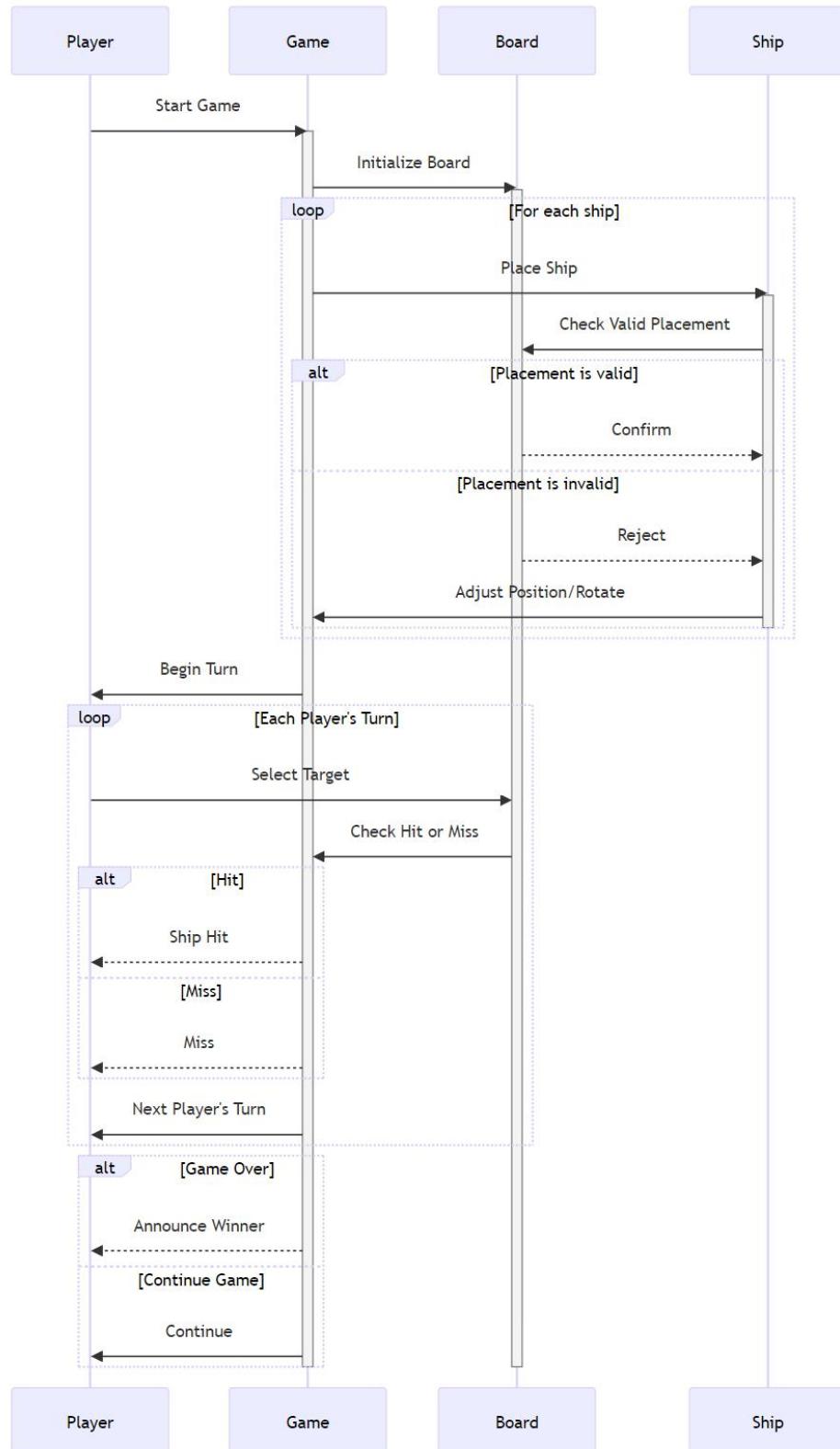
7. Fin de Jeu et Victoire

- **Conditions de Victoire** : Déterminer quand un joueur a gagné (par exemple, tous les navires adverses sont coulés).
- **Annonce des Résultats** : Afficher le gagnant et proposer de rejouer ou de quitter.

8. Tests et Débogage

- **Tests Unitaires** : Écrire des tests pour les différentes composantes et logiques du jeu.

Voici un diagramme de séquence illustrant le processus de création d'un jeu de bataille navale en C++ :



- 1) **Démarrage du jeu** : Le joueur initie le jeu.
- 2) **Initialisation du plateau** : Le jeu prépare le plateau de jeu.
- 3) **Placement des navires** : Pour chaque navire, le jeu vérifie si le placement est valide.
Si le placement est invalide, le navire est ajusté ou tourné.
- 4) **Début du tour** : Le jeu commence le tour du joueur.
- 5) **Tour de chaque joueur** : Le joueur choisit une cible, et le jeu vérifie si c'est un coup réussi ou manqué.
- 6) **Fin du jeu** : Le jeu se termine soit par l'annonce du gagnant, soit par la continuation pour le tour suivant.

7) Création de l'infrastructure:

Tout d'abord je crée le fichier header de ma classe qui s'appelle **CoreGame** ou (Le coeur du jeu en français), il représente la structure et l'idée de cette dernière:

```
#include <utility> // Pour std::pair
#include <string>

class typeCase {
    // Supposons que typeCase est une classe ou un struct déjà définie.
};

class CGrilleBatNavale {
public:
    static const int nbLig = 10;
    static const int nbCol = 10;

private:
    typeCase grille[nbLig][nbCol];

public:
    CGrilleBatNavale() {
        // Constructeur qui initialise la grille, par exemple avec des cases vides.
    }

    typeCase getCase(int ligne, int colonne) const {
        // Retourne la case de la grille à la position spécifiée.
        return grille[ligne][colonne];
    }

    void setCase(int ligne, int colonne, typeCase type) {
        // Définit la case de la grille à la position spécifiée avec le type donné.
        grille[ligne][colonne] = type;
    }

    void afficheGrille() const {
        // Affiche la grille dans la console ou l'interface utilisateur.
    }

    static std::pair<int, int> saisieJoueur() {
        // Permet au joueur de saisir une position sur la grille.
        return std::make_pair(0, 0); // À remplacer par la saisie réelle.
    }

    bool partiePerdu() const {
        // Détermine si la partie est perdue.
        return false; // À implémenter.
    }

    std::string serialisation() const {
        // Serialise l'état de la grille en chaîne de caractères.
        return ""; // À implémenter.
    }

    bool deserialisation(std::string trame) {
        // Deserialise l'état de la grille à partir d'une chaîne de caractères.
        return true; // À implémenter.
    }
}
```

ensuite je complète les méthodes dans le fichier de définition de ma classe.

Ce fichier implémentera les méthodes de la classe définies dans le fichier d'en-tête (.h)

Je crée un *main* de test. C'est un main de test assez basique et qui n'implémente pas de jeu complet. Il se contente de vérifier que les différentes parties de la classe fonctionnent comme prévu. Cela permet de voir la grille initiale, la position entrée par le joueur, et la grille sérialisée. La désérialisation est aussi appelée, mais puisque la méthode est vide, elle retourne simplement true.

```
#include "CoreGAME.h"
#include <iostream>

int main() {
    // Crée une grille de bataille navale.
    CoreGame grille;

    // Définit quelques cases avec des bateaux pour tester.
    grille.setCase(0, 0, CoreGame::typeCase::bateau);
    grille.setCase(1, 1, CoreGame::typeCase::bateau);
    grille.setCase(2, 2, CoreGame::typeCase::touche);
    grille.setCase(3, 3, CoreGame::typeCase::eau);

    // Affiche la grille.
    std::cout << "Grille initiale:" << std::endl;
    grille.afficheGrille();

    // Test de la saisie joueur.
    auto position = CoreGame::saisieJoueur();
    std::cout << "Position saisie par le joueur: "
        << position.first << ", " << position.second << std::endl;

    // Test pour vérifier si la partie est perdue.
    bool perdu = grille.partiePerdu();
    std::cout << "La partie est-elle perdue? " << (perdu ? "Oui" : "Non") << std::endl;

    // Test de la sérialisation.
    std::string grilleSerialisee = grille.serialisationAdversaire();
    std::cout << "Grille sérialisée:" << std::endl << grilleSerialisee << std::endl;

    // Test de la désérialisation.
    bool deserialisationReussie = grille.deserialisation(grilleSerialisee);
    std::cout << "Désérialisation réussie? " << (deserialisationReussie ? "Oui" : "Non") << std::endl;

    return 0;
}
```

Désormais j'avais une première grille, c'est un bon début:

La bataille Navale										
A	B	C	D	E	F	G	H	I	J	
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

8) L'implémentation d'une "IA":

Avant de pouvoir jouer avec un autre joueur par le biais d'une connexion *TCP*, il est nécessaire de pouvoir tester son jeu en local pour achever le codage du programme. Pour ce faire, rien de mieux que de coder une IA de test qui simulera le joueur. Pour implémenter un jeu de bataille navale avec une IA, On doit compléter la classe *CoreGame* avec des méthodes supplémentaires pour gérer la logique du jeu et l'IA. Voici une ébauche des étapes et méthodes nécessaires :

Initialisation du jeu :

- Placer les bateaux : Nous créerons une méthode pour placer les bateaux de manière aléatoire sur la grille pour le joueur et l'IA.

Déroulement d'un tour :

- Attaque du joueur : Ont utilisent la méthode *saisieJoueur* pour permettre au joueur de choisir une case à attaquer.
- Réponse de l'IA : Après chaque attaque du joueur, l'IA doit choisir une case à attaquer sur la grille du joueur.

Vérification des coups :

- Nous ajoutons des méthodes pour vérifier si une attaque touche un bateau, le manque, ou si un bateau est coulé.

Gestion de l'état du jeu :

- Vérifier la fin du jeu : Nous créons une méthode pour déterminer si tous les bateaux d'un joueur ont été coulés.

Logique de l'IA :

- Simple IA :Nous commençons donc avec une IA qui tire de manière aléatoire sur des cases non encore visées.

Boucle de jeu :

- Démarrez le jeu :Création d'une méthode pour démarrer le jeu et une boucle principale pour alterner entre les tours du joueur et de l'IA.

Globalement ça se résume à ajouter ces méthodes:

```
class CoreGame {  
    // ...  
  
    public:  
        // Ajout de nouvelles méthodes publiques pour le déroulement du jeu  
        void placerBateaux(); // Pour placer les bateaux de l'IA et du joueur  
        bool attaqueJoueur(int ligne, int colonne); // Pour que le joueur attaque l'IA  
        void attaqueIA(); // Pour que l'IA attaque le joueur  
        void jouer(); // Pour démarrer la boucle de jeu  
        bool estFinDuJeu() const; // Pour vérifier si la partie est terminée  
  
    // ...
```

Maintenant je peut jouer contre une IA et des caractère différent apparaissent sur la grille:
O = **vide** (*jamais touché*), **~** = **eau** (*tire manqué*), **X** = **touché**, **B** = **bateaux** (*ceux du joueur*)

```

Touché!
L'IA a manqué.
Bateaux coulés: 0 sur 0
Votre Grille:           Grille Adversaire:
0 ~ 0 0 ~ ~ ~ ~ ~ ~   ~ 0 0 0 0 ~ 0 0 0 0
~ 0 0 0 0 ~ 0 ~ 0 ~   0 0 0 0 ~ ~ ~ 0 0 0
0 0 0 ~ ~ 0 0 ~ 0 ~   0 0 0 0 0 ~ 0 0 0 0
~ ~ 0 ~ 0 ~ 0 0 0 ~   0 0 0 0 0 0 0 0 0 0
~ ~ ~ 0 0 ~ ~ ~ ~ 0   0 0 ~ 0 0 ~ ~ ~ ~ ~
~ ~ ~ 0 ~ 0 0 ~ 0 ~   0 ~ ~ 0 ~ 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0 ~
~ ~ X B X ~ 0 ~ 0 0   ~ ~ ~ 0 X 0 0 0 0 ~
~ 0 0 0 ~ ~ ~ 0 0 ~   ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
0 0 0 0 0 0 0 ~ 0 ~   ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
Entrez la ligne:
```

Néanmoins cette implémentation est très simplifiée. Dans un vrai jeu, on devrait gérer des détails tels que le nombre et la taille des bateaux, les règles pour placer les bateaux (pour éviter qu'ils se chevauchent), et une meilleure logique pour l'IA. Et vu qu'il s'agit d'un projet sérieux, nous irons jusqu'à ces détails.

```

void CoreGame::attaqueIA() {
    int ligne, colonne;
    do {
        ligne = std::rand() % nbLig;
        colonne = std::rand() % nbCol;
    } while (grille[ligne][colonne] == typeCase::touche || grille[ligne][colonne] == typeCase::eau); // \
    // L'IA attaque une case
    if (grille[ligne][colonne] == typeCase::bateau) {
        std::cout << espace5 << "\033[31mL'IA a touché\033[0m" << std::endl << std::endl;
        grille[ligne][colonne] = typeCase::touche;
    }
    else {
        std::cout << espace2 << "\033[34mL'IA a manqué\033[0m" << std::endl << std::endl;
        grille[ligne][colonne] = typeCase::eau;
    }
}
```

1. Choix d'une cible : L'IA choisit une case à attaquer sur la grille de jeu. Elle sélectionne aléatoirement des coordonnées (`ligne` et `colonne`) jusqu'à ce qu'elle trouve une case qui n'a pas encore été attaquée (c'est-à-dire une case qui n'est ni marquée comme "touchée" ni comme "eau").

2. Vérification et action :

- Si la case sélectionnée contient un bateau (`typeCase::bateau`), l'IA "touche" ce bateau. Cette action est marquée sur la grille (la case devient `typeCase::touche`) et un message s'affiche indiquant que l'IA a touché un bateau.

- Si la case ne contient pas de bateau, l'IA "manque" sa cible. Dans ce cas, la case est marquée comme "eau" (`typeCase::eau`) pour indiquer qu'aucun bateau n'est présent, et un message informe que l'IA a manqué sa cible.

En résumé, cette méthode permet à l'IA de choisir aléatoirement une case non attaquée à bombarder et de mettre à jour la grille de jeu en méthode du résultat de l'attaque.

9) Le placement des bateaux:

1. Introduction

La programmation du placement automatique des bateaux dans un jeu de bataille navale présente des défis significatifs, principalement en raison de la nécessité de gérer des contraintes variées et complexes.

2. Gestion des Bateaux pour Chaque Joueur

Chaque joueur doit posséder ses propres bateaux, un aspect que le programme doit reconnaître et gérer adéquatement. Cela implique la distinction entre les flottes de chaque joueur et la mise en place de mécanismes pour éviter toute confusion ou chevauchement.

3. Diversité des Positions et Tailles

Les bateaux doivent être placés de manière aléatoire sur la grille, tout en respectant des tailles variables (2, 3, 4, et 5 cases). Cette diversité de taille nécessite un algorithme capable d'évaluer l'espace disponible sur la grille pour placer correctement chaque bateau sans se chevaucher et donc vérifier si les cases adjacentes sont libres ou non.

4. Problèmes Rencontrés

- **Chevauchement des Bateaux :** Un problème majeur est le risque de chevauchement des bateaux, surtout dans des grilles de petite taille ou avec un grand nombre de bateaux.
- **Gestion de l'Espace Limité :** Il est parfois difficile de trouver un espace suffisant pour les plus grands bateaux, en particulier après que plusieurs petits bateaux aient été placés.
- **Optimisation de la Répartition :** Assurer une répartition équitable et stratégique des bateaux sur la grille peut être complexe, surtout si l'on cherche à éviter des configurations prévisibles.
- **Performance et Efficacité :** L'algorithme doit être suffisamment efficace pour exécuter le placement rapidement, même dans des scénarios de grille complexes.

5. Conclusion

Le placement automatique des bateaux dans un jeu de bataille navale est un aspect crucial qui requiert une attention particulière à la logique algorithmique et à la gestion des contraintes. Les défis liés à la diversité des tailles, la répartition équitable, et l'optimisation de l'espace rendent cette tâche particulièrement complexe et intéressante pour les développeurs de jeux.

Tout d'abord dans le constructeur on appellera la méthode **placerbateaux()** une fois avec **false**, et une autre fois avec **true**:

```
CoreGame::CoreGame() : nombreTotalBateaux(4), bateauxCoulés(0) {
    for (int i = 0; i < nbLig; ++i) {
        for (int j = 0; j < nbCol; ++j) {
            grille[i][j] = typeCase::vide;
            grilleAdversaire[i][j] = typeCase::vide; // Initialisation de la grille de l'adversaire
        }
    }
    placerBateaux(false); // Place un bateau pour le joueur
    placerBateaux(true); // Place un bateau pour l'IA
}
```

Mais pourquoi faire ? Tout simplement car la méthode **placerbateaux()** passe directement un test avec une expression conditionnelle. Si **pourAdversaire** est vrai (**true**), alors la valeur **grilleAdversaire** est utilisée ; sinon, la valeur **grille** est utilisée. **grille** qui est celle du joueur.

L'expression conditionnelle:

```
typeCase(*grilleCible)[nbCol] = pourAdversaire ? grilleAdversaire : grille;
```

Et enfin voici la méthode qui gère la logique du placement des bateaux selon leurs taille et leurs grille:

```
void CoreGame::placerBateaux(bool pourAdversaire) {
    typeCase(*grilleCible)[nbCol] = pourAdversaire ? grilleAdversaire : grille;
    const std::vector<int> taillesBateaux = { 3, 2, 4, 5 };

    for (int tailleBateau : taillesBateaux) {
        bool placementValide = false;
        while (!placementValide) {
            int direction = std::rand() % 2; // 0 pour horizontal, 1 pour vertical
            int ligne = std::rand() % nbLig;
            int colonne = std::rand() % nbCol;

            placementValide = true;
            for (int i = 0; i < tailleBateau; ++i) {
                int l = ligne + (direction == 0 ? 0 : i);
                int c = colonne + (direction == 1 ? 0 : i);
                if (l >= nbLig || c >= nbCol || grilleCible[l][c] != typeCase::vide || !caseAdjacenteLibre(l, c, grilleCible)) {
                    placementValide = false;
                    break;
                }
            }

            if (placementValide) {
                for (int i = 0; i < tailleBateau; ++i) {
                    int l = ligne + (direction == 0 ? 0 : i);
                    int c = colonne + (direction == 1 ? 0 : i);
                    grilleCible[l][c] = typeCase::bateau;
                }
            }
        }
    }
}
```

Décomposons le fonctionnement de la méthode **placerbateaux()** :

Le vecteur **taillesBateaux** ci-dessous représente les bateaux disponible par joueurs, chaque joueur doit avoir un bateau avec 2 cases, 3 cases, 4 cases et 5 cases selon les règles classique de la bataille navale disponible sur wikipedia : [cliquez ici](#)

```
const std::vector<int> taillesBateaux = { 3, 2, 4, 5 };
```

3. Boucle de Placement : Pour chaque bateau (en fonction de sa taille), la méthode tente de le placer sur la grille.

4. Choix de Position et Orientation : Elle choisit aléatoirement une ligne et une colonne pour le point de départ du bateau, ainsi qu'une orientation (horizontale ou verticale).

```
int direction = std::rand() % 2; // 0 pour horizontal, 1 pour vertical
int ligne = std::rand() % nbLig;
int colonne = std::rand() % nbCol;
```

5. Validation du Placement : Avant de placer un bateau, elle vérifie si toutes les cases nécessaires sont libres (`typeCase::vide`) et si elles n'ont pas de cases adjacentes occupées. Si le placement n'est pas valide, elle choisit une nouvelle position et réessaye.

```
placementValide = true;
for (int i = 0; i < tailleBateau; ++i) {
    int l = ligne + (direction == 0 ? 0 : i);
    int c = colonne + (direction == 1 ? 0 : i);
    if (l >= nbLig || c >= nbCol || grilleCible[l][c] != typeCase::vide || !caseAdjacenteLibre(l, c, grilleCible)) {
        placementValide = false;
        break;
    }
}
```

6. Placement du Bateau : Une fois un emplacement valide trouvé, le bateau est placé sur la grille, en marquant les cases correspondantes comme `typeCase::bateau`.

```
if (placementValide) {
    for (int i = 0; i < tailleBateau; ++i) {
        int l = ligne + (direction == 0 ? 0 : i);
        int c = colonne + (direction == 1 ? 0 : i);
        grilleCible[l][c] = typeCase::bateau;
    }
}
```

Cette méthode illustre la complexité de créer un algorithme efficace pour le placement automatique des bateaux, en tenant compte des contraintes de taille, d'orientation, et de la proximité des autres bateaux.

Vérifier les cases adjacentes du plateau avec la méthode `caseAdjacenteLibre()` :

```
bool CoreGame::caseAdjacenteLibre(int ligne, int colonne, typeCase(*grilleCible)[nbCol]) {
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
            int l = ligne + i;
            int c = colonne + j;
            if (l >= 0 && l < nbLig && c >= 0 && c < nbCol) {
                if (grilleCible[l][c] != typeCase::vide) {
                    return false; // Une case adjacente n'est pas vide
                }
            }
        }
    }
    return true; // Toutes les cases adjacentes sont libres
}
```

La méthode `CoreGame::caseAdjacenteLibre(int ligne, int colonne, typeCase(*grilleCible)[nbCol])` vérifie si toutes les cases adjacentes à une case donnée dans une grille de jeu sont libres (c'est-à-dire, dans l'état "vide"). Voici comment elle fonctionne :

Analyse de la Méthode

Parcours des Cases Adjacentes :

- La méthode utilise deux boucles imbriquées pour parcourir toutes les cases adjacentes à la case spécifiée par ligne et colonne.
- Les indices *i* et *j* sont utilisés pour déplacer le point de focalisation autour de la case cible, couvrant les cases dans un bloc 3x3 centré sur la case cible.

Calcul des Coordonnées Adjacentes :

- `int l = ligne + i;`
- `int c = colonne + j;`
- Ces lignes calculent les coordonnées des cases adjacentes. Pour une case donnée, cela inclut la case elle-même, ses voisins horizontaux, verticaux et diagonaux.
-

Vérification des Limites de la Grille :

- `if(l >= 0 && l < nbLig && c >= 0 && c < nbCol) {`
- Cette condition s'assure que les coordonnées calculées se trouvent bien à l'intérieur des limites de la grille, évitant ainsi les débordements de tableau.

Vérification de l'État de la Case :

- `if(grilleCible[l][c] != typeCase::vide) { return false; }`
- Cette ligne vérifie si la case actuellement examinée est dans un état différent de "vide" (défini par `typeCase::vide`). Si c'est le cas, cela signifie qu'au moins une des cases adjacentes n'est pas libre, et la méthode retourne `false`.

Conclusion :

- Si aucune des cases adjacentes n'est occupée (toutes sont dans l'état "vide"), la méthode retourne `true`, indiquant que toutes les cases adjacentes à la case spécifiée sont libres.

10) La boucle du jeu:

J'ai créé une nouveau main pour remplacer le main de test du début:

```
int main()
{
    std::srand(static_cast<unsigned int>(std::time(nullptr))); // Réinitialiser le générateur de nombres aléatoires
    SautLaLigne;
    CoreGame jeu;
    jeu.jouer();

    if (jeu.partiePerdu()) {
        std::cout << espace << "Vous avez perdu la partie." << std::endl;
    }
    else {
        std::cout << espace << "Félicitations ! Vous avez gagné !" << std::endl;
    }

    return 0;
}
```

Dans le main ci-dessus j'appel la méthode **jouer()** avec l'objet instancié de la classe **CoreGame**. Et c'est cette méthode qui va faire la boucle du jeu. Examinons cette méthode de plus près:

```
void CoreGame::jouer() {
    // Boucle principale du jeu, alternant entre le joueur et l'IA
    while (!estFinDuJeu()) {

        afficheGrille();
        std::pair<int, int> saisie = saisieJoueur();
        int ligne = saisie.first;
        int colonne = saisie.second;
        std::cout << espace3 << "L'adversaire joue..." ;
        Sleep(1000);
        system("cls");

        std::cout << std::endl;
        if (attaqueJoueur(ligne, colonne)) {
            std::cout << std::endl << std::endl << std::endl << std::endl;
            std::cout << espace4 << "\033[34mTouch\202!\033[0m" << std::endl << std::endl;
        }
        else {
            std::cout << std::endl << std::endl << std::endl << std::endl;
            std::cout << espace4 << "\033[31mManqu\202!\033[0m" << std::endl << std::endl;
        }
        attaqueIA();
    }
}
```

La première chose que l'on constate c'est que cette méthode crée une boucle de tour par tour entre le joueur 1 et le joueur 2 (L'IA) tant que le jeu n'est pas fini.

