



# Rapport final de