La condition est la suivante, le jeu tournera tant que la méthode **estFinDuJeu()** renvoie une valeur fausse (false).

Pour vérifier si le jeu a atteint sa fin, voici comme est composé la méthode **estFinDuJeu()**:

```
bool CoreGame::estFinDuJeu() const {
    bool joueurPerdu = true, adversairePerdu = true;

    for (int i = 0; i < nbLig; ++i) {
        for (int j = 0; j < nbCol; ++j) {
            if (grille[i][j] == typeCase::bateau) {
                joueurPerdu = false;
            }
            if (grilleAdversaire[i][j] == typeCase::bateau) {
                adversairePerdu = false;
            }
        }
    }

    if (joueurPerdu) {
        std::cout << "Vous avez perdu la partie." << std::endl;
    }
    else if (adversairePerdu) {
        std::cout << "Vous avez gagn\u2022 la partie !" << std::endl;
    }

    return joueurPerdu || adversairePerdu;
}
```

Le méthode **estFinDuJeu()** de type booléenne crée deux variable booléenne placé en true. Ensuite elle parcourt la grille en vérifiant l'état du jeu de la grille du joueur 1 et 2. Si une seule case "bateau" est trouvée sur la grille du joueur 1, alors la valeur de la variable bool du joueur passe en false et la méthode renvoie false. Alors le jeu continue, dans le cas inverse ou la méthode ne trouve pas de bateaux sur la grille du joueur 1. Alors l'analyse de la grille se termine est la variable du joueur 1 reste à true, il remplit donc la condition du "if" et un message apparaît pour signaler sa victoire et le la méthode retourne true, ce qui cause la fin de la boucle de jeu. Et c'est le même mécanisme pour le joueur 2.

Retournons à la méthode **jouer()**, une fois entré dans le **while**, c'est là que le déroulant du jeu tour par tour commence.

11) Actualiser la grille du jeu:

Passons à cette méthode:

```
afficheGrille();
```

En suivant le déroulement de la méthode **jouer()** une fois entré dans le main, le programme poursuit sa route dans la méthode **afficheGrille()**, cette méthode ne sert qu'à actualiser la grille de jeu des deux joueur :

```
void CoreGame::afficheGrille() const {
    std::cout << espace << "Votre Grille:" << "           Grille Adversaire:\n";
    // Afficher les repères de colonne
    std::cout << espace << " ";
    for (int i = 0; i < nbCol; ++i) {
        std::cout << "\033[90m" << i << "\033[0m" << " ";
    }
    std::cout << " ";
    for (int i = 0; i < nbCol; ++i) {
        std::cout << "\033[90m" << i << "\033[0m" << " ";
    }
    std::cout << std::endl;
    for (int i = 0; i < nbLig; ++i) {
        // Afficher le repère de ligne
        std::cout << espace << "\033[90m" << i << "\033[0m" << " ";
        // Afficher la grille du joueur
        for (int j = 0; j < nbCol; ++j) {
            afficherCaractereAvecCouleur(grille[i][j], false);
            std::cout << " ";
        }
        std::cout << "\t" << "\033[90m" << i << "\033[0m" << " "; // Repère de ligne pour la grille de l'adversaire
        // Afficher la grille de l'adversaire
        for (int j = 0; j < nbCol; ++j) {
            // Afficher seulement les tirs effectués sur la grille de l'adversaire
            if (grilleAdversaire[i][j] == typeCase::touche || grilleAdversaire[i][j] == typeCase::eau) {
                afficherCaractereAvecCouleur(grilleAdversaire[i][j], true);
            }
            else {
                // Afficher '0' pour les cases vides et celles contenant des bateaux non découverts
                afficherCaractereAvecCouleur(typeCase::vide, true);
            }
            std::cout << " ";
        }
        std::cout << std::endl;
    }
}
```

Globalement: La méthode appelée `afficheGrille()` s'occupe de l'affichage des grilles de jeu pour la bataille navale. Cette méthode affiche à la fois ma grille en tant que joueur et la grille de mon adversaire. Pour améliorer la lisibilité de l'affichage, elle utilise des repères de colonne et de ligne.

Je commence par afficher un en-tête indiquant clairement les deux grilles, la mienne et celle de mon adversaire. Ensuite, j'utilise des boucles pour afficher les numéros de colonne, en utilisant une couleur grisée pour les rendre plus visibles.

En parcourant les lignes de ma grille, j'affiche d'abord le repère de ligne correspondant, suivi des caractères qui représentent ma grille. J'utilise la fonction `afficherCaractereAvecCouleur()` pour afficher les caractères avec la couleur appropriée, ce qui rend la grille bien plus agréable à regarder. Ensuite, j'ajoute une tabulation pour séparer visuellement les deux grilles.

Pour la grille de mon adversaire, je n'affiche que les tirs que j'ai effectués, comme les cases touchées ou l'eau. Les cases vides et celles contenant des bateaux non découverts sont représentées par la lettre 'O', ce qui me permet de garder une trace de mes tirs précédents.

En résumé, ma méthode `afficheGrille()` offre un affichage clair et lisible des grilles de jeu pour la bataille navale, avec des repères de colonne et de ligne pour faciliter ma navigation et ma compréhension du statut des cases sur les deux grilles. Elle contribue ainsi à mon expérience de jeu en me permettant de suivre l'évolution de la partie.

12) Saisie du joueur:

Ensuite la boucle passe par cette ligne, mais pourquoi est-ce que je l'ai ajouté?:

```
std::pair<int, int> saisie = saisieJoueur();
```

Il est d'abord nécessaire de comprendre que dans la méthode `saisieJoueur()` , j'effectue la saisie des coordonnées de l'utilisateur, c'est-à-dire la ligne et la colonne où il souhaite effectuer un tir dans le jeu de la bataille navale.

```
std::pair<int, int> CoreGame::saisieJoueur() {
    // lire les entrées du joueur.
    int ligne, colonne;
    std::cout << std::endl << espace
        << "Entrez la ligne: ";
    std::cin >> ligne; std::cout << std::endl;
    std::cout << espace
        << "Entrez la colonne: ";
    std::cin >> colonne; std::cout << std::endl;
    return { ligne, colonne };
}
```

Voici une explication de ce que fait chaque partie de cette méthode :

1. Je déclare deux variables entières `ligne` et `colonne` pour stocker les valeurs que l'utilisateur va entrer.
2. J'affiche un message demandant à l'utilisateur d'entrer la ligne où il souhaite effectuer son tir en utilisant `std::cout`. La variable `espace` est un **#define** pour formater l'affichage.
3. J'utilise `std::cin` pour lire l'entrée de l'utilisateur dans la variable `ligne`.
4. J'affiche ensuite un message similaire pour demander à l'utilisateur d'entrer la colonne où il souhaite effectuer son tir.
5. J'utilise à nouveau `std::cin` pour lire l'entrée de l'utilisateur dans la variable `colonne`.
6. Finalement, je retourne une paire d'entiers en utilisant la syntaxe `{ ligne, colonne }` , ce qui signifie que je retourne les valeurs de `ligne` et `colonne` sous forme de paire. Cette paire représente les coordonnées (ligne, colonne) que l'utilisateur a saisies pour son tir.

En résumé, cette méthode `saisieJoueur()` est responsable de la saisie des coordonnées de tir de l'utilisateur (ligne et colonne) et renvoie ces coordonnées sous forme de paire d'entiers. Ces coordonnées seront utilisées pour déterminer où le joueur souhaite effectuer son tir dans le jeu de la bataille navale.

La boucle du jeu passe ensuite par ces deux lignes:

```
int ligne = saisie.first;
int colonne = saisie.second;
```

Dans le même contexte que précédemment, les lignes suivantes :

```
int ligne = saisie.first;
int colonne = saisie.second;
```

servent à extraire les valeurs de la paire d'entiers renvoyée par la méthode `saisieJoueur()` et à les stocker dans les variables `ligne` et `colonne`. Cette opération est importante car elle permet d'obtenir les coordonnées que l'utilisateur a saisies pour son tir.

En détail :

- `saisie` est la paire d'entiers qui contient les coordonnées (ligne, colonne) saisies par l'utilisateur dans la méthode `saisieJoueur()`.
- `saisie.first` correspond à la première composante de la paire, c'est-à-dire la ligne que l'utilisateur a saisie.
- `saisie.second` correspond à la deuxième composante de la paire, c'est-à-dire la colonne que l'utilisateur a saisie.

En assignant ces valeurs aux variables `ligne` et `colonne`, le programme obtient facilement les coordonnées que l'utilisateur a spécifiées pour son tir, ce qui lui permet ensuite de traiter ces données pour effectuer les opérations de jeu nécessaires, telles que la vérification de la validité des coordonnées et la mise à jour de la grille de jeu en fonction de la tentative de tir de l'utilisateur.

13) gérer les attaques:

Pour suivre l'implémentation de la logique du jeu de la bataille navale, j'ai du créer ces lignes de test dans ma boucle de jeu, voyons de plus près à quoi elles correspondent:

```
if (attaqueJoueur(ligne, colonne)) {
    std::cout << std::endl << std::endl << std::endl << std::endl;
    std::cout << espace4 << "\033[34mTouch\202!\033[0m" << std::endl << std::endl;
}
else {
    std::cout << std::endl << std::endl << std::endl << std::endl;
    std::cout << espace4 << "\033[31mManqu\202!\033[0m" << std::endl << std::endl;
}
```

Ces lignes de code sont responsables de l'affichage du résultat d'une attaque du joueur dans le jeu de la bataille navale. Elles se trouvent dans une structure conditionnelle qui évalue le résultat de la méthode `attaqueJoueur()` et affiche un message en conséquence.

Voici une explication de ce que font ces lignes :

1. La méthode `attaqueJoueur(int ligne, int colonne)` est appelée pour vérifier l'attaque du joueur à la position spécifiée par les coordonnées `ligne` et `colonne`. Cette méthode renvoie `true` si l'attaque touche un bateau et `false` si elle manque la cible.

2. La condition `if (attaqueJoueur(ligne, colonne))` vérifie le résultat de l'attaque du joueur. Si l'attaque est un succès (c'est-à-dire qu'elle touche un bateau), le code dans le bloc `if` est exécuté. Sinon, le code dans le bloc `else` est exécuté.

3. Si l'attaque est un succès, les lignes suivantes sont exécutées dans le bloc `if` :

- Quatre lignes vides sont ajoutées pour séparer visuellement le résultat précédent.
- Un message est affiché en utilisant `std::cout` pour informer le joueur qu'il a touché un bateau.

Le texte est affiché en bleu ("\\033[34m") pour indiquer une réussite.

4. Si l'attaque est un échec (manque la cible), les lignes suivantes sont exécutées dans le bloc `else` :

- Quatre lignes vides sont ajoutées pour séparer visuellement le résultat précédent.
- Un message est affiché en utilisant `std::cout` pour informer le joueur qu'il a manqué sa cible.

Le texte est affiché en rouge ("\\033[31m") pour indiquer un échec.

En résumé, ces lignes de code permettent de donner un retour visuel au joueur après qu'il ait effectué une attaque. Le résultat de l'attaque (touché ou manqué) est déterminé par la méthode `attaqueJoueur()`, et un message approprié est affiché en fonction de ce résultat pour informer le joueur de l'issue de son attaque.

Voici plus en détaille la méthode `attaqueJoueur()` :

```
bool CoreGame::attaqueJoueur(int ligne, int colonne) {
    // Si la case a déjà été attaquée, ne rien faire
    if (grilleAdversaire[ligne][colonne] == typeCase::touche || grilleAdversaire[ligne][colonne] == typeCase::eau) {
        return false;
    }

    // Vérification simple de la case attaquée
    if (grilleAdversaire[ligne][colonne] == typeCase::bateau) {
        grilleAdversaire[ligne][colonne] = typeCase::touche; // Marquer la case comme touchée sur la grille de l'adversaire
        verifierBateauCoule(ligne, colonne, true); // Vérifier si un bateau a été coulé sur la grille de l'adversaire
        return true;
    }
    else {
        grilleAdversaire[ligne][colonne] = typeCase::eau; // Marquer la case comme eau (manqué) sur la grille de l'adversaire
        return false;
    }
}
```

Et enfin nous y voila, la fin de la boucle tour par tour se conclue donc avec l'attaque du joueur adverse:

attaqueIA();

Observons cette méthode plus en détaille:

```
void CoreGame::attaqueIA() {
    int ligne, colonne;
    do {
        ligne = std::rand() % nbLig;
        colonne = std::rand() % nbCol;
    } while (grille[ligne][colonne] == typeCase::touche || grille[ligne][colonne] == typeCase::eau); // éviter d'attaquer une case déjà touchée
    if (grille[ligne][colonne] == typeCase::bateau) {
        std::cout << espace5 << "\033[31mL'IA a touché un bateau !" << std::endl << std::endl;
        grille[ligne][colonne] = typeCase::touche;
    }
    else {
        std::cout << espace2 << "\033[34mL'IA a manqué." << std::endl << std::endl;
        grille[ligne][colonne] = typeCase::eau;
    }
}
```

La méthode `attaqueIA()` est responsable de la gestion de l'attaque de l'IA (intelligence artificielle). Voici une explication de ce que fait cette méthode :

- 1. Un couple d'entiers `ligne` et `colonne`** est déclaré pour représenter les coordonnées de l'attaque de l'IA.
- 2. À l'intérieur d'une boucle `do-while`**, l'IA génère aléatoirement des valeurs pour `ligne` et `colonne` en utilisant `std::rand() % nbLig` et `std::rand() % nbCol`. Ces valeurs sont générées de manière aléatoire pour simuler l'attaque aléatoire de l'IA.

3. La boucle `do-while` s'assure que les coordonnées générées ne correspondent pas à des cases déjà attaquées. Elle vérifie si la case correspondante dans la grille du joueur (`grille[ligne][colonne]`) est soit `typeCase::touche` (déjà touchée) ou `typeCase::eau` (manquée). Si les coordonnées correspondent à une case déjà attaquée, la boucle régénère de nouvelles coordonnées aléatoires jusqu'à ce qu'elle obtienne une case non attaquée.

4. Une fois que des coordonnées valides sont générées, l'IA effectue son attaque en vérifiant le contenu de la case correspondante dans la grille du joueur ('grille[[ligne]][colonne]').

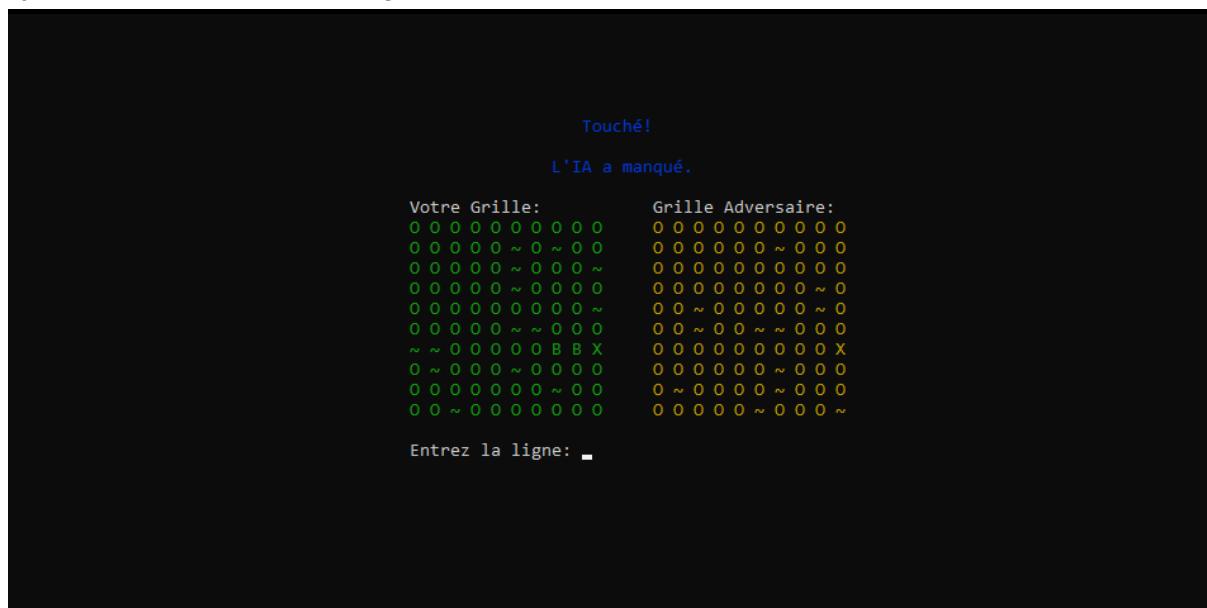
5. Si la case contient un bateau (typeCase::bateau), l'IA affiche un message indiquant qu'elle a touché un bateau en utilisant `std::cout`. Le texte est affiché en rouge ("\\033[31m") pour indiquer une réussite, et la case est marquée comme `typeCase::touche` dans la grille du joueur pour indiquer qu'elle a été touchée.

6. Si la case ne contient pas de bateau (c'est-à-dire `typeCase::eau`), l'IA affiche un message indiquant qu'elle a manqué sa cible. Le texte est affiché en bleu ("\033[34m") pour indiquer un échec, et la case est marquée comme `typeCase::eau` dans la grille du joueur pour indiquer qu'elle a été manquée.

En résumé, cette méthode `attaquerIA()` permet à l'IA de choisir aléatoirement une case non encore attaquée dans la grille du joueur, d'effectuer une attaque à cet endroit, et d'afficher un message approprié pour informer le joueur du résultat de l'attaque de l'IA.

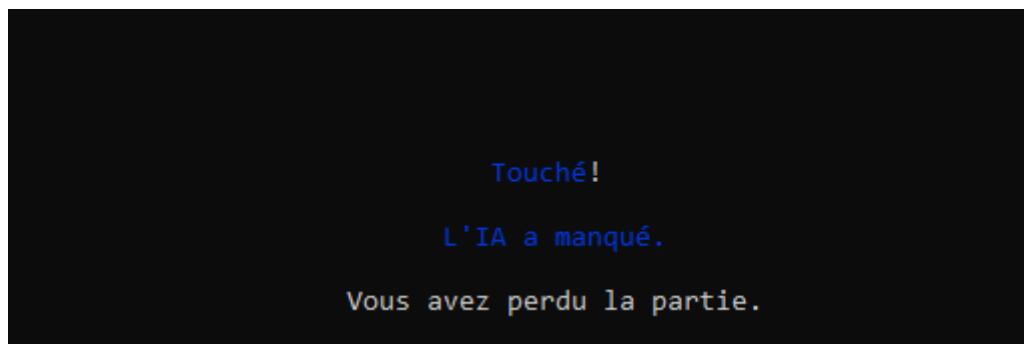
14) Historique des problèmes rencontré:

Ajout des colonnes de renseignement:

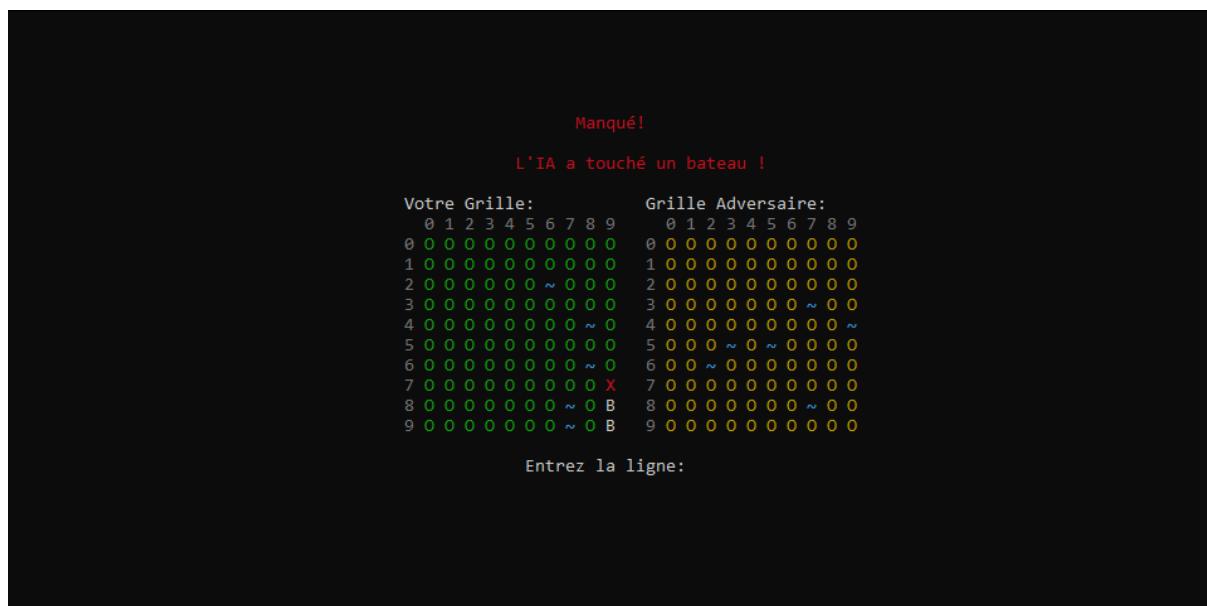


Je rencontre un problème, car quand je fais couler le seul bateau de 3 cases de mon adversaire, c'est moi qui perds. Pourquoi ? Tout simplement parce que son bateau était aussi le mien pour le programme. Alors, quand je gagnais, en fait, c'est lui qui gagnait. Car le programme vérifie d'abord la victoire de l'adversaire, qui renvoie forcément "true".

On voit bien que malgré le fait que l'IA rate son tir et que je réussisse le mien, je perds quand même.



Ajout des couleurs spécifiques à chaque caractère pour rendre le jeu plus lisible :



Ajout de la possibilité de générer **plusieurs** bateaux avec un **emplacement différent** de celui du joueur adverse!

```
Touché!  
L'IA a manqué.  
  
Votre Grille:          Grille Adversaire:  
 0 1 2 3 4 5 6 7 8 9  0 1 2 3 4 5 6 7 8 9  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 0 0 0 0 B B X B 0 0 1 0 0 B B B B X 0 0 0  
2 0 0 0 B 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0  
3 0 0 0 B 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 B  
4 0 0 0 B 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 B  
5 0 0 ~ 0 0 0 0 0 0 0 5 0 0 0 0 0 ~ B 0 0 B  
6 0 0 0 0 0 0 0 0 0 0 6 0 B B B 0 0 B 0 0 B  
7 0 0 0 0 0 0 0 0 0 B 7 0 0 0 0 0 0 0 0 0 0 0  
8 0 0 0 0 0 0 0 0 0 B 8 0 0 0 0 0 0 0 0 0 0 0  
9 B B B B B 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0 0  
  
Entrez la ligne:
```

Bateaux séparés d'une casse, il ne se chevauche plus :

```
Votre Grille:          Grille Adversaire:  
 0 1 2 3 4 5 6 7 8 9  0 1 2 3 4 5 6 7 8 9  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0  
2 0 0 0 B B B B B 0 0 2 0 0 B 0 0 B B 0 0 0  
3 0 0 0 0 0 0 0 0 0 0 3 0 0 B 0 0 0 0 0 0 0 0  
4 0 0 0 0 0 0 0 0 0 0 4 0 0 B 0 0 0 0 0 0 0 0  
5 0 0 B 0 0 B B 0 0 5 0 0 B 0 0 0 0 0 0 0 0  
6 0 0 B 0 0 0 0 0 0 0 6 0 0 B 0 0 0 0 0 0 0 0  
7 0 0 B 0 B 0 0 0 0 0 7 0 0 0 0 0 0 B 0 0 0  
8 0 0 B 0 B 0 0 0 0 0 8 0 0 0 0 0 0 B 0 0 0  
9 0 0 0 0 B 0 0 0 0 0 9 0 B B B B 0 B 0 0 0
```

Entrez la ligne: ▶

A présent les bateaux sont cachés sur la grille adverse:

```
Touché!  
L'IA a manqué.  
  
Votre Grille:          Grille Adversaire:  
 0 1 2 3 4 5 6 7 8 9    0 1 2 3 4 5 6 7 8 9  
0 0 0 0 0 0 0 0 0 0    0 0 0 0 0 0 0 0 0 0  
1 0 0 0 0 ~ 0 0 ~ 0 ~ 1 0 0 0 0 0 0 0 0 ~  
2 ~ 0 0 X B B X B 0 0 2 0 0 0 0 ~ ~ 0 ~ 0  
3 0 0 0 0 0 0 0 0 0 ~ 3 0 0 0 0 0 ~ 0 0 0  
4 0 0 0 0 0 0 0 0 0 0 4 0 0 X 0 0 ~ ~ 0 0 ~  
5 0 0 B 0 0 B B 0 0 5 0 0 0 ~ 0 0 0 0 0 0  
6 0 ~ X ~ 0 0 0 ~ 0 0 6 0 0 0 0 0 0 0 0 0 0  
7 ~ 0 B 0 B 0 0 0 0 0 7 0 0 0 0 ~ 0 0 0 ~ 0  
8 0 0 X 0 B 0 0 0 0 ~ 8 0 0 0 0 0 X X 0 0 0  
9 0 0 0 0 B 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0  
  
Entrez la ligne: -
```

15) La sérialisation:

La sérialisation est le processus de conversion d'une structure de données en une séquence de caractères ou de bits qui peut être facilement stockée, transmise ou utilisée ultérieurement pour reconstituer la structure de données d'origine. Notre but ici, c'est donc de parvenir à convertir l'état de notre grille en chaîne de caractères. La méthode `serialisationJoueur()` réalise la sérialisation de l'état actuel de la grille du joueur en une chaîne de caractères.

(j'ai ctrl + H, en remplaçant tout les occurrences "é" par "\202". Voilà pourquoi mes commentaires sont bizarre)

```
std::string CoreGame::serialisationJoueur() const {  
    // Convertit l'\202tat actuel de la grille en une chaîne de caractères.  
    std::string result;  
    for (int i = 0; i < nbLig; ++i) {  
        for (int j = 0; j < nbCol; ++j) {  
            result += std::to_string(static_cast<int>(grille[i][j])) + " ";  
        }  
        result += "\n";  
    }  
    return result;
```

Voici ce que fait le code en détail :

1. La méthode `serialisationJoueur()` retourne une chaîne de caractères (`std::string`) qui représente l'état actuel de la grille du joueur.

2. Elle initialise une chaîne de caractères vide appelée `result` qui servira à stocker la sérialisation de la grille.

3. Ensuite, elle utilise deux boucles `for` imbriquées pour parcourir toutes les cases de la grille. La première boucle (`for (int i = 0; i < nbLig; ++i)`) parcourt les lignes de la grille, et la deuxième boucle (`for (int j = 0; j < nbCol; ++j)`) parcourt les colonnes de chaque ligne.

4. À l'intérieur de la boucle interne, la méthode effectue les actions suivantes :

- **Elle convertit** le contenu de chaque case en un entier en utilisant `static_cast<int>(grille[i][j])`. Car `grille[i][j]` est de type énumération (`typeCase`), et il est converti en un entier pour être stocké dans la chaîne de caractères.

- **Elle ajoute cet entier converti** à la chaîne `result`, suivi d'un espace pour séparer les valeurs de chaque case.

5. Après avoir parcouru toutes les colonnes d'une ligne, la méthode ajoute un caractère de nouvelle ligne (`\n`) à la chaîne `result`. Cela permet de passer à la ligne suivante dans la chaîne de caractères pour représenter la grille de manière organisée.

6. Une fois que toutes les cases de la grille ont été parcourues, la méthode retourne la chaîne `result` qui contient la sérialisation complète de la grille du joueur.

En résumé, la méthode `serialisationJoueur()` parcourt la grille du joueur, convertit le contenu de chaque case en un entier et les assemble dans une chaîne de caractères avec des espaces et des retours à la ligne pour représenter la grille sous forme de texte. Cette chaîne de caractères peut être utilisée pour stocker ou transmettre l'état actuel de la grille du joueur.

Voici ci dessous un exemple des grilles sérialisé en ajoutant l'appel de la fonction des deux grille dans la boucle du jeu:

```
void CoreGame::jouer() {
    // Boucle principale du jeu, alternant entre le joueur et l'IA
    while (!estFinDuJeu()) {

        afficherBateauxCoules();
        afficheGrille();
        std::pair<int, int> saisie = saisieJoueur();
        int ligne = saisie.first;
        int colonne = saisie.second;
        std::cout << espace3 << "L'adversaire joue...";
        Sleep(1000);
        system("cls");
        std::cout << "MOI: " << std::endl << serialisationJoueur() << std::endl;
        std::cout << "ADVERSAIRE: " << std::endl << serialisationAdversaire();
        std::cout << std::endl;
        if (attaqueJoueur(ligne, colonne)) {
            std::cout << std::endl << std::endl << std::endl;
            std::cout << espace4 << "\033[34mTouch\202!\033[0m" << std::endl << std::endl;
        }
        else {
            std::cout << std::endl << std::endl << std::endl;
            std::cout << espace4 << "\033[31mManqu\202!\033[0m" << std::endl << std::endl;
        }
        attaqueIA();
    }
}
```

```
MOI:
0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 1 1 0 0
0 1 1 1 0 0 0 0 3 0
0 0 0 0 0 0 3 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 3 0 0 0 0 0
3 0 0 0 0 3 1 1 1 1
0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 2 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

```
ADVERSAIRE:
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 3 0 0 0 0
0 0 0 1 2 2 2 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 1 0 0
0 0 2 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 3 1
0 0 1 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 1
```

Manqué!

L'IA a manqué.

Votre Grille:	Grille Adversaire:
0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9
0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0
1 ~ 0 0 0 0 0 B B 0 0	1 0 0 0 0 0 ~ 0 0 0 0
2 0 B B B 0 0 0 0 ~ 0	2 0 0 0 0 X X X 0 0 0
3 0 0 0 0 0 0 ~ 0 0 0	3 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0	4 0 0 0 0 0 0 ~ 0 0 0
5 0 0 0 ~ 0 0 0 0 0 0	5 0 0 0 0 0 0 0 0 0 0
6 ~ 0 0 0 0 ~ B B B B	6 0 0 X 0 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 0 0 0	7 0 0 X 0 0 0 0 0 0 ~ 0
8 0 B B B B X 0 0 0 0	8 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 ~ 0	9 0 0 0 0 0 0 0 0 0 0 0

La sérialisation est afficher en suivant la logique suivante:

- 0 = case vide
- 1 = case bateau
- 2 = case touche
- 3 = case eau
- 4 = case erreur

c'est donc les cases de cette enum class qui comme un tableau commence à 0:

```
enum class typeCase { vide, bateau, touche, eau, erreur = 99};
```

enum associé au numéro de sérialisation:

```
#define espaces5 "\t\t\t\t\t\t"
#define SautLaLigne std::cout<<std::endl<<std::endl<<std::endl<<std::endl

class CoreGame {
public:
    enum class typeCase { vide, bateau, touche, eau, erreur = 99 };
    static const int nbLig = 10;
    static const int nbCol = 10;

private:
    typeCase grille[nbLig][nbCol];
    typeCase grilleAdversaire[nbLig][nbCol];
    int nombreTotalBateaux; // obsolete
    int bateauxCoulés; //obsolete

public:
    // Constructeur qui initialise la grille, par exemple avec des cases
    CoreGame();

    // Retourne la case de la grille à la position spécifiée.
    typeCase getCase(int ligne, int colonne) const;

    // Définit la case de la grille à la position spécifiée avec le type
    void setCase(int ligne, int colonne, typeCase type);

    // Affiche la grille dans la console ou l'interface utilisateur.
    void afficheGrille() const;
}

MOI:
0 0 0 0 0 0 0 3 0 0
0 0 0 0 3 0 0 3 0 0
0 3 1 3 0 0 0 0 0 3
0 0 1 0 0 0 0 0 0 3
0 0 1 0 0 0 1 1 0 0
3 0 2 0 0 0 3 0 3 0
0 3 0 0 0 0 0 0 0 0
1 0 3 0 0 0 0 0 0 0
2 0 0 0 0 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0

ADVERSAIRE:
0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 2
0 0 0 2 0 0 0 0 0 2
0 0 0 2 3 0 2 2 0 2
0 0 0 2 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

grille associé au trame de sérialisation:

```
Touché!
L'IA a manqué.

Votre Grille:          Grille Adversaire:
 0 1 2 3 4 5 6 7 8 9   0 1 2 3 4 5 6 7 8 9
 0 0 0 0 0 0 0 ~ 0 0   0 0 0 0 0 0 0 X 0 0
 1 0 0 0 0 ~ 0 0 ~ 0 0 1 0 0 0 0 0 0 X 0 0
 2 0 ~ B ~ 0 0 0 0 0 ~ 2 0 0 0 0 0 0 X 0 0
 3 0 0 B 0 0 0 0 0 ~ 0 3 0 0 0 0 0 0 X 0 X
 4 0 0 B 0 0 0 B B 0 0 4 0 0 X 0 0 0 0 0 X
 5 ~ 0 B 0 0 0 ~ 0 ~ 0 5 0 0 X ~ 0 X X 0 X
 6 0 ~ 0 0 0 0 0 0 0 0 6 0 0 X 0 0 0 0 0 0 0
 7 B 0 ~ 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0
 8 X 0 0 0 0 B B B B B 8 0 0 0 0 0 0 0 0 0 0
 9 B 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0

Entrez la ligne: 6
Entrez la colonne: 9
L'adversaire joue...

MOI:
0 0 0 0 0 0 0 3 0 0
0 0 0 0 3 0 0 3 0 0
0 3 1 3 0 0 0 0 0 3
0 0 1 0 0 0 0 0 0 3
0 0 1 0 0 0 1 1 0 0
3 0 2 0 0 0 3 0 3 0
0 3 0 0 0 0 0 0 0 0
1 0 3 0 0 0 0 0 0 0
2 0 0 0 0 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0

ADVERSAIRE:
0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0 2
0 0 0 2 0 0 0 0 0 2
0 0 0 2 3 0 2 2 0 2
0 0 0 2 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

NOTE: (Sur le jeu, dans la grille de l'adversaire en ligne 6, colonne 9. On voit une case vide (O) alors que sur la trame il y a un touché (X) = 2, c'est tout simplement car j'ai pris le screenshot du jeu avant la trame.)

Voici désormais la méthode de désérialisation :

```
bool CoreGame::deserialisationAdversaire(const std::string& trame) {
    std::istringstream iss(trame);
    std::string ligneTrame;
    int numLigne = 0;

    while (std::getline(iss, ligneTrame) && numLigne < nbLig) {
        std::istringstream issLigne(ligneTrame);
        int valeur;
        int numColonne = 0;

        while (issLigne >> valeur && numColonne < nbCol) {
            grilleAdversaire[numLigne][numColonne] = static_cast<typeCase>(valeur);
            numColonne++;
        }

        numLigne++;
    }

    return (numLigne == nbLig);
}
```

La méthode `CoreGame::deserialisationAdversaire(const std::string& trame)` dans le contexte du jeu de bataille navale est utilisée pour convertir l'état sérialisé de la grille de l'adversaire (reçu sous forme d'une chaîne de caractères) en une structure de données interne. Voici comment elle fonctionne :

Analyse de la Méthode

Initialisation :

- `std::istringstream iss(trame);` : Crée un flux d'entrée basé sur la chaîne de caractères reçue (trame), qui contient l'état sérialisé de la grille de l'adversaire.
- `std::string ligneTrame;` : Déclare une variable pour stocker chaque ligne de la trame.
- `int numLigne = 0;` : Initialise un compteur pour les lignes de la grille.

Traitements des Lignes :

- La boucle `while (std::getline(iss, ligneTrame) && numLigne < nbLig)` lit chaque ligne de la trame sérialisée jusqu'à ce que toutes les lignes (nbLig) soient traitées.
- À l'intérieur de cette boucle, chaque `ligneTrame` est traitée individuellement.

Traitements des Colonnes :

- `std::istringstream issLigne(ligneTrame);` : Pour chaque ligne, un nouveau flux d'entrée est créé pour extraire les valeurs individuelles représentant l'état des cases de la grille.
- `int valeur;` : Variable pour stocker chaque valeur extraite de la ligne.
- `int numColonne = 0;` : Compteur pour les colonnes de la grille.

Remplissage de la Grille :

- La boucle interne `while (issLigne >> valeur && numColonne < nbCol)` extrait chaque valeur de la ligne et la place dans la grille.
- `grilleAdversaire[numLigne][numColonne] = static_cast<typeCase>(valeur);` : Chaque valeur est convertie en un type approprié (`typeCase`, est une énumération représenter l'état de chaque case) et stockée dans la grille de l'adversaire à l'emplacement correspondant.
- Les compteurs `numColonne` et `numLigne` sont incrémentés après chaque valeur et chaque ligne traitée, respectivement.

Vérification de la Complétude :

- `return (numLigne == nbLig);` : La méthode renvoie true si le nombre de lignes traitées correspond au nombre total de lignes attendues dans la grille (`nbLig`), indiquant que la déserialisation s'est déroulée avec succès.

Conclusion

Cette méthode est cruciale pour la mise à jour de l'état de la grille de l'adversaire dans le jeu. Elle assure que les données reçues sur le réseau sont correctement converties en une structure de grille interne, permettant au jeu de refléter fidèlement l'état courant du jeu de l'adversaire. La déserialisation est une étape clé dans les jeux en réseau, permettant de synchroniser l'état du jeu entre différents joueurs.

Réalisation de test unitaire Catch2 :

Créer un test unitaire en **Catch2** pour la classe **CoreGame**, nous devons identifier les fonctions et comportements clés à tester. Parmi les méthodes de **CoreGame**,

plusieurs peuvent être testées :

Constructeur CoreGame : Vérifier que la grille et la grille de l'adversaire sont initialisées correctement.

afficheGrille : S'assurer que les grilles sont affichées correctement. Ceci peut être difficile à tester directement, mais on peut tester les composants utilisés par cette fonction.

placerBateaux : Tester si les bateaux sont placés correctement sur la grille.

attaqueJoueur et attaqueIA : Vérifier si les attaques sont gérées correctement.

verifierBateauCoule : Assurer que cette méthode détecte correctement lorsqu'un bateau est coulé.

partiePerdu et estFinDuJeu : S'assurer que ces fonctions retournent les bonnes valeurs selon l'état du jeu.

J'ai donc créé un test unitaire qui utilise deux sections pour tester des aspects différents du placement des bateaux :

La grille ne doit pas être entièrement vide après placement : Vérifie qu'après l'exécution du constructeur, la grille n'est pas entièrement vide.

Les bateaux ne doivent pas se chevaucher : Assure que les bateaux placés ne se chevauchent pas.

Voici mon test unitaire pour la méthode *placerBateaux*:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "CoreGame.h"

TEST_CASE("Les bateaux sont correctement placés sur la grille",
"[CoreGame]") {
    CoreGame game;

    SECTION("La grille ne doit pas être entièrement vide après placement") {
        bool empty = true;
        for (int i = 0; i < nbLig; ++i) {
            for (int j = 0; j < nbCol; ++j) {
                if (game.grille[i][j] != typeCase::vide) {
                    empty = false;
                    break;
                }
            }
            if (!empty) break;
        }
        REQUIRE_FALSE(empty);
    }

    SECTION("Les bateaux ne doivent pas se chevaucher") {
        bool overlap = false;
        for (int i = 0; i < nbLig; ++i) {
            for (int j = 0; j < nbCol; ++j) {
                if (game.grille[i][j] == typeCase::bateau) {

                    for (int di = -1; di <= 1; ++di) {
                        for (int dj = -1; dj <= 1; ++dj) {
                            if (di == 0 && dj == 0) continue;
                            int ni = i + di, nj = j + dj;
                            if (ni >= 0 && ni < nbLig && nj >= 0 &&
                                nj < nbCol) {
                                if (game.grille[ni][nj] == typeCase::bateau)
                                {
                                    overlap = true;
                                    break;
                                }
                            }
                        }
                    }
                    if (overlap) break;
                }
            }
            if (overlap) break;
        }
        if (overlap) break;
    }
    REQUIRE_FALSE(overlap);
}
}
```

16) Mise en place de la fonctionnalité multi Joueurs:

Pour que le jeu soit en mesure de fonctionner en mode multijoueur sur un réseau. Voici un résumé de ce que nous avons mis en place :

1. Communication Réseau :

- Utilisation de `TCPClient` pour envoyer et recevoir des données entre les joueurs.
- Échange des états de jeu et des attaques par le réseau.

2. Gestion des Attaques :

- Méthodes pour envoyer une attaque (`envoyerAttaque`) et pour recevoir et traiter une attaque de l'adversaire (`recevoirAttaque` et `traiterAttaqueAdversaire`).

3. Sérialisation et Désérialisation :

- Utilisation des méthodes de sérialisation et de désérialisation pour envoyer l'état des grilles entre les joueurs.

4. Boucle de Jeu :

- Modification de la boucle de jeu principale pour alterner entre les actions du joueur local et l'attente des actions du joueur distant.

5. Vérification des Conditions de Victoire :

- Vérification continue pour déterminer si un joueur a gagné ou perdu.

L'intégration :

Afin de tester le programme pour savoir si il sera compatible avec l'intégration, j'ai codé une classe TCP simple qui fonctionné avec le serveur de Traian

```
#pragma once
#ifndef TCPCLIENT_H
#define TCPCLIENT_H
#define _WINSOCK_DEPRECATED_NO_WARNINGS

#include <iostream>
#include <winsock2.h>
#include <string>
#include "BsBDD.h"

class CoreGame;

class TCPClient {
private:
    WSADATA WSADATA;
    SOCKET serverSocket;
    SOCKADDR_IN serverAddr;
    bool isConnected;
    CoreGame* __Coregame;
public:
    TCPClient();
    ~TCPClient();
    bool connectToServer(const std::string& address, int port);
    bool sendMessage(const std::string& message);
    std::string receiveMessage();
    void closeConnection();
};

#endif // TCPCLIENT_H
```

Et bien évidemment le cpp qui avec :

```
TCPCClient::TCPClient() : isConnected(false) {
    WSAStartup(MAKEWORD(2, 0), &WSAData);
}

TCPClient::~TCPClient() {
    if (isConnected) {
        closesocket(serverSocket);
    }
    WSACleanup();
}

bool TCPClient::connectToServer(const std::string& address, int port) {
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    serverAddr.sin_addr.s_addr = inet_addr(address.c_str());
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);

    if (connect(serverSocket, (SOCKADDR*)&serverAddr, sizeof(serverAddr)) != 0) {
        std::cerr << "Connexion au serveur échouée." << std::endl;
        return false;
    }

    std::cout << "Connecté au serveur!" << std::endl;
    isConnected = true;
    return true;
}

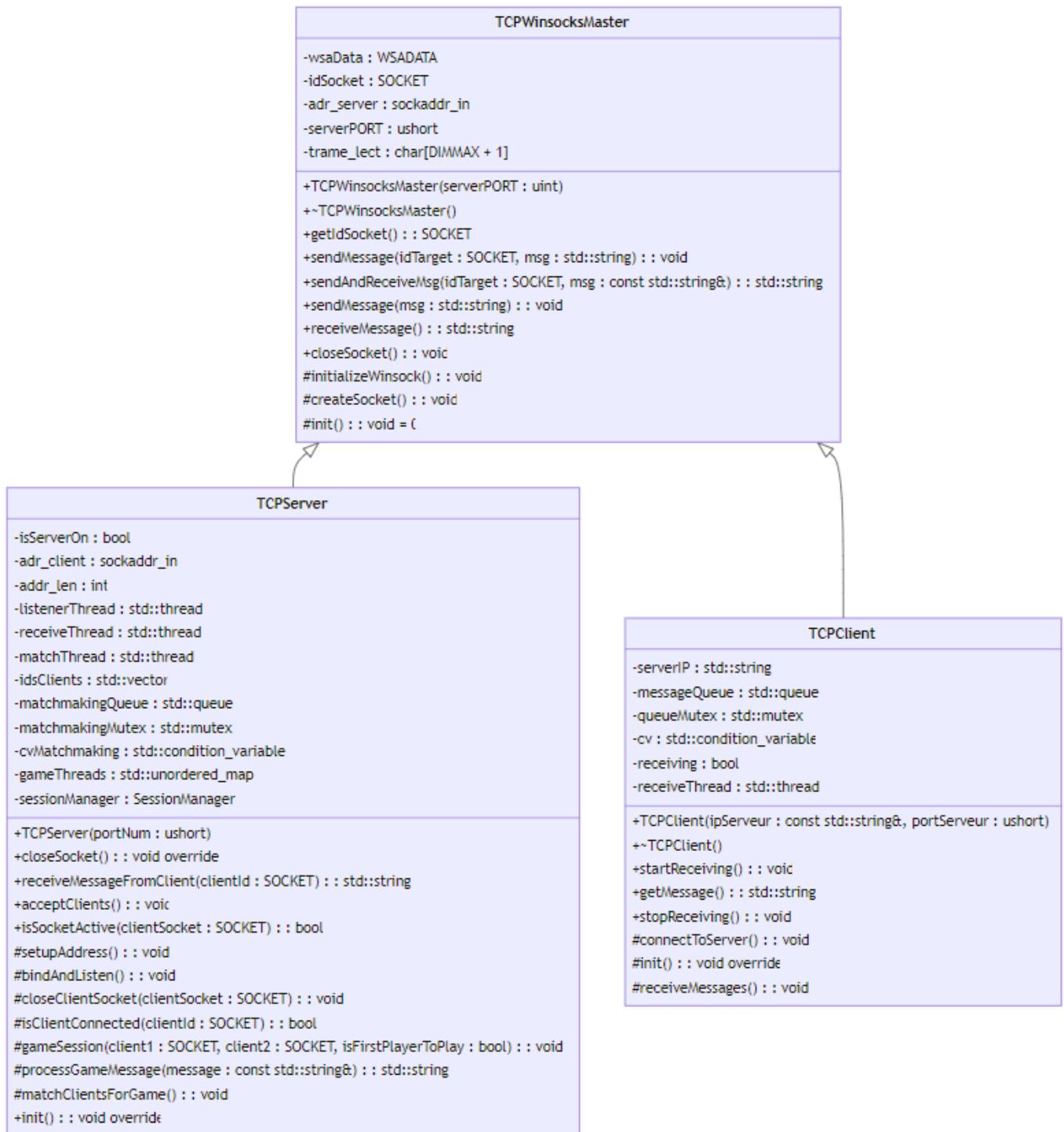
bool TCPClient::sendMessage(const std::string& message) {
    if (send(serverSocket, message.c_str(), message.size(), 0) < 0) {
        std::cerr << "Échec de l'envoi du message." << std::endl;
        return false;
    }
    return true;
}

std::string TCPClient::receiveMessage() {
    char buffer[1024] = { 0 };
    int bytesReceived = recv(serverSocket, buffer, sizeof(buffer), 0);
    if (bytesReceived < 0) {
        std::cerr << "Erreur de réception du message." << std::endl;
        return "";
    }
    return std::string(buffer, bytesReceived);
}

void TCPClient::closeConnection() {
    if (isConnected) {
        closesocket(serverSocket);
        isConnected = false;
        std::cout << "Connexion fermée." << std::endl;
    }
}
```

Ma classe TCP de test a très rapidement été remplacée avec le code final de Traïan. Car c'était sa partie, ma classe TCP n'était que temporaire.

Voici le diagramme UML des classes TCP :



(oups, petit erreur du logiciel qui a généré “.” au lieu de “:”, la raison est simple, le logiciel met automatiquement les ‘?’, et j’ai marqué *Fun(): void* au lieu de *Fun() void* qui m’aurait automatiquement généré les deux points, je m’en rend compte maintenant et j’ai plus la sauvegarde, de plus dans un diagramme UML on ne met pas de *std::* Et le programme a aussi enlevé le type de collection du vecteur, décidément...)

Pour faire fonctionner le jeu avec un serveur TCP il est d'abord nécessaire de créer des clients. Voici un exemple de l'instanciation d'un client :

```
int main()
{
    std::srand(static_cast<unsigned int>(std::time(nullptr)));
    TCPClient client("10.187.52.32", 12345);
    //std::cout << client.receiveMessage();
    CoreGame jeu(&client);
    jeu.jouer(); //boucle du jeu

    return 0;
}
```

Puis on passe l'adresse IP et le port dans le constructeur de mon instance (client) :

Déclaration:

```
CoreGame(TCPClient* tcpClient);
```

Définition:

```
CoreGame::CoreGame(TCPClient* tcpClient) : client(tcpClient) {}
```

Dans TCPClient :

```
TCPClient::TCPClient(const std::string& serverIP, ushort serverPORT)
    : TCPWinsocksMaster(serverPORT), serverIP(serverIP){
    init();
}
```

Les plus gros changement se feront dans la boucle du jeu, entre les échange des joueurs:

```
// Le joueur effectue une attaque
std::pair<int, int> saisie = saisieJoueur();
int ligne = saisie.first;
int colonne = saisie.second;
attaqueJoueur(ligne, colonne);
envoyerAttaque(saisie.first, saisie.second);

// Envoie l'attaque et l'état actuel de la grille
```

Allons voir de plus près la méthode *attaqueJoueur* et la méthode *envoyerAttaque*. La méthode *attaqueJoueur* n'a rien de nouveau. Si les coordonnées choisies tombent sur une case de type eau ou bien de type vide, alors on ne fait rien. Si c'est un bateau, on met à jour la case de l'adversaire en "touché"

```
bool CoreGame::attaqueJoueur(int ligne, int colonne) {
    // Si la case a déjà été attaquée, ne rien faire
    if (grilleAdversaire[ligne][colonne] == typeCase::touche || grilleAdversaire[ligne][colonne] == typeCase::eau) {
        return false;
    }

    // Vérification simple de la case attaquée
    if (grilleAdversaire[ligne][colonne] == typeCase::bateau) {
        grilleAdversaire[ligne][colonne] = typeCase::touche; // Marquer la case comme touchée sur la grille de l'adversaire
        verifierBateauCoule(ligne, colonne, true); // Vérifier si un bateau a été coulé sur la grille de l'adversaire
        return true;
    }
    else {
        grilleAdversaire[ligne][colonne] = typeCase::eau; // Marquer la case comme eau (manqué) sur la grille de l'adversaire
        return false;
    }
}
```

Par contre, la méthode *envoyerAttaque* est un peu différente :

```
void CoreGame::envoyerAttaque(int ligne, int colonne) {
    try {
        std::string etatGrille = serialisationAdversaire();
        client->sendMessage(etatGrille);
    }
    catch (const std::exception& e) {
        std::cerr << "Erreur lors de l'envoi de l'attaque : " << e.what() << std::endl;
        // Gestion de la reconnexion ou notification à l'utilisateur ici
    }
}
```

La méthode `CoreGame::envoyerAttaque(int ligne, int colonne)` et la méthode `TCPWinsocksMaster::sendMessage(std::string msg)` jouent un rôle crucial dans la communication entre les joueurs et la gestion des états du jeu. Voici une explication détaillée de chaque méthode :

```
void TCPWinsocksMaster::sendMessage(SOCKET idTarget, std::string msg)
{
    if (send(idTarget, msg.c_str(), msg.size() + 1, 0) == SOCKET_ERROR) {
        throw std::runtime_error("Send Failed SocketId : " + std::to_string(idTarget) + " msg : " + msg);
    }
}
```

Méthode CoreGame::envoyerAttaque(int ligne, int colonne)

Cette méthode est responsable de l'envoi des détails d'une attaque dans le jeu de bataille navale. Voici ce que chaque partie fait :

- **Tentative d'envoi d'une attaque** : La méthode commence par un bloc try, qui est utilisé pour gérer les exceptions qui peuvent survenir pendant l'exécution du code à l'intérieur du bloc.
- **Sérialisation de l'état de l'adversaire** : `std::string etatGrille = serialisationAdversaire();` Cette ligne appelle la méthode `serialisationAdversaire()`, que nous avons étudiée plus tôt, elle est responsable de la préparation de l'état actuel de la grille de l'adversaire sous forme de chaîne de caractères pour l'envoi.
- **Envoi de l'état de la grille** : `client->sendMessage(etatGrille);` Cette ligne appelle la méthode `sendMessage` sur l'objet `client`, en lui passant l'état de la grille sérialisé. Car `client` est une instance de `TCPWinsocksMaster` pour utiliser pour la communication réseau.
- **Gestion des exceptions** : Le bloc catch est utilisé pour attraper toute exception de type `std::exception` qui pourrait être levée pendant l'exécution du bloc try. Si une exception est attrapée, un message d'erreur est affiché, indiquant qu'il y a eu un problème lors de l'envoi de l'attaque.

Méthode TCPWinsocksMaster::sendMessage(std::string msg)

Cette méthode est responsable de l'envoi d'un message (dans ce cas, l'état de la grille) sur un réseau en utilisant Winsocks, une bibliothèque de sockets pour Windows.

- **Envoi du message** : La méthode utilise la fonction `send` de Winsocks pour envoyer le message. `idSocket` est le descripteur de socket utilisé pour la communication, `msg.c_str()` convertit le message `std::string` en un tableau de caractères C, et `msg.size() + 1` spécifie la longueur du message à envoyer, y compris le caractère de terminaison.
- **Gestion des erreurs** : Si `send` retourne `SOCKET_ERROR`, une exception `std::runtime_error` est levée avec le message "Send Failed", indiquant que l'envoi du message a échoué.

Conclusion

En résumé, `CoreGame::envoyerAttaque` prépare et envoie l'état de la grille de l'adversaire via le réseau, en utilisant `TCPWinsocksMaster::sendMessage`. Cette communication est essentielle pour le jeu de bataille navale, permettant aux joueurs de suivre les attaques et les mouvements de l'adversaire. La gestion des erreurs dans ces méthodes assure une meilleure robustesse du jeu, en traitant les problèmes de connexion ou d'autres erreurs potentielles.

Ensuite dans la suite du code j'ai du ajouter une méthode de réception de la trame adverse :

(appelé dans la boucle du jeu)

```
// Attendre et traiter l'attaque de l'adversaire
recevoirAttaque(); // Reçoit l'attaque de l'adversaire et met à jour les grilles
```

Voici la méthode :

```
void CoreGame::recevoirAttaque() {
    std::string message = client->receiveMessage();
    std::istringstream iss(message);
    int ligne, colonne;
    iss >> ligne >> colonne;

    // Traiter l'attaque
    traiterAttaqueAdversaire(ligne, colonne);

    // Recevoir et déserialiser l'état de la grille de l'adversaire
    std::string trameGrilleAdversaire = client->receiveMessage();
    deserialisationAdversaire(trameGrilleAdversaire);
}
```

détail de la méthode *receiveMessage* :

```
std::string TCPWinsocksMaster::receiveMessage()
{
    uint trameLenght = recv(idSocket, trame_lect, DIMMAX, 0);

    trame_lect[trameLenght] = '\0';
    return trame_lect;
}
```

La réception d'une attaque dans un jeu de bataille navale implique deux méthodes principales : *CoreGame::recevoirAttaque()* et *TCPWinsocksMaster::receiveMessage()*. Explorons le fonctionnement de ces méthodes :

Méthode CoreGame::recevoirAttaque()

Cette méthode gère la réception d'une attaque de l'adversaire. Voici les étapes clés :

- **Réception du message** : `std::string message = client->receiveMessage();`; Cette ligne appelle la méthode `receiveMessage` sur l'objet `client`. `Client` est une instance de `TCPWinsocksMaster` pour la communication réseau.
- **Extraction des coordonnées de l'attaque** : Le message reçu est ensuite traité pour en extraire les coordonnées de l'attaque. Un `std::istringstream` est utilisé pour séparer les éléments du message. Les coordonnées ligne et colonne sont extraites en utilisant l'opérateur `>>`.
- **Traitement de l'attaque** : `traiterAttaqueAdversaire(ligne, colonne);`; Cette ligne appelle une méthode pour traiter l'attaque de l'adversaire en utilisant les coordonnées extraites.
- **Réception de l'état de la grille de l'adversaire** : La méthode reçoit un autre message qui représente l'état de la grille de l'adversaire après l'attaque. Ce message est également reçu par l'appel `client->receiveMessage()`.
- **Déserialisation de l'état de la grille** : `deserialisationAdversaire(trameGrilleAdversaire);`; Cette ligne appelle une méthode pour convertir l'état de la grille reçu sous forme de chaîne de caractères en une structure de données utilisable par le jeu, donc en un tableau bidimensionnel.

Méthode TCPWinsocksMaster::receiveMessage()

Cette méthode est responsable de la réception d'un message sur un réseau via Winsocks.

- **Réception du message** : La fonction `recv` de Winsocks est utilisée pour recevoir des données sur le socket `idSocket`. `trame_lect` est le tableau de caractères où les données reçues sont stockées, et `DIMMAX` spécifie la taille maximale du message à recevoir.
- **Terminaison de la chaîne** : Après la réception du message, un caractère de terminaison `\0` est ajouté à la fin du tableau `trame_lect` pour s'assurer qu'il est correctement formaté comme une chaîne de caractères en C.
- **Retour du message** : La méthode renvoie le message reçu sous forme de chaîne de caractères `std::string`.

Conclusion

En résumé, `CoreGame::recevoirAttaque` gère la réception des coordonnées d'une attaque et de l'état de la grille de l'adversaire, tandis que `TCPWinsocksMaster::receiveMessage` s'occupe de la réception bas-niveau des messages sur le réseau. Ensemble, ces méthodes permettent une interaction en temps réel entre les joueurs dans le jeu de bataille navale, assurant que les attaques et les réponses sont correctement communiquées et traitées.

17) La base de données:

Dans le contexte de notre jeu de bataille navale, on a eu l'idée de créer une base de données. j'ai donc créer une classe pour la base de données :

```
class BsBDD {  
  
private:  
    sql::Driver* driver;  
    sql::Connection* con;  
    sql::PreparedStatement* pstmt;  
    sql::ResultSet* res;  
    std::string userId;  
  
public:  
    BsBDD();  
    ~BsBDD();  
  
    void Connexion();  
    void setPseudo(std::string name);  
    void connectToDB(const std::string& dbURI, const std::string& userName, const std::string& password);  
    bool login(const std::string& idPlayer, const std::string& password);  
    bool registerUser(const std::string& idUser, const std::string& password);  
    void loadPlayerData();  
    void savePlayerData();  
    void BonusWin();  
    void incrementNbGames();  
    void incrementNbLostGames();  
    void incrementNbWonGames();  
    void displayPlayerInfo();  
};
```

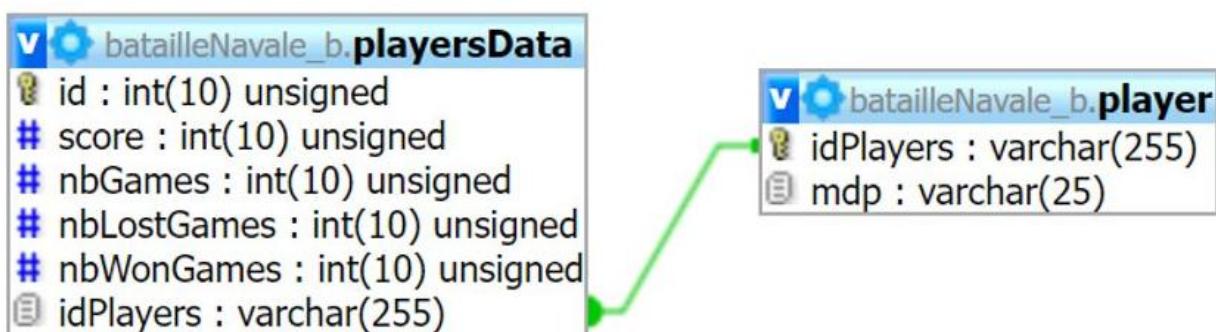
Nous remercions aussi **Mr Dauriac** de nous avoir offert l'opportunité d'avoir notre propre base de données sur le serveur du lycée.

Et nous tenons aussi à remercier **Mr Dartois** de nous avoir prêté un serveur externe au lycée.

La base de données a été créer via phpMyAdmin



voici le diagramme des tables :

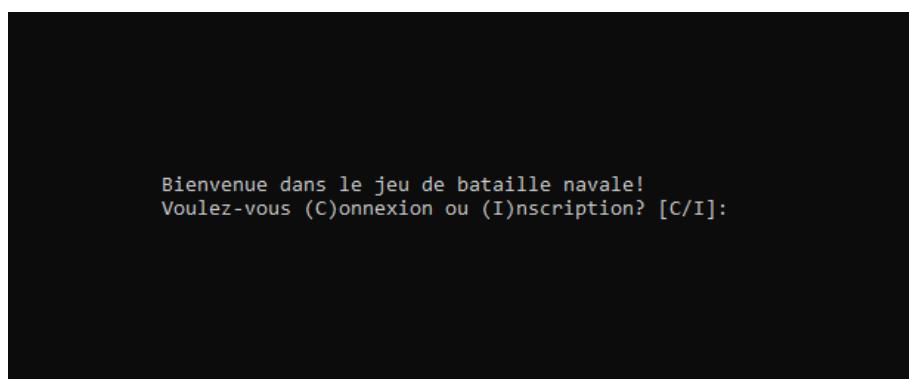


Requête SQL pour la création de la *player* et *playersData* :

```
CREATE TABLE player (
    idPlayers VARCHAR(255) PRIMARY KEY,
    mdp VARCHAR(25) NOT NULL,
    INDEX idx_idPlayers (idPlayers)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
CREATE TABLE playersData (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    score INT UNSIGNED NOT NULL CHECK (score >= 0 AND score <= 1000000),
    nbGames INT UNSIGNED NOT NULL,
    nbLostGames INT UNSIGNED NOT NULL,
    nbWonGames INT UNSIGNED NOT NULL,
    idPlayers VARCHAR(255) NOT NULL,
    FOREIGN KEY (idPlayers) REFERENCES player(idPlayers),
    INDEX idx_idPlayers (idPlayers)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Désormais nous pouvons créer un nouveau compte ou nous connecter.



Si je me connecte avec (**ID, mdp**) value (**snir, snir**)

```
Informations de l'utilisateur snir:
Score: 10500
Nombre de jeux joués: 43
Nombre de jeux perdus: 8
Nombre de jeux gagnés: 35
Votre Grille:           Grille Adversaire:
  0 1 2 3 4 5 6 7 8 9   0 1 2 3 4 5 6 7 8 9
  0 B B B 0 0 B B 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
  2 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  4 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  5 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  6 0 0 0 0 B B B B B 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  7 0 0 0 0 0 0 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  8 0 0 B B B B 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  9 0 0 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Entrez la ligne:

Désormais, toutes les actions du joueur auront des conséquences sur ses statistiques.

Par exemple, lorsque le joueur gagne, il remporte un bonus et le jeu incrémente son nombre de victoires. À l'inverse, si le joueur perd, alors on incrémente son nombre de défaites, *obj* est une instance de la classe *BsBDD* (BattleShip Base de données) :

```

bool CoreGame::estFinDuJeu() {
    bool joueurPerdu = true, adversairePerdu = true;

    for (int i = 0; i < nbLig; ++i) {
        for (int j = 0; j < nbCol; ++j) {
            if (grille[i][j] == typeCase::bateau) {
                joueurPerdu = false;
            }
            if (grilleAdversaire[i][j] == typeCase::bateau) {
                adversairePerdu = false;
            }
        }
    }

    if (joueurPerdu) {
        std::cout << espace << "Vous avez perdu la partie." << std::endl;
        obj.incrementNbLostGames();
    }
    else if (adversairePerdu) {
        std::cout << espace << "Vous avez gagn\u2022 la partie !" << std::endl;
        obj.BonusWin();
        obj.incrementNbWonGames();
    }
}

return joueurPerdu || adversairePerdu;
}

```

En conséquence, la boucle de jeu a elle aussi changé un petit peu.

```

void CoreGame::jouer() {
    obj.Connexion();
    system("cls");
    SautLaLigne
    obj.displayPlayerInfo();
    // Boucle principale du jeu, alternant entre le joueur et l'IA
    while (!estFinDuJeu()) {
        afficherBateauxCoules(); // a supprimer
        afficheGrille();
        std::pair<int, int> saisie = saisieJoueur(); // Utilisez le type explicite au lieu de 'auto'
        int ligne = saisie.first;
        int colonne = saisie.second;
        std::cout << espace3 << "L'adversaire joue...";
        //Sleep(1000);
        system("cls");
        std::cout << "MOI: " << std::endl << serialisationJoueur() << std::endl;
        std::cout << "ADVERSAIRE: " << std::endl << serialisationAdversaire();
        std::cout << std::endl;
        if (attaqueJoueur(ligne, colonne)) {
            std::cout << std::endl << std::endl << std::endl;
            std::cout << espace4 << "\u033[34mTouch\u2022!\u033[0m" << std::endl << std::endl;
        }
        else {
            std::cout << std::endl << std::endl << std::endl;
            std::cout << espace4 << "\u033[31mManqu\u2022!\u033[0m" << std::endl << std::endl;
        }
        attaqueIA();
    }
    obj.incrementNbGames();
}

```

Voici une version amélioré de la boucle de connexion :

```
void CoreGame::jouer() {  
  
    do {  
        obj.Connexion(); // affiche les cout de connexion et essaie de se connecter  
        system("cls"); // efface les elements de la console  
    } while (!obj.isConnect()); // on recommence tant que la connexion est pas valide  
  
    SautLaLigne  
    obj.displayPlayerInfo(); // on affiche les information du joueur connecté  
    std::cout << std::endl;  
}
```

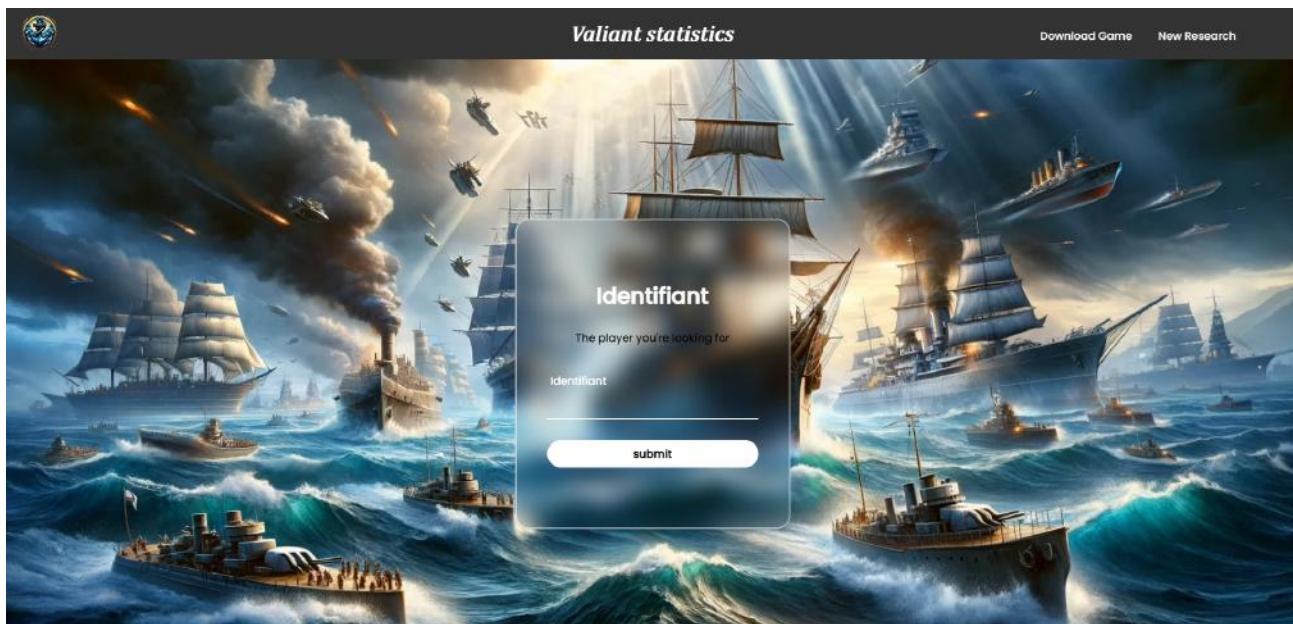
Cette boucle continue de tenter de se connecter en utilisant `obj.Connexion()` et efface la console après chaque tentative, jusqu'à ce que la méthode `isConnect()` de l'objet `obj` renvoie `true`, indiquant que la connexion a été établie avec succès.

Bataille Navale avec Extension Web

En complément de notre projet de bataille navale en C++ avec surcouche graphique SFML, j'ai créé un site web dédié. Ce site offre une plateforme pour visualiser les statistiques des joueurs de manière publique et facilite le téléchargement du jeu.

La création de ce site web représente une étape importante dans la valorisation de notre projet de bataille navale. Il ne se limite pas à être une simple extension de notre jeu, mais joue un rôle clé dans la construction d'une communauté active et dans la facilitation de l'accès au jeu. Ce développement web complète notre projet initial en C++, créant ainsi une expérience de jeu complète, de la partie technique à la partie communautaire.

Une page de recherche : (le fond est animé grâce à Adobe After Effect - 2h de travail)



Une page de résultat :

The screenshot shows a results page for a player named 'Valiant'. At the top, there's a logo and the title 'Valiant statistics'. Below the title, there are five performance metrics displayed in boxes: Score (8750), Nombre de parties jouées (38), Nombre de parties perdues (7), Nombre de parties gagnées (31), and K/D du joueur (4.43). The background of the page features a blurred image of a naval battle scene.

Page d'information et statistiques :

The screenshot shows the 'Valiant' game information and statistics page. At the top, there's a logo and the title 'Valiant statistics'. Below the title, there's a section titled 'Valiant' with a detailed description of the game: "Dive into the captivating world of 'Epic Naval Battles,' a real-time strategy game that reinvents the classic naval battle game for modern players. Compatible with Windows 8 and earlier, this game offers an immersive and strategic experience in the world of naval warfare." There's also a 'Key Features' section listing various game modes and mechanics. To the right of the text, there's a circular icon featuring an anchor and waves. The bottom of the page has a large 'User Statistics' section with a sub-section titled 'Advanced User Statistics Management' containing detailed information about data tracking and security.

User Statistics

Advanced User Statistics Management:

Epic Naval Battles is equipped with a sophisticated statistics tracking feature, allowing players to track and analyze their performance over time. This feature is based on a robust and secure database, designed to store a variety of information related to each user's gaming experience.

Database Features:

Storage of Statistics: The database records information such as the number of wins/losses, strategies used, the effectiveness level of different fleets, and much more.

Performance Analysis: Players can view their statistics to assess their progress, identify the strengths and weaknesses of their strategies, and adjust their approach for future battles.

Comparison with Other Players: Statistics also allow for comparison with other players in the rankings, thus promoting healthy and motivating competition.

Data Connection and Security:

To access these statistics, players must create an account and log in to the game. This step ensures not only the security and confidentiality of personal data, but also a personalized experience for each user.

User Account: Creating a user account is simple and secure. It allows saving game progress and accessing personal statistics from anywhere. **Data Protection:** We take data security very seriously. All information stored in the database is protected by advanced security measures to prevent any unauthorized access. By integrating these features, Epic Naval Battles offers a richer and more personalized gaming experience, allowing players to engage more deeply with their own performance and with the overall game community.



Page présentation du multijoueurs :

Multplayer Experience

Clean Graphics and Intense Multiplayer Experience:

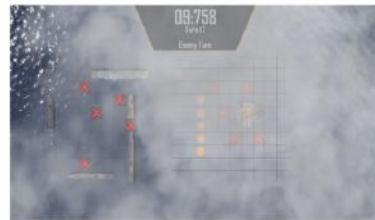
In "Epic Naval Battles," players are greeted with sleek and elegant 2D graphics. This visual approach offers a clear and immersive style, allowing players to fully focus on strategy and tactics without distraction. The simplicity of the graphics promotes a better understanding of game elements and a smooth user experience, even on older systems.

The game particularly shines in its multiplayer mode, where players can compete against real online opponents. This feature adds an extra dimension to the game, turning each match into a unique and captivating challenge. Players can test their skill and ingenuity against human opponents, each bringing their own style of play and strategies.

This level of real-time competition creates intense and memorable gameplay moments, where each decision can turn the tide of a battle.

The multiplayer mode of "Epic Naval Battles" is not just a platform to test individual skills, but also a way to connect with the global gaming community. Players can join teams, participate in tournaments, and share their experiences and strategies with other game enthusiasts.

In summary, "Epic Naval Battles" combines minimalist aesthetics with dynamic gameplay to offer an experience that is both visually pleasing and strategically demanding. Join the community and prove your skill in the art of naval warfare in this world of intense and strategic naval battles.



Page des termes de conditions d'utilisation :

Terms of Use

Terms of Use:

Acceptance of Terms: By installing, copying, or using this game, you agree to be bound by these terms and conditions.

Use License: This game is provided under a license, not for sale. This license allows you to use one copy of the game on a single computer or device at a time.

Restrictions: You may not distribute, sell, lease, sub-license, modify, decompile, disassemble, or create derivative works based on this game without prior written permission from the publisher.

Ownership: The game and all rights, including copyright and other intellectual property rights, belong to the publisher. This license does not transfer any ownership rights of the game to you.

Online Use: If the game offers online features, you must comply with all online conduct rules and not use the game for illegal activities or to transmit offensive content.

Updates and Modifications: The publisher reserves the right to update or modify the game at its discretion, without prior notification.

Warranty: The game is provided "as is" without any kind of warranty, explicit or implicit.

Limitation of Liability: The publisher will not be liable for any direct, indirect, incidental, special, punitive, or consequential damages resulting from the use or inability to use the game.

Termination: This license is effective until terminated. Your rights under this license will automatically terminate without notice from the publisher if you fail to comply with its terms.

Governing Law: This license shall be governed by and construed in accordance with the laws of the publisher's country.



Page de téléchargement :

Download the Game

Welcome aboard, Commander!

You are about to dive into the captivating world of "Epic Naval Battles," where strategy, tactics, and action meet in an intense maritime confrontation. Get ready for an epic adventure on the seas against opponents from around the world.

How to Download the Game:

Check System Requirements: Make sure your computer meets the minimum requirements for a smooth gaming experience. Click the Download Button: Find the download button on this page and click on it to start the process. Install the Game: Once the download is complete, open the installation file and follow the instructions on the screen to install the game on your computer.

System Requirements:

Operating System: Windows 8 or higher
Processor: i3-6100 or higher
RAM Memory: 4 GB DDR4 RAM or higher
Hard Disk Space: 1.5 GB or higher
Graphics Card: Intel HD Graphics 730 or higher

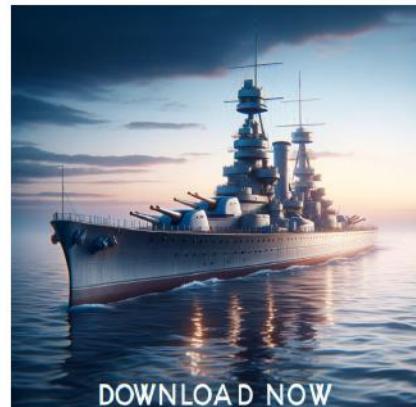
About the Game:

"Epic Naval Battles" offers strategic gameplay with sleek 2D graphics, developed in SFML, providing a clear and immersive visual experience. With its dynamic multiplayer mode, compete against real players, test your skills, and climb the rankings.

Key Features:

Join the Battle: Click the button below to start your download and join the battle to become the undisputed master of the seas!

[Download](#)
[Download_Zip](#)



DOWNLOAD NOW

J'ai créé l'installateur grâce au logiciel *inno setup*, qui permet via le langage de programmation *Pascal* de créer un installateur.



Bataille Navale en C++ avec Surcouche Graphique SFML

Introduction

Dans le cadre de notre mini-projet, notre groupe a développé une version améliorée du jeu classique de bataille navale. En utilisant la bibliothèque SFML, nous avons ajouté une surcouche graphique pour transformer le jeu en console en une expérience visuelle enrichie.

Surcouche Graphique SFML

SFML nous a permis de créer une interface utilisateur graphique intuitive, affichant des grilles de jeu dynamiques, des navires, et les résultats des actions de jeu. Cette intégration a rendu le jeu plus interactif et visuellement attrayant.

Mode Multijoueur

Nous avons implémenté un mode multijoueur utilisant des sockets TCP, permettant aux joueurs de se connecter et de s'affronter en temps réel. Cette fonctionnalité apporte une dimension compétitive et sociale au jeu.

Système de Comptes et Statistiques

Un système de comptes permet aux joueurs de suivre leurs statistiques personnelles, telles que les victoires, les défaites et les performances. Ces données enrichissent l'expérience de jeu en offrant un suivi des progrès et en stimulant la compétition.

Boutique d'Items

La boutique offre de nouvelles photos de profil pour rendre votre profil plus attrayant.

Expérience Utilisateur Améliorée

Des animations, effets sonores et visuels de haute qualité ont été intégrés pour améliorer l'immersion. Chaque élément contribue à une expérience utilisateur plus riche et plus engageante.

Je ne vais pas développer toute la partie de la création du graphique car ça serait trop long et que ce n'était pas demandé à l'origine. Ce que je viens de présenter correspond à une semaine de travail sur 4. Toutefois, je pense que survoler les étapes peut être intéressant.



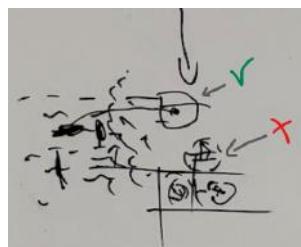
En gros ça c'est résumé à réaliser plusieurs classes pour rester organisé, la plupart de nos classe utilise le polymorphisme. Ce qui a été très pratique.

Nous avons fait nos propre classe SFML.

Exemple de la création d'une image pour le SFML :

Tout d'abord, avant de créer il faut être sûr de ce que l'on souhaite. Nous avions fait un brainstorming avant de commencer à coder, alors je sais ce que je dois faire.

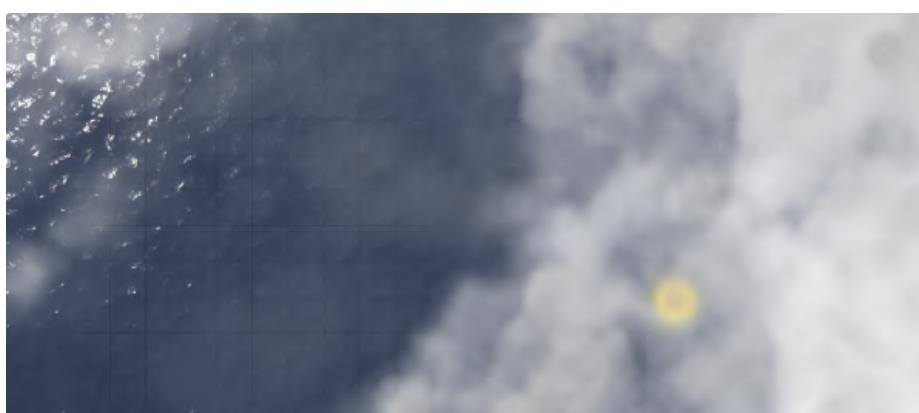
L'idée est simple, reproduire une lueurs de feu dans la case de l'adversaire, plus précisément à l'endroit où le bateau à été touché. Voici le plan :



Comme le montre le dessin du brainstorming (je sais qu'il est pas compréhensible) nous avons eu deux idée, la première été de d'ouvrir le nuage a l'endroit tiré pour que le joueurs puisse voir le feu. Et la seconde était de créer une lueur de feu.

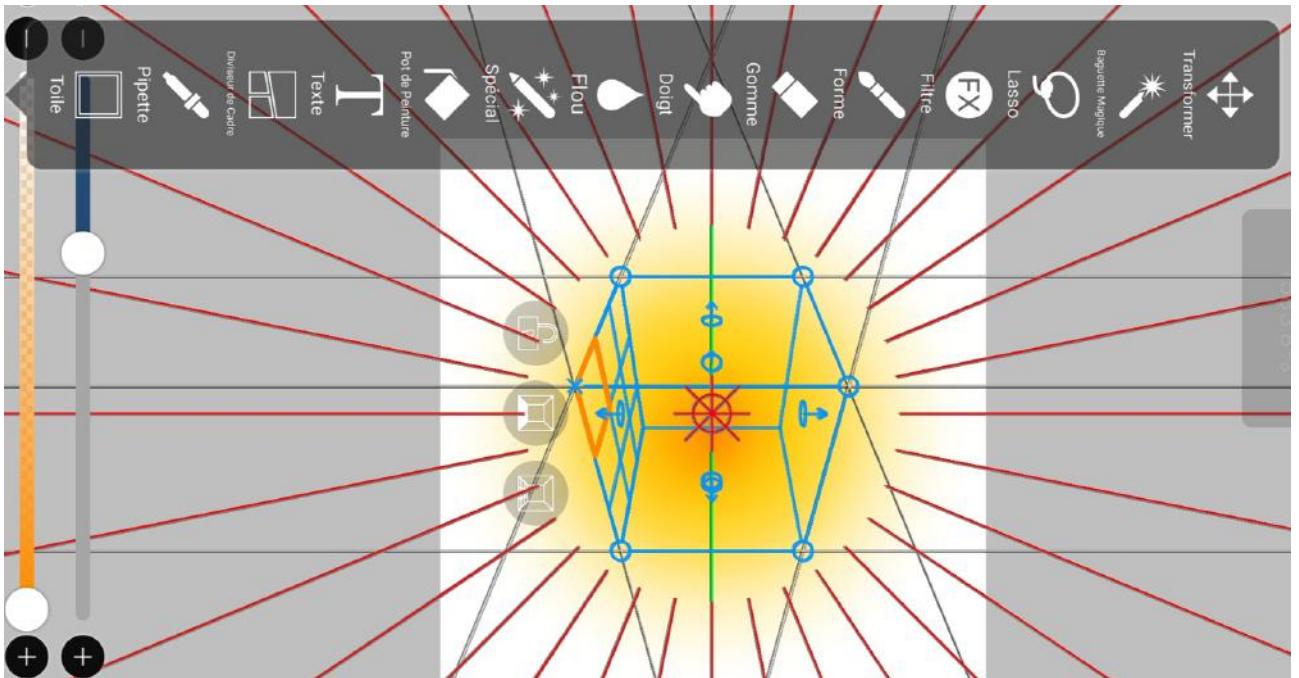


Nous avons alors essayé de créer des nuages capables de laisser des troue quand un tire dessus. Toutefois le résultat était horrible et nous aurait demandé trop de temp pour faire quelques chose de jolie



Nous avons alors retenu l'idée des lueurs de feu.

Plus qu'à créer l'objet :



Une fois créé, je peux désormais l'intégrer à ma classe.

Dans la classe settings, qui sert globalement à charger tous les chemins, à mettre toutes les tailles ou autres je vais créer un struct de `sfxSettings`, et y ajouter le chemin de l'image en y attribuant le nombre d'image disponible de cette référence. En effet, c'est des images animé alors elle ont plusieurs versions d'elle même, ici c'est huit.

```
const std::string enemyFirePath = "ressources/UI/vfx/vfx_ennemyFire";
const int nbPlayerFire = 8;
```

Maintenant, dans la classe `GameVfx` on ajoute l'attribut correspondant :

```
std::vector<sf::Texture> enemyFireTextures;
```

Puis on incrémente le vecteur avec nos images :

```
for (int i = 0; i < gameVfxSettings.nbEnemyFire; i++)
{
    sf::Texture t;
    t.loadFromFile(gameVfxSettings.enemyFirePath + std::to_string(i) + ".png");
    enemyFireTextures.push_back(t);
}
```

Ensuite la méthode ici permet de faire appel à l'animation, (pas détaillé ici) :

```
void GameVfx::CreateFireCell(int x, int y, bool isEnnemy)
{
    sf::Vector2f pos = getGridPos(isEnnemy);

    std::vector<sf::Texture*>* textures;
    isEnnemy ? textures = &enemyFireTextures : textures = &playerFireTextures;
```

Et on peut appeler la méthode au bon moment (quand le bateaux est touché) :

```
//if oponent no hits
if (attackCell != BattleshipCore::CellType::hit)
{
    gameInfoPanel->updateGameInfo("Your Turn");
    gameState = GameState::Attacking;
    application->fxobj->createSfx(SfxManager::sfx::water);
}
else {
    gameVfx->CreateFireCell(attckPos.x, attckPos.y, false);
    application->fxobj->createSfx(SfxManager::sfx::explosion);
    if (battleShipCore.CheckIfBoatDown(attckPos.y, attckPos.x, false, false, -1, -1))
        application->fxobj->createSfx(SfxManager::sfx::sinkBoat);
}
```

On peut voir par exemple une explosion tout juste enclenché et une lueur de feu sous les nuages qui cache les bateaux de l'adversaire :



Pour faire bouger deux nuages de manière consécutive de sorte à ce qu'il y est toujours un nuages, on peut faire comme ceci :

création du struct adapté dans *sfxSettings* :

```
struct CloudSettings {
    const sf::Vector2f cloudSize = sf::Vector2f(2500, 2500);
    const sf::Vector2f Origin = sf::Vector2f(cloudSize.x / 2, cloudSize.y / 2);
    float anglesDegresAnimate = 0;
    const sf::Vector2f cloudPositionAnimate[2] = { sf::Vector2f(0, -400), sf::Vector2f(2100, -400) };

    const int explodeOpacitySpeed = 10;
    const float minExplodeOpacity = 0;

    const std::string cloudImgPath = "ressources/UI/cloud.png";
    const std::string cloudImgPath2 = "ressources/UI/cloud2.png";
};
```

J'ajoute les deux attribut nécessaire à la manipulation de mon nuage :

```
sf::Texture cloudTextureAnimate;
std::vector<EntityRectangle> cloudsStaticAnimate;
```

Je charge la texture a part :

```
cloudTextureAnimate.loadFromFile(cloudSettings.cloudImgPath);
```

Et je push mon *EntityRectangle* dans mon vecteur d'*EntityRectangle*, la classe *EntityRectangle* est une classe spécifique à notre programme qui sert à créer les éléments visuels du jeu.

```
for (int i = 0; i < 2; i++)
{
    cloudsStaticAnimate.push_back(EntityRectangle(cloudSettings.cloudSize,
        cloudsStaticAnimate.at(i).setTexture(&cloudTextureAnimate));
}

cloudSettings.cloudPositionAnimate[i], cloudTextureAnimate, sf::Color(255, 255, 255, 172)));
```

la ligne sélectionnée en bleu est la même ligne que j'ai coupé.

Très important il est nécessaire de dessiner les nuages, ici ce sont des vecteurs alors on utilise une boucle :

```
// Dessiner les nuages
for (int i = 0; i < cloudsStaticAnimate.size(); i++)
{
    cloudsStaticAnimate.at(i).draw(window);
}
```

Et pour la fin, je créer l'animation du nuages dans la boucle *Update* :

```
void update() {
    // Mettre à jour la position des nuages mobiles
    for (size_t i = 0; i < cloudsStaticAnimate.size(); i++) {
        sf::Vector2f position = cloudsStaticAnimate[i].getPosition();

        // Logique de mouvement spécifique
        if (position.x < -2500) {
            cloudsStaticAnimate[i].setPosition(sf::Vector2f(1000, -400));
        }
        cloudsStaticAnimate[i].move(-0.040, 0);
    }
}
```

Et désormais, un nuage bouge et crée une occlusion ambiante dans le jeu, ce qui crée une meilleure atmosphère, une meilleure ambiance.

Toutefois les animations ont rencontré des problème :

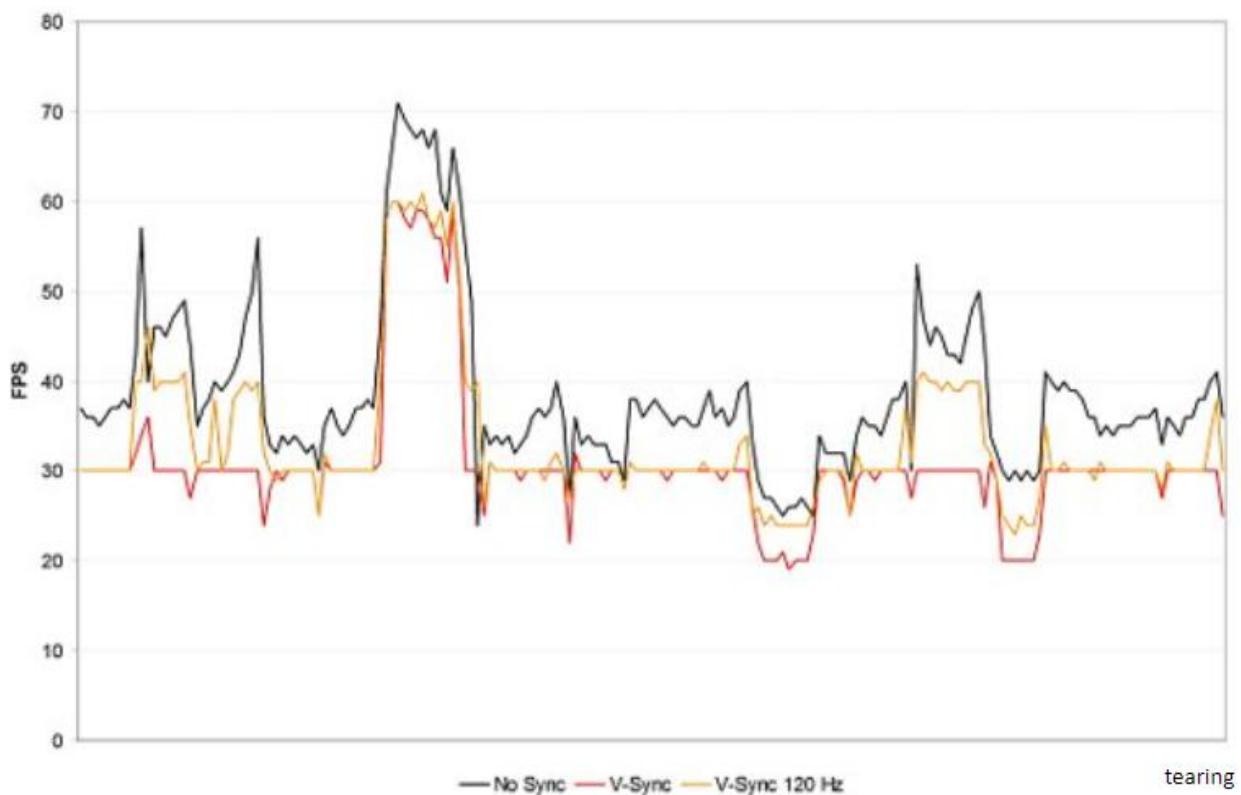
Le jeu fonctionnait bien sur un écran 60Hz mais était super accéléré sur un écran 165Hz :



Et le pourquoi du comment était évident. La boucle du jeu qui réactualise tous les événements du jeu était synchronisée à la vitesse à laquelle le processeur envoyait les images au GPU.

Pour éviter les effets de tearing (ou déchirement d'images en français) nous avons opté pour une solution simple et efficace.

Exemple de stabilité suivant si le tearing fait effet ou pas (v-sync on et off)



Le V-Sync sert à annuler le tearing en brident le GPU

En conclusion, nous allons manuellement brider l'affichage maximum d'images par seconde. Le jeu s'actualise seulement et uniquement grâce à la boucle de jeu. donc, si nous stoppons la boucle le temps de la fréquence égale à du 60 FPS (frame rate per second), donc du 60Hz, alors on aura bien 60 images par seconde. cela représente une période de 16.67ms soit 0.01667 secondes.



Voici le code à mettre à la fin de la boucle :

```
//FPS
sf::Time elapsedTime = clock.getElapsedTime();

// Pour 60 FPS, le temps par frame devrait être environ 16.67 millisecondes
const int frameTime = 1000 / 60; // Environ 16.67 millisecondes

if (elapsedTime.asMilliseconds() < frameTime) {
    sf::sleep(sf::milliseconds(frameTime) - elapsedTime);
}

clock.restart();
```

Ce code sert donc à réguler le taux de rafraîchissement du jeu à 60 images par seconde (FPS). Voici une explication simplifiée de ce qu'il fait :

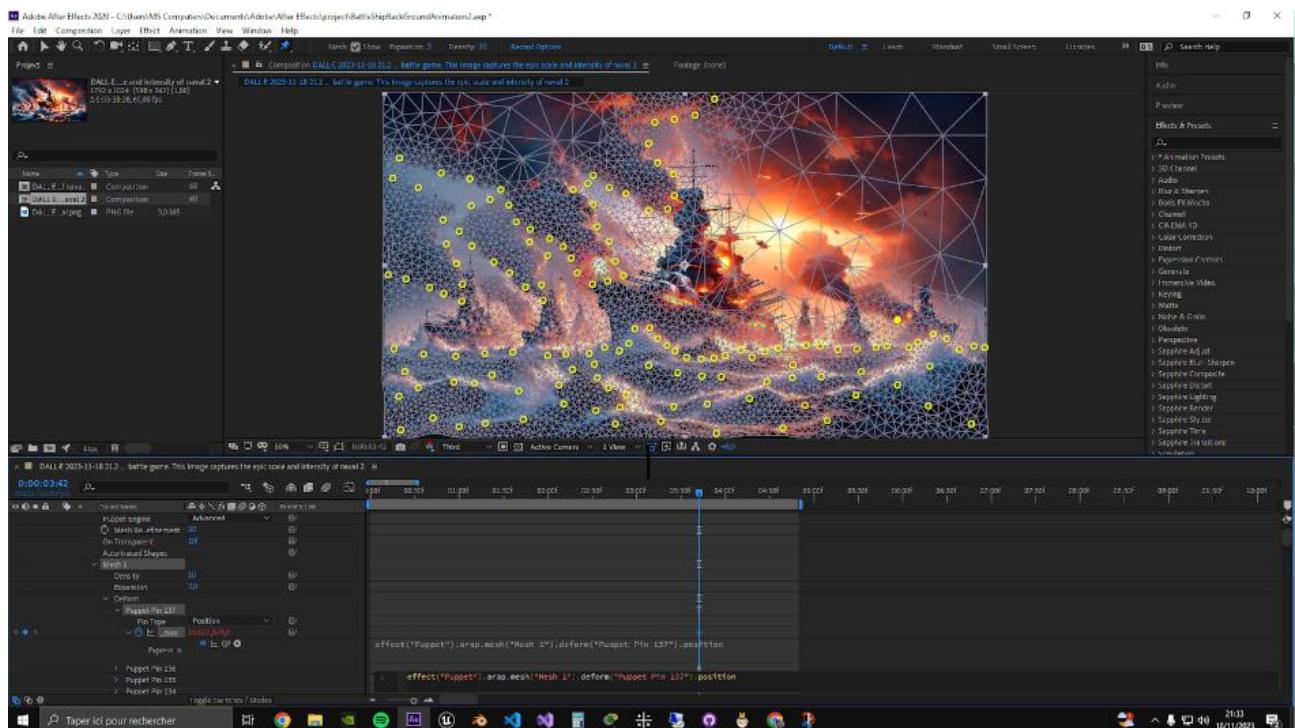
- **Mesure du Temps Écoulé** : Le code commence par obtenir le temps écoulé depuis la dernière mise à jour de l'horloge (`clock.getElapsedTime()`). Cela indique combien de temps s'est écoulé depuis la dernière frame (image).
- **Calcul du Temps par Frame** : Il définit `frameTime` comme le temps que chaque frame devrait prendre pour atteindre 60 FPS. Pour 60 FPS, chaque frame devrait durer environ 16.67 millisecondes.
- **Contrôle du Taux de Rafraîchissement** : Le code vérifie si le temps écoulé depuis la dernière frame est inférieur à 16.67 millisecondes. Si c'est le cas, cela signifie que la frame s'est exécutée trop rapidement pour atteindre 60 FPS. Pour corriger cela, le code fait "dormir" le programme (`sf::sleep`) pendant la durée restante pour atteindre 16.67 millisecondes.
- **Réinitialisation de l'Horloge** : Après cette pause, l'horloge est réinitialisée (`clock.restart()`), marquant le début du calcul du temps pour la prochaine frame.

En résumé, ce code s'assure que chaque frame dure suffisamment longtemps pour maintenir un taux de rafraîchissement stable de 60 FPS, en introduisant une pause si nécessaire pour ralentir le taux de rafraîchissement.

Création de l'images d'accueil :

Ca n'a aucun lien avec le BTS snir mais je trouve que c'est intéressant alors j'en parle sur une page :

Les animations de mes images de fond comme celle ci est celle du site on était créer via Adobe After Effect, il suffit avec l'outil puppet (qui dans ce contexte peut se traduire par punaise) de marquer des points d'ancrage ainsi que des points d'événement. Ensuite il est possible via le langage de programmation JavaScript de créer des événements sur les puppets, ce qui peut créer des animations.

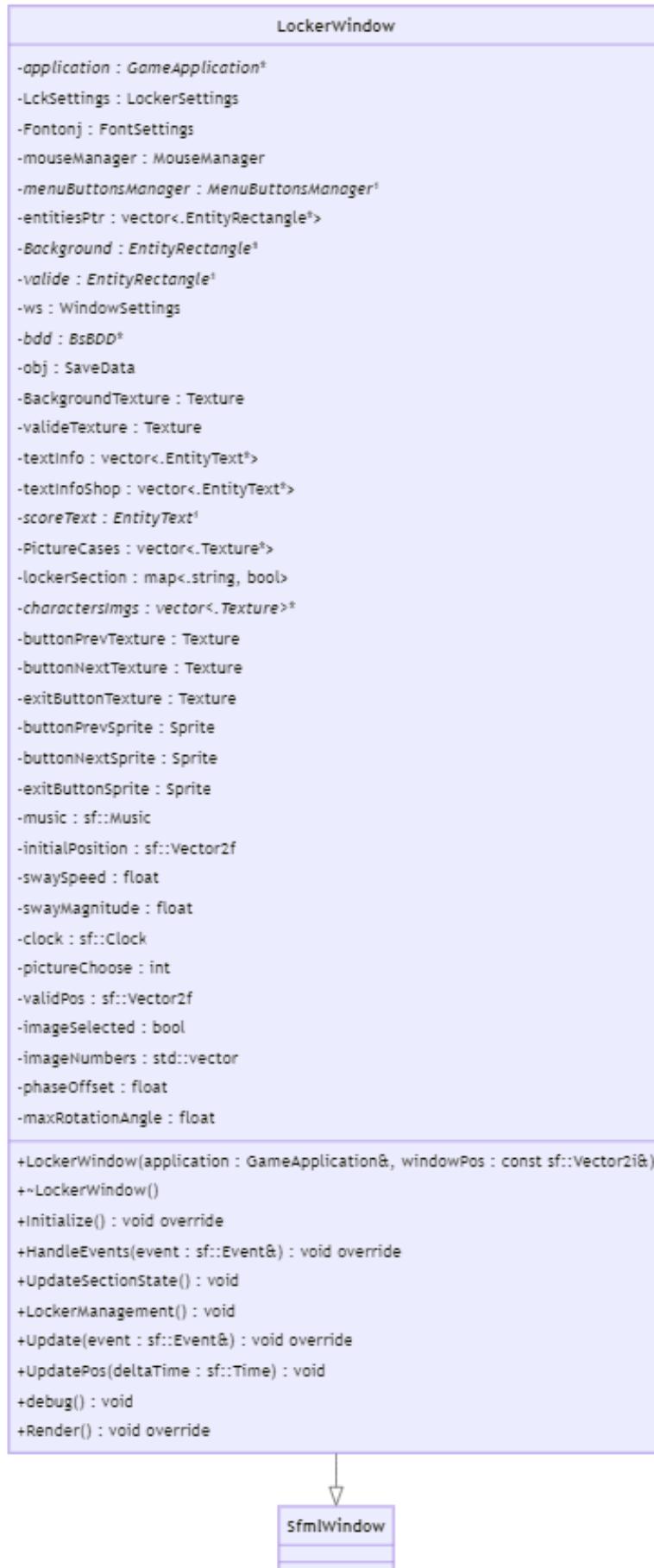


Ensuite j'extract toutes mes images et je peux les faire jouer dans la classe *MenuWindow* via des boucles.

Affichage des items :

Dans les idées du jeu, la programmation de la boutique était un peu flou, est-ce que nous allons faire des affichages au scroll ? Si oui vertical, horizontal ? Ou bien on pouvait juste faire une page simple ? Étant chargé de la création de la boutique, j'ai décidé de créer une boutique avec plusieurs sections. Une section pour chaque groupe de "photo de profil".

Voici le diagramme UML de la classe *LockerWindow* :



Je passe les détails, mais pour ce qui est de l'affichage des items, voici comment cela fonctionne :

```
// Définitions des marges et taille des images
float new_image_width = 304.8f; // Largeur de l'image après la mise à l'échelle
float new_image_height = 304.8f; // Hauteur de l'image après la mise à l'échelle
float margin_each_side = 27.8f; // Marge sur les côtés
float margin_each_top_bottom = 65.8f; // Marge en haut et en bas

// Tableau indiquant l'indice de début de chaque section
int section_starts[] = { 0, 5, 9, 12, 19, 23, 29, 32, 40 };
std::vector<sf::Vector2f> imagePositions;

// Boucle sur toutes les images
charactersImg = &application->getCharactersImg();
for (int i = 0; i < 48; ++i) {
    // Trouvez la section actuelle en fonction de l'indice 'i'
    int section = 0;
    while (i >= section_starts[section + 1]) {
        section++;
    }

    // Calculez la position dans la section (l'indice relatif à la section)
    int index_in_section = i - section_starts[section];

    // Calculez la position en x et y en tenant compte du début d'une nouvelle section
    float pos_x = (index_in_section % 4) * (new_image_width + margin_each_side) + margin_each_side;
    float pos_y = (index_in_section / 4) * (new_image_height + margin_each_top_bottom) + margin_each_top_bottom;

    // Ajoutez la nouvelle entité à la fin du vecteur
    entitiesPtr.push_back(new EntityRectangle(sf::Vector2f(280, 280), sf::Vector2f(pos_x, pos_y), charactersImg->at(i)));
    textInfoShop.push_back(new EntityText(LckSettings.font, sf::Vector2f(pos_x, pos_y), LckSettings.characterSize, std::to_string(imageNumbers[i])));
}
```

Le code est conçu pour organiser et afficher une série d'éléments graphiques. Le code vise à placer ces images dans plusieurs sections distinctes sur l'écran, en veillant à ce que chaque nouvelle section commence en haut à gauche de l'écran. Voici une explication détaillée de la logique et du fonctionnement de ce code :

Structure et Organisation

Définition des Marges et Tailles d'Images :

- Les variables *new_image_width* et *new_image_height* définissent la largeur et la hauteur de chaque image après mise à l'échelle.
- Les marges (*margin_each_side* et *margin_each_top_bottom*) sont utilisées pour définir l'espace entre les images, aussi bien sur les côtés qu'en haut et en bas.

Tableau des Débuts de Section :

- Le tableau *section_starts* indique les indices de début pour chaque section. Par exemple, si *section_starts[1] = 5*, cela signifie que la deuxième section commence à l'indice 5 dans le vecteur d'images.

Boucle sur Toutes les Images :

- Le vecteur *charactersImg* contient les images à afficher.
- La boucle parcourt ce vecteur pour placer chaque image dans la section appropriée.

Logique de Placement

Détermination de la Section Actuelle :

- Pour chaque image, le code détermine d'abord dans quelle section elle doit apparaître. Cela est accompli en comparant l'indice de l'image (*i*) avec les valeurs dans *section_starts*.

Calcul de la Position de l'Image :

- *index_in_section* : Il s'agit de l'indice de l'image dans sa section spécifique.
- *pos_x* et *pos_y* : Ces variables calculent la position exacte de l'image sur l'écran en fonction de *index_in_section*, en tenant compte des marges et de la taille des images.
- Le calcul de *pos_x* et *pos_y* permet de s'assurer que chaque nouvelle section commence en haut à gauche et que les images sont disposées en grille.

Ajout des Images au Vecteur :

- Chaque image est ensuite ajoutée au vecteur *entitiesPtr* avec sa position calculée. Ceci assure que chaque image est positionnée correctement sur l'écran selon la section à laquelle elle appartient.

Ajout des Textes Correspondants :

- Pour chaque image, un texte est également créé et ajouté au vecteur *textInfoShop*. Ce texte est positionné près de l'image correspondante pour fournir des informations supplémentaires (comme le prix ou le nom de l'image).

Conclusion

Ce code permet donc de gérer efficacement la disposition visuelle d'une collection d'images dans un environnement graphique, avec une organisation claire en sections distinctes. Chaque section débute en haut à gauche, créant une présentation structurée et ordonnée pour l'utilisateur.

Pour gérer l'affichage des items en fonctions des sections, j'utilise des *std::map* :

```
for (int i = 0; i < 10; ++i) {
    std::string key = "section" + std::to_string(i);
    lockerSection[key] = false;
}
lockerSection["section0"] = true;
```

Chaque section est assimilée à une clé.

Et bien évidemment j'ai donc un attribut de référence :

```
int section_starts[9] = { 0, 5, 9, 12, 19, 23, 29, 32, 40 };
```

Il indique chacun des début de mes sections.

J'ai donc créer 2 boutons qui appelle la méthodes *UpdateSectionState* :

```
// Vérifier le clic sur le bouton précédent
if (buttonPrevSprite.getGlobalBounds().contains(mousePosition.x, mousePosition.y)) {
    currentSectionIndex = std::max(-1, currentSectionIndex - 1); // Ne pas aller en dessous de 0
    UpdateSectionState();
}

// Vérifier le clic sur le bouton suivant
else if (buttonNextSprite.getGlobalBounds().contains(mousePosition.x, mousePosition.y)) {
    currentSectionIndex = std::min(8, currentSectionIndex + 1); // Ne pas dépasser 8
    UpdateSectionState();
}
```

Cette méthode permet de passer en *false* la section précédente et en *true* la nouvelle section :

```
void LockerWindow::UpdateSectionState() {
    // Mettre toutes les clés à false
    for (auto& pair : lockerSection) {
        pair.second = false;
    }

    // Mettre la clé de la section actuelle à true
    std::string key = "section" + std::to_string(currentSectionIndex + 1);
    lockerSection[key] = true;
}
```

L'handle event du clique:

```
if (event.type == sf::Event::MouseButtonPressed) {
    if (event.mouseButton.button == sf::Mouse::Left) {
        sf::Vector2i.mousePosition = sf::Mouse::getPosition(window);

        for (int i = p; i < p + l; ++i) {
            if (i < entitiesPtr.size()) {
                auto entity = entitiesPtr.at(i);
                if (entity->getShape().getGlobalBounds().contains(mousePosition.x, mousePosition.y)) {
                    std::cout << "Le prix de la requette vaut : " << imageNumbers[i] << " à l'emplacement : "
                        << i << " du vecteur, et le score de la bdd vaut : " << stoi(application->getBddObj().getScore()) << std::endl;
                    if (imageNumbers[i] <= stoi(application->getBddObj().getScore())) {
                        pictureChoose = i;
                        bdd->setIdPicture(pictureChoose);
                        std::cout << "Image " << pictureChoose << " cliquée" << std::to_string(bdd->getIdPicture()) << std::endl;
                        // Mettre à jour validPos avec la position de l'image cliquée
                        validPos = entity->getPosition(); // Assurez-vous que votre entité a une méthode getPosition()
                        application->fxobj->createSfx(SfxManager::sfx::click);
                        application->fxobj->setSfxVolume(100);
                        // Mettre à jour la position de l'entité 'valide' avec la nouvelle position
                        valide->setPosition(validPos);
                        imageSelected = true;
                        break;
                    }
                    else
                        application->fxobj->createSfx(SfxManager::sfx::wrong);
                }
            }
        }
    }
}
```

Ci-dessous, la méthode qui dessine les items en fonction des sections en *true*

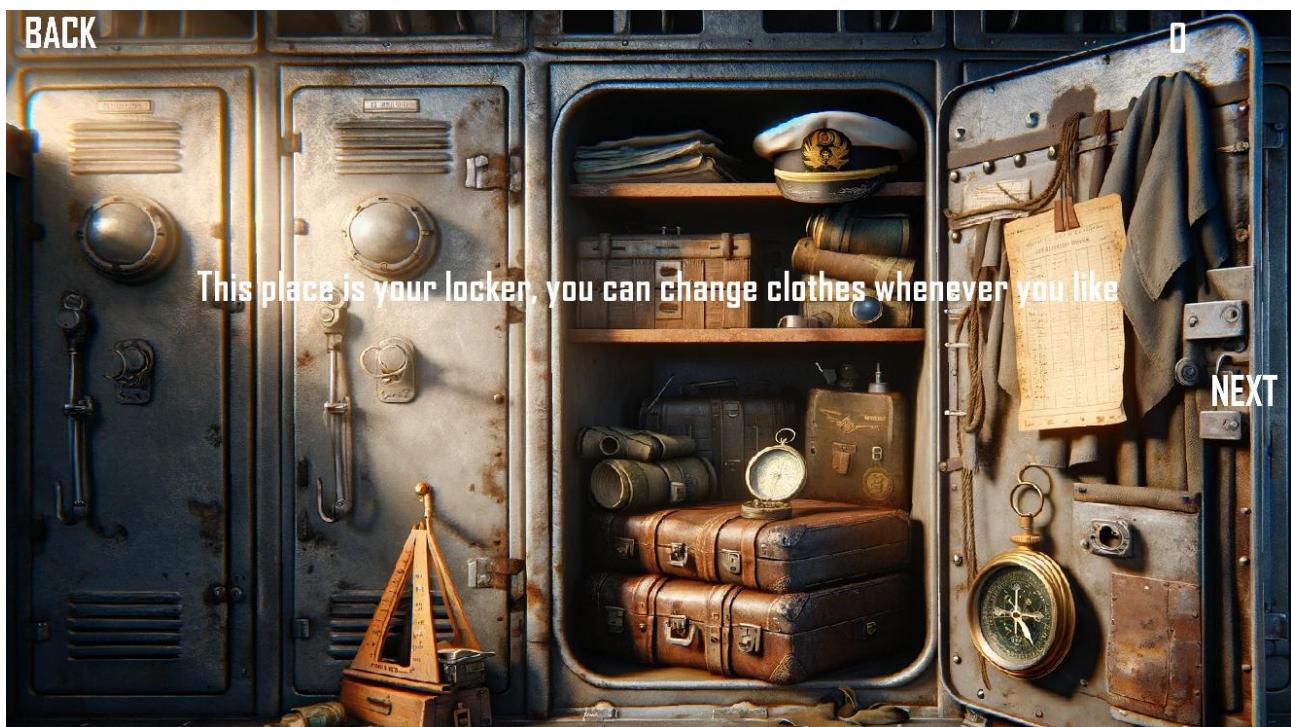
```
void LockerWindow::LockerManagement() {

    if (lockerSection["section0"]) {
        p = 0; l = 5, s = 0;
    }
    else if (lockerSection["section1"]) {
        p = 0; l = 5, s = 1;
    }
    else if (lockerSection["section2"]) {
        p = 5; l = 4, s = 2;
    }
    else if (lockerSection["section3"])
        p = 9, l = 3, s = 3;
    else if (lockerSection["section4"])
        p = 12, l = 7, s = 4;
    else if (lockerSection["section5"])
        p = 19, l = 4, s = 5;
    else if (lockerSection["section6"])
        p = 23, l = 6, s = 6;
    else if (lockerSection["section7"])
        p = 29, l = 3, s = 7;
    else if (lockerSection["section8"])
        p = 32, l = 8, s = 8;
    else if (lockerSection["section9"])
        p = 40, l = 8, s = 9;

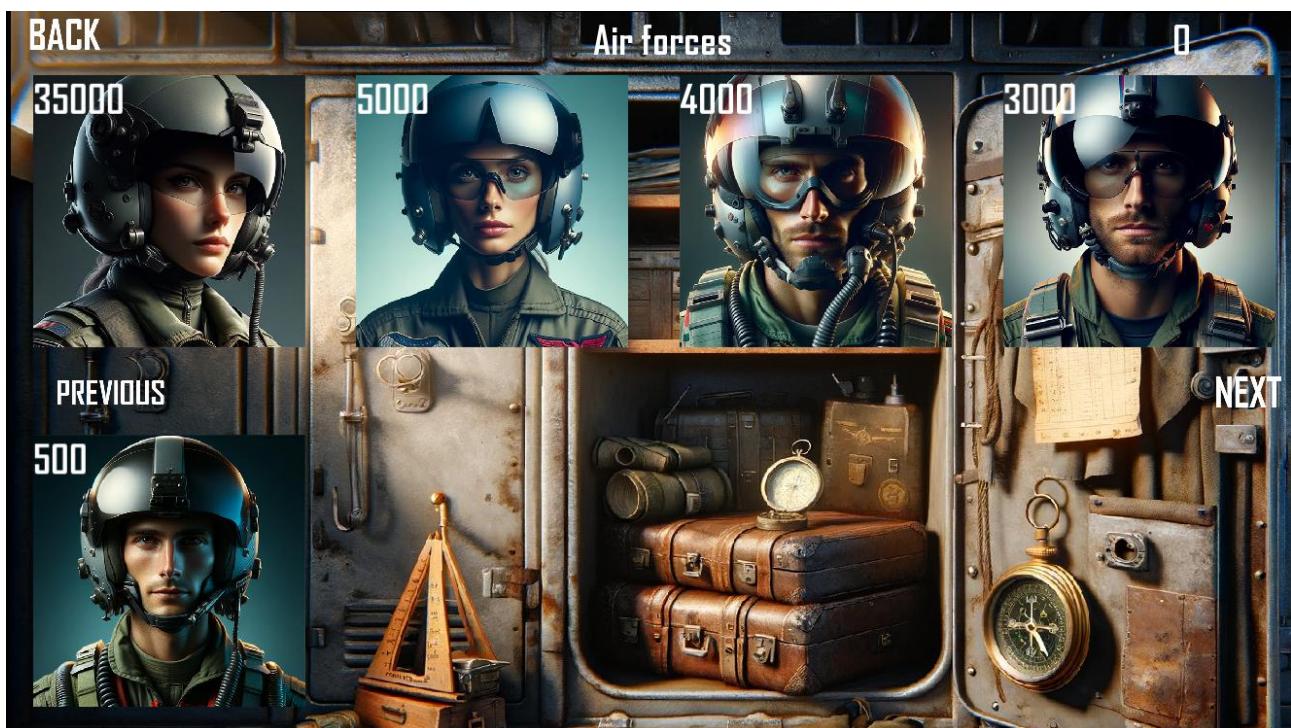
    if (lockerSection["section1"] || lockerSection["section2"] ||
        lockerSection["section3"] || lockerSection["section4"] ||
        lockerSection["section5"] || lockerSection["section6"] ||
        lockerSection["section7"] || lockerSection["section8"] ||
        lockerSection["section9"]) {
        for (int i = p; i < p + l; ++i) {
            if (i < entitiesPtr.size()) {
                entitiesPtr.at(i)->draw(window);
                if (i < textInfoShop.size())
                    textInfoShop.at(i)->draw(window);
            }
        }
        textInfo.at(s - 1)->draw(window);
    }

    if (lockerSection["section0"])
        textInfo.at(9)->draw(window);
}
```

Voici une démonstration de la section 0 (avec fond animé + son ambiance) :



Section 1 :



Section 4 :



Section 6 :



Etc, il y en a neuf.

Je passe à côté de centaines de choses (forcément il y a 40 classes) alors je préfère m'arrêter ici car ça devient trop long.

Conclusion

En conclusion, la réalisation de ce jeu de bataille navale en multijoueur a été une expérience enrichissante et formatrice. En utilisant le langage C++ et la bibliothèque SFML, nous avons pu approfondir nos compétences en programmation et en développement de jeux vidéo. Ce projet nous a permis de comprendre l'importance de la collaboration et de la communication au sein d'une équipe, essentielle pour la réussite d'un projet informatique complexe.

En outre, nous avons été confrontés à divers défis techniques, notamment la gestion du réseau pour le multijoueur, l'optimisation des performances et l'interface utilisateur intuitive. Surmonter ces obstacles nous a apporté une expérience précieuse et des connaissances que nous pourrons appliquer dans nos futures carrières professionnelles.

Nous tenons à exprimer notre gratitude envers tous ceux qui ont contribué à ce projet. Nous adressons un remerciement particulier aux enseignants qui nous ont apporté une aide précieuse, notamment M. Dartois pour ses conseils et le prêt de ses serveurs personnels, ainsi qu'à M. Dauriac pour nous avoir permis d'obtenir une base de données sur le serveur du lycée, et pour leur soutien tout au long du processus.

Ce projet s'inscrit parfaitement dans le cadre de notre BTS, car il nous a permis de mettre en pratique les théories et les techniques apprises en classe dans un contexte réel et tangible. Nous sommes fiers du produit final et sommes convaincus que cette expérience contribuera de manière significative à notre développement en tant que professionnels de l'informatique.

Raphael Laurent

IV/Journal de B.Traian :

A/Création d'une classe chargement de Json

Nous souhaitions au début faire un système permettant de changer la langue du jeu. Pour savoir si c'était possible, j'ai le premier jour crée une classe en utilisant la bibliothèque [nlohman json](#) pour charger les textes depuis un fichier .json correspondant à la langue choisie.

```
class Localization {  
  
private:  
    /**  
     * @brief Contains text translations loaded from JSON files.  
     * The key represents the text identifier, and the value is the translated text.  
     */  
    std::unordered_map<std::string, std::string> translations;  
  
public:  
    /**  
     * @brief Constructor of the Localization class.  
     * Initializes the translations map by loading data from a language-specific JSON file.  
     *  
     * @param language The language for which to load the translations (e.g., "fr", "en"). If "", then do not initialize.  
     */  
    Localization(const std::string& language = "");  
  
    /**  
     * @brief initializes translations by reading the JSON associated with the name.  
     */  
    void init(const std::string& language);  
  
    /**  
     * @brief Retrieves a translated string corresponding to the given key.  
     *  
     * @param key The key identifying the text to retrieve.  
     * @return std::string The translated text if the key is found, otherwise an error message.  
     */  
    std::string getString(const std::string& key);  
  
private:  
    /**  
     * @brief Loads translation data from a JSON file.  
     * Reads the JSON file corresponding to the specified language and fills the 'translations' map.  
     *  
     * @param language The language for which the JSON file should be loaded.  
     */  
    void loadLanguageFile(const std::string& language);
```

Figure 1 : Fichier Déclaration

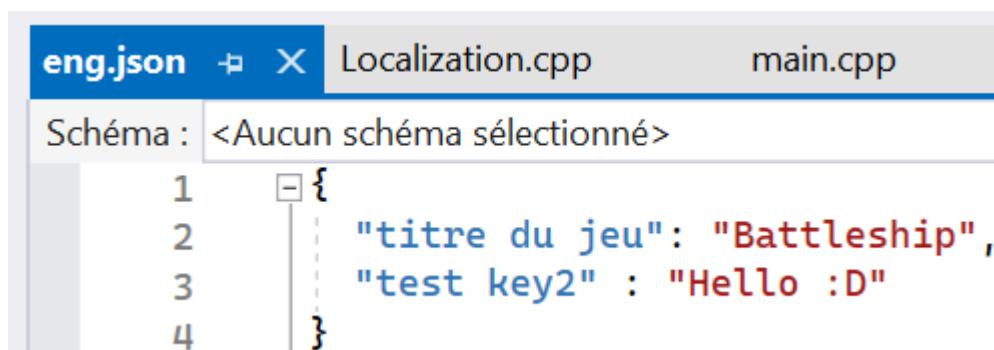


Figure 2 : Contenu d'un fichier json

```

int main()
{
    Localization l("fr");
    std::cout << l.getString("titre du jeu") << std::endl;

    Localization l2("eng");
    std::cout << l2.getString("titre du jeu") << std::endl;

```

Figure 3 : Utilisation de la classe

On aura en sortie :

Bataille navale
Battleship

Cependant, plus tard, pour l'intégration graphique nous avons utilisé des images contenant du texte. Pour éviter de perdre trop de temps à l'adaptation d'images, nous avons abandonné ce système.

B/Création de la base réseau

1/Structure réseau

J'ai commencé par créer le système réseau du jeu :

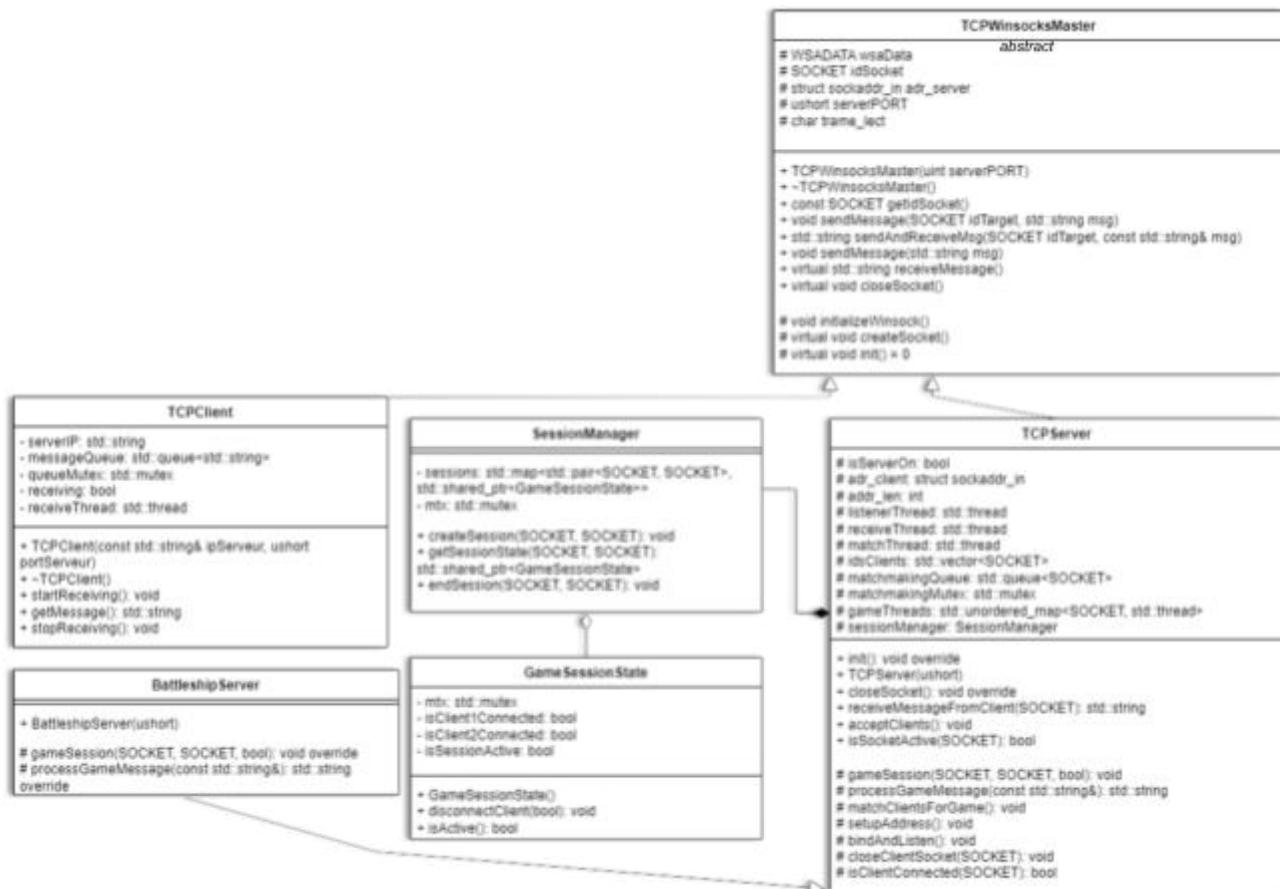


Figure 4 : Diagramme de classes de la partie réseau

Prenons les classes dans l'ordre, tout d'abord j'ai commencé par TCPWinsockMaster qui est la classe mère abstraite. Elle permet d'initialiser Winsocks et possède les méthodes communes comme "receiveMessage()" et contient des attributs communs comme le numéro de socket et le port du serveur. Elle est abstraite car sa méthode init() est virtuelle pure. Cela s'explique par le fait que le client et le serveur n'ont pas la même initialisation. Le serveur doit bind son port et se mettre en écoute tandis que le client se connecte au serveur.

Cette classe est dérivée pour créer une classe TCPClient et TCPServer, répondant ainsi au cahier des charges demandant une classe commune entre le joueur et le serveur. Le TCPClient est générique et peut être réutilisé dans d'autres projets.

Le TCPServer devait l'être aussi, mais j'ai codé la partie matchmaking à l'intérieur, il faudrait revisiter le code. La classe fille de TCPServer : BattleshipServer s'occupe de la session de jeu (il aurait dû gérer aussi le matchmaking). Ces 2 fonctionnalités sont traitées dans un thread chacun que nous détaillerons par la suite.

Suite à des problèmes de crash du serveur lorsqu'un client quittait brutalement le jeu pendant la partie ou le matchmaking. J'ai réalisé 2 classes supplémentaires (GameSessionState et SessionManager) afin d'augmenter la résistance du serveur. Elles seront détaillées un peu plus bas.

2/Fonctionnement de l'acceptation d'un client

Le matchmaking doit lier les clients deux par deux. Pour cela, on le serveur en écoute va ajouter dans un std::queue les clients qui se connectent au serveur.

Un std::queue est une file d'attente reprenant les principes d'un std::vector mais optimisé pour la gestion de file d'attente comme ici avec le matchmaking.

Pour accepter les connexions, nous utilisons la méthode "select()" de winsocks. Cependant, c'est une fonction bloquante. C'est-à-dire que le code s'arrête en attendant une connexion. Pour résoudre ce problème, nous lançons la méthode depuis un std::thread. Ainsi la méthode s'exécutera en parallèle.

```
listenerThread = std::thread(&TCPServer::acceptClients, this);
```

Figure 5 : Lancement du thread qui accepte les clients

Les arguments sont : l'adresse mémoire de la méthode, l'adresse de la classe qui contient le thread.

Nous avons un autre problème, c'est que le thread doit vérifier si le serveur ne s'est pas éteint, mais la fonction bloquante empêche la vérification continue. Alors j'ai ajouté un timeout à cette méthode, pour que la boucle while() vérifie si le serveur est toujours actif. Si non, le thread pourra s'arrêter.

```

while (isServerOn) {
    fd_set readfds;
    FD_ZERO(&readfds);
    FD_SET(idSocket, &readfds);
    timeval timeout;
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;

    if (select(idSocket + 1, &readfds, nullptr, nullptr, &timeout) > 0)
    {
        addr_len = sizeof(adr_client);
        SOCKET clientSocket = accept(idSocket, (struct sockaddr*)&adr_client, &addr_len);
    }
}

```

Figure 6 : Timeout pour pouvoir arrêter le thread

le bool isServerOn peut passer à false via d'autres méthodes si par exemple on demande la fermeture du serveur manuellement, ou si un problème majeur est détecté.

Lorsqu'un client est détecté par select(), nous l'acceptons. S'il n'y a pas de problèmes, on vient l'ajouter dans la liste des clients (on ne l'utilise pas, mais c'était mis en place au cas où). Puis on vient l'ajouter dans le std::queue.

Cet ajout se fait en utilisant un mutex. Le mutex permet la protection d'une variable/attribut lors de l'utilisation de plusieurs thread pour éviter que plusieurs modifient en même temps. std::lock_guard débloque l'attribut la durée de vie de ce mutex (ici, en dehors du if).

```

addr_len = sizeof(adr_client);
SOCKET clientSocket = accept(idSocket, (struct sockaddr*)&adr_client, &addr_len);

if (clientSocket != INVALID_SOCKET) {
    std::cout << "a Client is connected : " << clientSocket << std::endl;
    idsClients.push_back(clientSocket);
    std::lock_guard<std::mutex> lock(matchmakingMutex);
    matchmakingQueue.push(clientSocket);
    cvMatchmaking.notify_one();
}

```

Figure 7 : Acceptation d'un client

cvMatchmaking est un attribut de type std::condition_variable (*j'ai oublié de le mettre dans le diagramme de classe*). On va notifier qu'il y a un nouveau joueur à ce attribut qui est en "wait()" dans le thread matchmaking que nous allons traiter juste après.

3/Fonctionnement du matchmaking

Le matchmaking est donc aussi dans un thread car c'est une boucle infinie (jusqu'à la fermeture du serveur).

```
while (isServerOn) {
    std::unique_lock<std::mutex> lock(matchmakingMutex);
    cvMatchmaking.wait(lock, [this] { return matchmakingQueue.size() >= 2 || !isServerOn; });
}
```

Figure 8 : Acceptation d'un client

le cvMatchmaking.wait() vient attendre un appelle "notify_one()" pour exécuter ce qu'il y a dans les {} . Dedans, on va sortir du wait s'il y a 2 joueurs dans la file d'attente ou que le serveur est désactivé (on sortira de la boucle directement si c'est désactivé). Ensuite nous allons gérer l'appairage entre ces 2 clients.

```
SOCKET client1 = matchmakingQueue.front();
matchmakingQueue.pop();
if (!isClientConnected(client1))
    continue;

SOCKET client2 = matchmakingQueue.front();
matchmakingQueue.pop();
if (!isClientConnected(client2)) {
    matchmakingQueue.push(client1);
    continue;
}
```

Figure 9 : Récupération des sockets des 2 clients & test si alive

On vient récupérer les sockets des 2 premiers clients de la file d'attente. Pour gérer le cas où un des clients a quitté le jeu pendant le matchmaking sans appuyer sur le bouton en jeu "quitter matchmaking". Dans la méthode isClientConnected() nous envoyez un message "matchmaking ok" et le client doit y répondre "OK". La méthode retourne si le client lui a répondu ou pas. Si le client 1 ne répond pas, on revient au début de la boucle.

Pour le 2e client il y a une subtilité, on fait la même vérification, mais s'il n'est plus présent, nous devons remettre le client1 dans la file d'attente.

```
std::cout << "Matchmaking made between : " << client1 << " and " << client2 << std::endl;
sessionManager.createSession(client1, client2);
gameThreads[client1] = std::thread(&TCPserver::gameSession, this, client1, client2, false);
gameThreads[client2] = std::thread(&TCPserver::gameSession, this, client2, client1, true);
```

Figure 10 : Démarrage des threads gameSession

Sinon, si les 2 joueurs sont bien actifs. On vient créer une session entre 2 clients dans le sessionManager qui permettra de gérer si un des clients quitte le jeu via l'utilisation d'un booléen partagé. Puis on va lancer pour chacun un thread qui leur permettra de communiquer avec l'autre client.

Le choix de 2 threads n'est pas le plus efficace, mais le plus facile à mettre en place.
Pour résumer le fonctionnement, voici un diagramme de séquence :

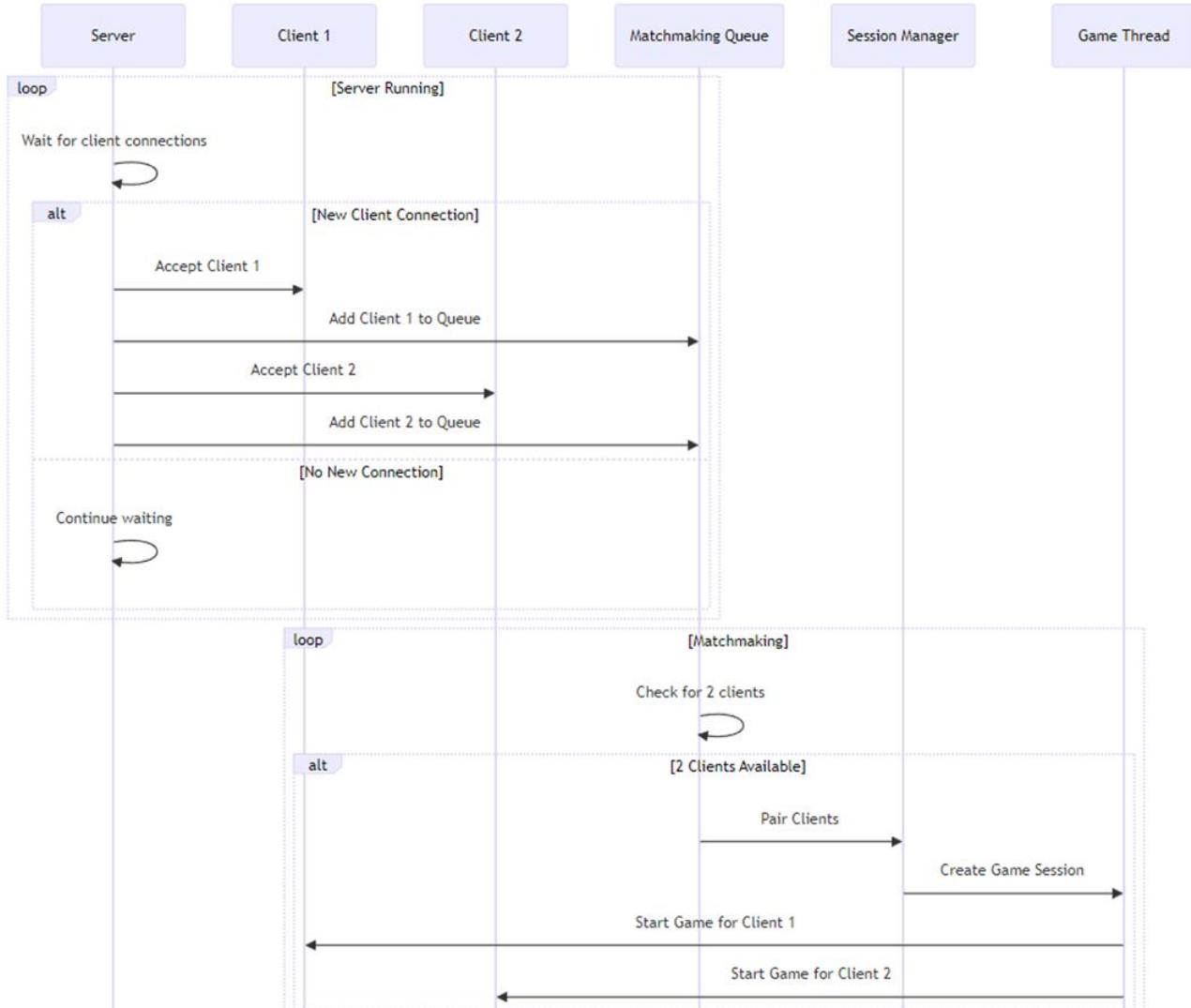


Figure 10 : Diagramme de séquences reprenant les étapes du matchmaking

4/Code pour les messages

```
switch (message.at(0))
{
    case 'G':
        return MessageType::Game;
    case 'B':
        return MessageType::BattleGrid;
    case 'M':
        return MessageType::Chat;
    case 'F':
        return MessageType::End;
    case 'P':
        return MessageType::PlayerInfo;
    default :
        return MessageType::Unknown;
}
```

Figure 11: Traitement du type de message

Pour gérer correctement la réception des messages, on ajoute un indicateur devant le message (emplacement 0) qui permet d'être traité par les clients.

5/Fonctionnement d'une session de jeux

```
void BattleshipServer::gameSession(SOCKET client1, SOCKET client2, bool isFirstPlayerToPlay)
```

Figure 12 : Méthode gameSession de BattleshipServer

Nous utilisons pour le jeu une classe fille à TCPServer pour spécialiser le gameSession pour notre jeu. Le thread visera donc cette méthode.

```
//first message
std::string message = "";
isFirstPlayerToPlay ? message = "GStart" : message = "GWait";
std::cout << "Sent to " << client2 << " " << message << std::endl;
sendMessage(client2, message);
```

Figure 13 : Envoie d'un message au client pour savoir s'il commence ou non

Avant d'entrer dans la boucle du jeu, on informe les clients de qui commence. C'est simplement déterminé au lancement du thread selon si c'est le client1 (il commence) ou client2 (il attend).

```

//GameLoop
try {

    while (isServerOn && sessionState->isActive()) {
        std::string messageFromClient1 = receiveMessageFromClient(client1);

        // if client quit
        if (messageFromClient1.empty()) {
            std::cout << "Someone leaves" << std::endl;
            sessionState->disconnectClient(isFirstPlayerToPlay);
            sendMessage(client2, "Fin");
            break;
        }

        std::cout << "Message received from " << client1 << " send to " << client2;
        std::string response = processGameMessage(messageFromClient1);
        send(client2, response.c_str(), response.length(), 0);
        if (messageFromClient1 == "SLEAVE")
            break;
    }
}

catch (const std::exception& e) {
    std::cerr << "Exception dans gameSession: " << e.what() << std::endl;
}

```

Figure 14 : Gestion de boucle de jeu

Tant que le serveur est actif et qu'un des joueurs n'a pas quitté, on va récupérer les messages envoyés par le joueur pour l'envoyer à son adversaire. Si jamais la chaîne est vide, cela veut dire que le client a quitté. Dans ce cas on indique à la session des 2 joueurs que le joueur a quitté et on l'informe vient de quitter brusquement le jeu. On sort aussi de la boucle si à la fin du jeu, le joueur décide d'appuyer sur le bouton "retour au menu". En effet, les clients ne sont pas déconnectés automatiquement à la fin de la partie pour qu'ils puissent discuter via le chat de discussion. Lorsque le client quitte via le bouton, il envoie "SLEAVE".

La loop est dans un try catch qui permet de récupérer les erreurs s'il y en a car chacune de mes méthodes possède un throw().

A la sortie de la loop, on ferme les sockets des clients du côté serveur en attendant un peu avant pour la synchronisation.

```

std::chrono::milliseconds dt1(1000);
std::this_thread::sleep_for(dt1);
closeClientSocket(client1);
closeClientSocket(client2);

```

Figure 15 : Gestion de boucle de jeu

Résumé du fonctionnement :

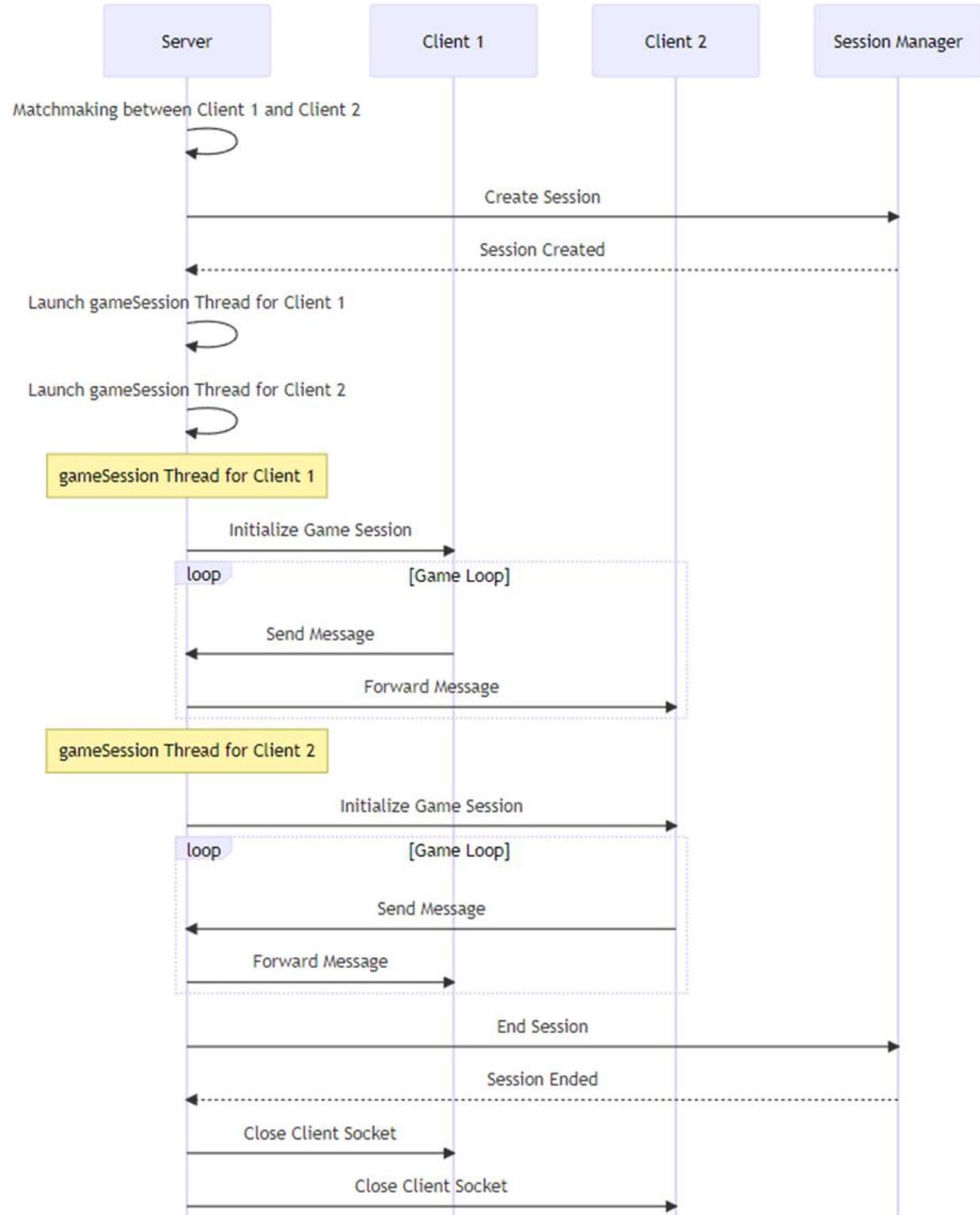


Figure 16 : Diagramme de la gestion de la boucle de jeu

C/Création base graphique

La partie graphique représente plus des deux tiers du temps passé sur le projet. Nous allons y passer très rapidement en indiquant quelques points techniques intéressants.

1/Eviter le hard coding et les nombres magiques

Afin d'éviter au maximum de laisser des nombres magiques dans le code, nous avons créé un fichier de déclaration qui contient différentes const struct contenant toutes les valeurs que nous devons placer manuellement. Par exemple prenons les paramètres de la grille :

```
struct PlayerHUDSettings {  
  
    const sf::Vector2f hudSize = sf::Vector2f(1600, 213);  
    const std::string hudPath = "ressources/UI/ui_playersPanel.png";  
  
    const int nameSize = 80;  
    const int kdSize = 25;  
    const int scoreSize = 30;  
  
    const int pictureSize = 70;  
    const sf::Vector2f picturePos = sf::Vector2f(25, 25);  
  
    const sf::Color nameColor = sf::Color(255, 255, 255);  
    const sf::Color kdColor = sf::Color(180, 180, 180);  
    const sf::Color scoreColor = sf::Color(200, 200, 200);  
  
    const sf::Vector2f namePos = sf::Vector2f(180, 0);  
    const sf::Vector2f scorePos = sf::Vector2f(180, 120);  
    const sf::Vector2f kdPos = sf::Vector2f(180, 95);  
  
};
```

Figure 17: Valeurs constantes pour l'affichage des infos du joueur

```
#include "Settings.h"  
  
class PlayerHud  
{  
private:  
    PlayerHUDSettings playerHudSettings;
```

Figure 18 : Inclusion d'un struct PlayerHUDSettings venant de Settings.h

```

PlayerHud::PlayerHud(sf::Font& font, bool isEnemy, std::string name, std::string kd, std::string score, sf::Texture& playerTexture)
{
    this->isEnemy = isEnemy;

    if (!isEnemy)
    {
        bgTexture.loadFromFile(playerHudSettings.hudPath);
        background = new EntityRectangle(playerHudSettings.hudSize, sf::Vector2f(0, 0), bgTexture);
    }
    else
    {
        background = nullptr;
    }

    nameTxt = new EntityText(font, playerHudSettings.namePos, playerHudSettings.nameSize, name, playerHudSettings.nameColor);
    kdTxt = new EntityText(font, playerHudSettings.kdPos, playerHudSettings.kdSize, "K/D: " + kd, playerHudSettings.kdColor);
    scoreTxt = new EntityText(font, playerHudSettings.scorePos, playerHudSettings.scoreSize, "Score: " + score, playerHudSettings.scoreColor);
    playerPicture = new EntityCircle(playerHudSettings.pictureSize, playerHudSettings.picturePos, playerTexture);
}

```

Figure 19 : Utilisation du struct dans le constructeur de PlayerHud

Ainsi, les valeurs données ont un sens lors de la lecture du code, et peuvent être rapidement modifiées depuis Settings.h sans aller rechercher la partie du code qui gère les valeurs.

2/Les fenêtres

Pour gérer les fenêtres, nous utilisons du polymorphisme. Pour cela nous avons une classe mère abstraite SFMLWindow qui possède la boucle de l'affichage d'une fenêtre de jeu SFML. Nous dérivons cette classe pour chaque fenêtre dont on a besoin. Nous avons 4 fenêtres : Chargement du jeu, Menu, Magasin, Jeu.

Le diagramme ci-dessous vient montrer le principe pour les 2 fenêtres principales.

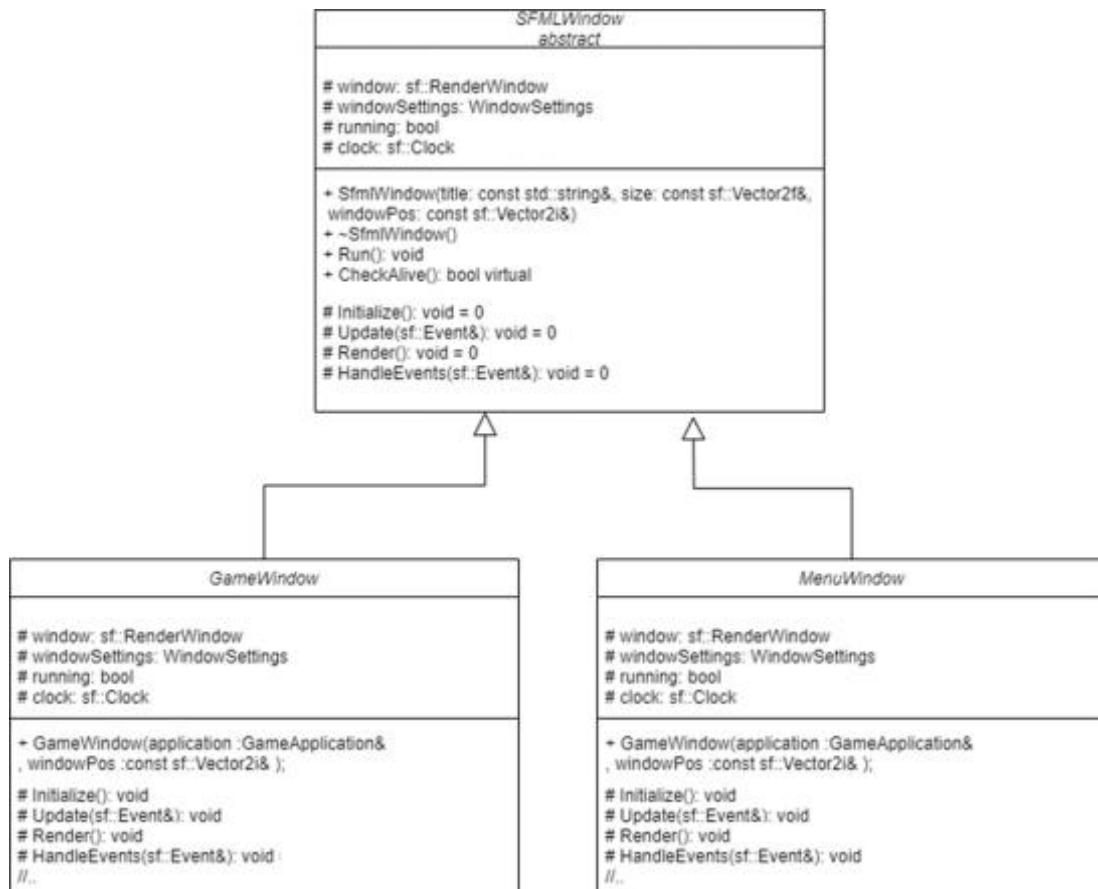


Figure 20 : Diagramme de classe des fenêtres

La gestion des fenêtres est gérée dans la classe “GameApplication” qui vient gérer le changement des fenêtres.

Selon la fenêtre choisie, il va pointer sur cette dernière. Ainsi, dans la boucle qui dessine la fenêtre, nous pouvons faire :

```
private:  
    SfmlWindow* currentWindow;  
Dans le fichier déclaration de GameApplication.h  
  
void GameApplication::Run() {  
  
    while (running)  
    {  
        if (currentWindow != nullptr)  
            currentWindow->Run();  
    }  
}  
Dans le fichier définition de GameApplication.cpp
```

Figure 21 : Utilisation du polymorphisme pour les fenêtres

3/Les éléments graphiques

Maintenant que nous avons des fenêtres, nous devons désormais créer des éléments visuels sur ces dernières. Pour plus de simplicité, j'ai mis en place nos propres classes facilitant l'ajout d'éléments sur nos fenêtres. Ces éléments vont tous dériver de la classe mère Entity afin de pouvoir utiliser le polymorphisme lors de l'affichage.

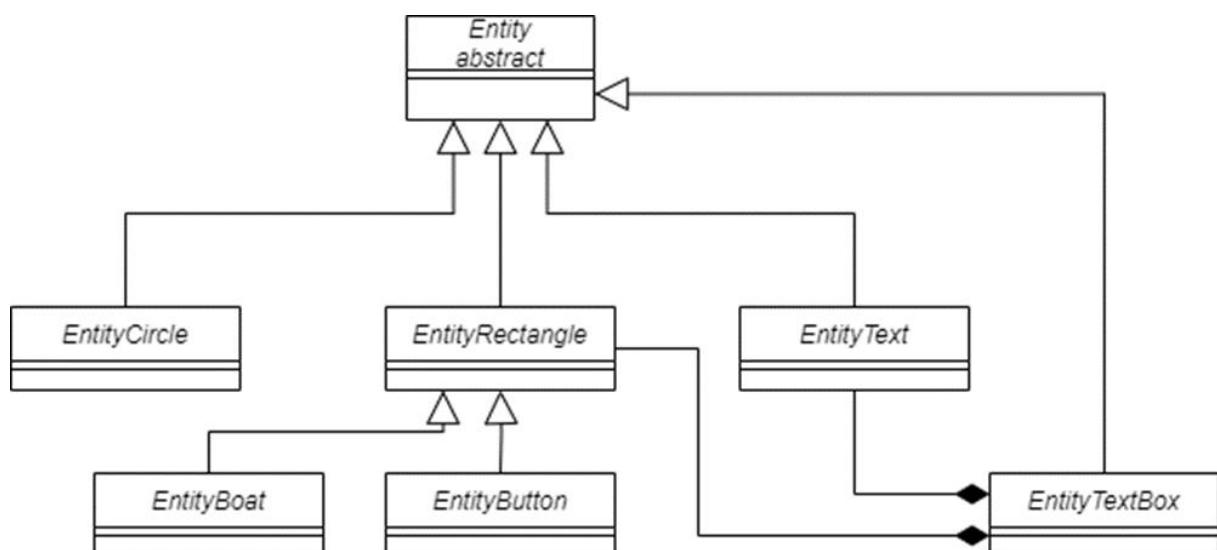


Figure 22 : Diagramme simplifié des classes des différents éléments à afficher

D/Intégration Jeu au graphique

L'intégration au jeu est plutôt simple. Nous devons seulement récupérer les coordonnées du bateau sur la grille en calculant sa position selon l'origine de la grille et la taille de chacun de ses carreaux. En divisant la position et en faisant une conversion implicite en int, on peut récupérer sa position de 0 à 9 sur la grille. (La méthode n'est appelée que lorsque le bateau est sur la grille).

```
sf::Vector2f Boat::AnchoredPosition()
{
    GridSettings grid;
    int gridSize = grid.squareSize / grid.nbPixels;

    int anchoredX = shape.getPosition().x / gridSize;
    int anchoredY = shape.getPosition().y / gridSize;

    return sf::Vector2f(anchoredX, anchoredY);
}
```

Figure 23 : Récupération des coordonnées sur la grille.

Avec ces 2 coordonnées, il est facile d'utiliser le système créé par mon camarade Raphaël pour gérer toute la logique des placements des bateaux, des attaques et détection de bateaux coulés.

E/Ajouts de features graphiques

En utilisant les méthodes de mon camarade, je peux simplement ajouter les éléments graphiques récupérant les données envoyées par ces méthodes. Il est compliqué de montrer le code car à ce stade, nous utilisons de nombreuses classes et méthodes. De ce fait, il faudrait prendre beaucoup de captures.

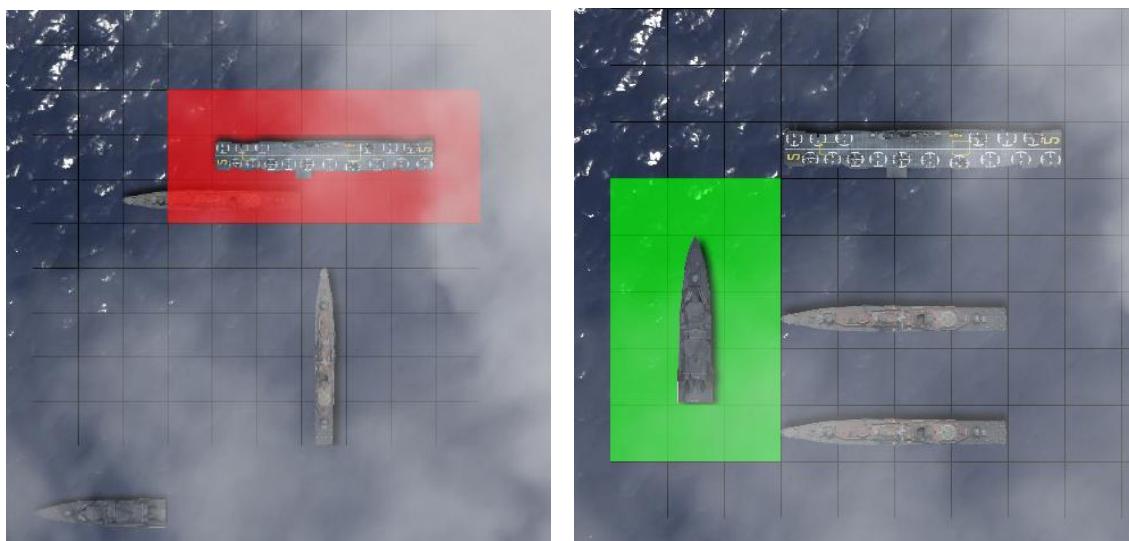


Figure 24 : Placement d'un bateau

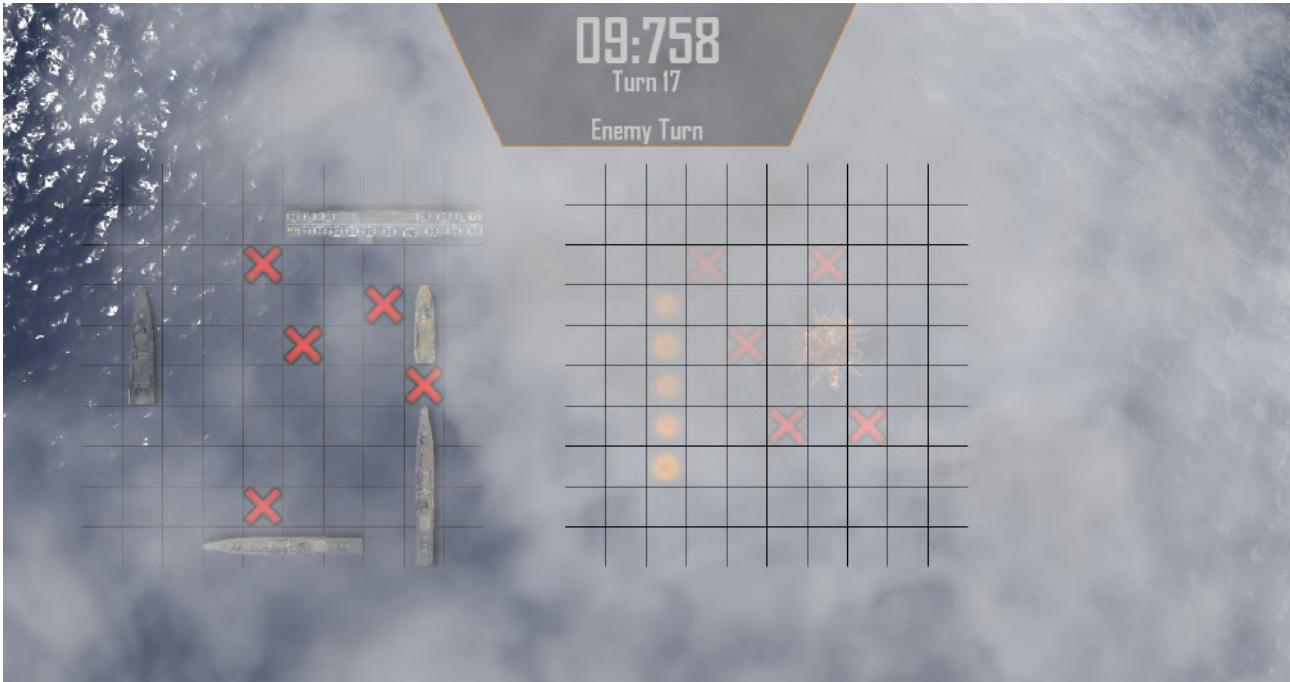


Figure 25 : Placement d'un bateau

F/Données de joueurs

En utilisant la classe de base de données de mon camarade, nous récupérons les données des joueurs et l'envoyons au client adversaire. Nous pouvons simplement afficher les données sur l'interface comme ceci :

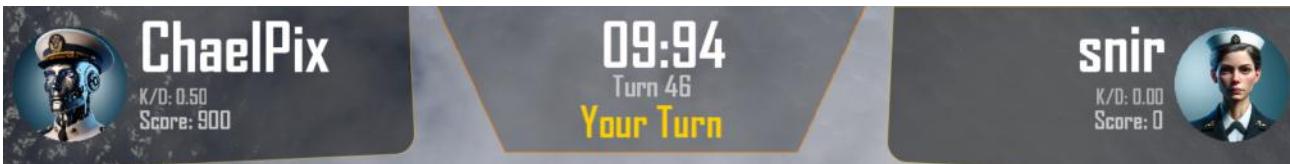


Figure 26 : Affichage des infos des joueurs

H/Chat de discussion

Maintenant que nous avons les données des joueurs, dont notamment leur pseudo. Nous pouvons créer un chat de discussion entre les deux joueurs. Pour que les messages soient différenciés des coordonnées d'attaques, nous ajoutons en première lettre le caractère 'M' afin que la machine state du jeu comprenne que c'est un message et non un signal de changement de tour. Ainsi, nous pouvons avoir 2 clients qui discutent ensemble.

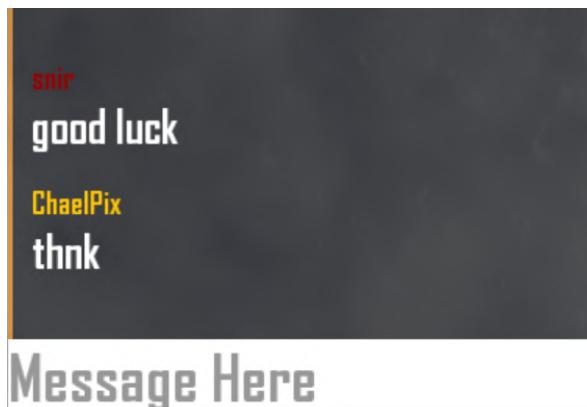


Figure 26 : Placement d'un bateau

I/Partie Physique : Fonctionnement d'un thread au niveau CPU

Afin d'avoir une partie physique en lien avec mes tâches, j'ai choisi d'expliquer le fonctionnement d'un thread logiciel au sein du processeur afin de comprendre comment le système fonctionne de manière hardware.

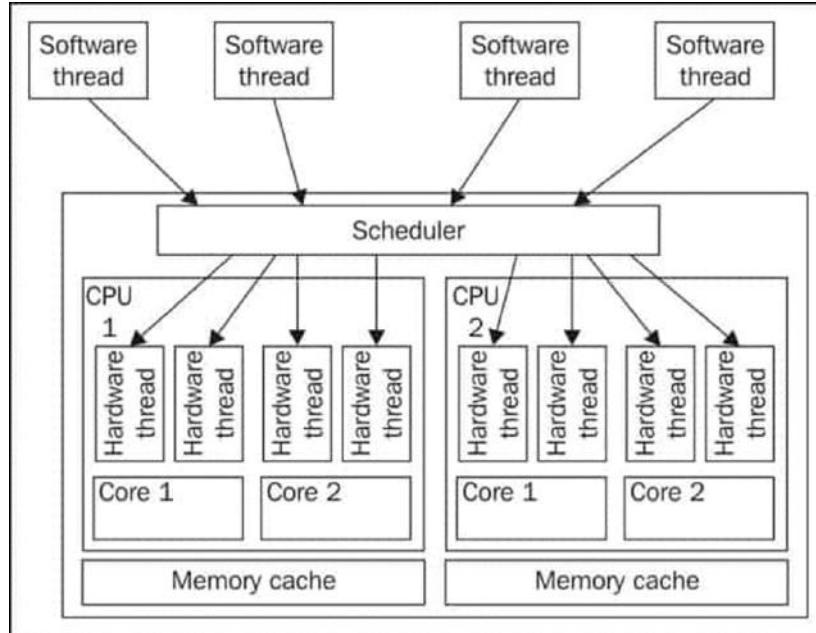


Figure 27 : Illustration fonctionnement des threads dans un processeur

Les threads logiciels en C++ sont gérés par le système d'exploitation et sont exécutés par le processeur par des threads matériels sur des cœurs spécifiques. Un planificateur (Scheduler) alloue un thread logiciel à un thread matériel, ce dernier est placé dans une file d'attente pour être exécuté par le cœur correspondant. Cette opération dépend de la priorité et de l'état actuel du cœur du processeur qui peut être libre en train d'exécuter un autre thread.

Le système d'exploitation utilise des techniques telles que le "time slicing" pour partager le temps processeur entre plusieurs threads, permettant ainsi une exécution quasi simultanée. Cette technique assure que chaque thread reçoit un segment de temps pour exécuter ses instructions sur le cœur du processeur avant de passer au thread suivant dans la file d'attente.

Le cache de mémoire joue également un rôle important dans la performance des threads. Lorsque des données ou des instructions sont récupérées de la mémoire principale, elles sont stockées dans le cache pour un accès plus rapide. Ainsi, lorsque le thread matériel est exécuté, il peut accéder aux données nécessaires avec moins de latence. Il est à noter que les données fréquemment accédées par les threads sont susceptibles d'être conservées dans le cache, ce qui réduit considérablement le temps d'accès par rapport à la mémoire principale.

Chaque cœur possède généralement son propre cache de niveau 1 (L1), tandis que le cache de niveau 2 (L2) peut être partagé entre plusieurs cœurs ou exclusif à un seul. Les processeurs multicœurs peuvent également disposer d'un cache de niveau 3 (L3) partagé entre tous les cœurs, permettant ainsi une gestion efficace des données utilisées par différents threads.

V/Conclusion :

Voici une capture d'écran en jeu du résultat que nous avons. Il y a pleins de détails comme l'affichage du tour de la personne, le nombre de tour, le curseur de souris affichant une case de couleur verte ou rouge selon si l'emplacement est touché ou non, le chrono devant de plus en plus rouge, les flammes de bateaux devenant rouges quand le bateau est coulé, les nuages mobiles, le fond de l'océan animé, les sfx (sound effects) etc. Le rapport pourrait durer des pages et des pages...

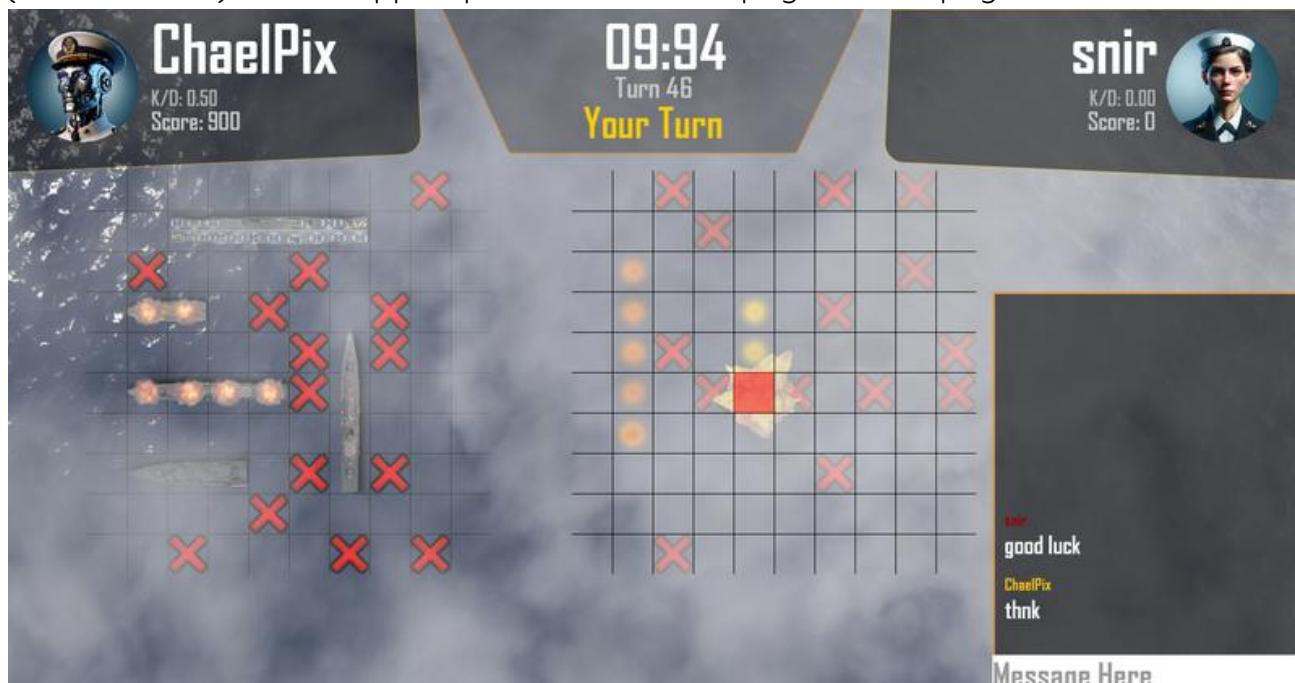


Figure 1 : Capture d'écran du jeu

Nous avons été heureux de pouvoir présenter un jeu complet et fonctionnel. Nous avons beaucoup appris à travers notre mini-projet.
Nous remercions M.Dartois pour l'accès à son VPS (virtual private server) ainsi que tous les professeurs pour avoir répondu à nos questions.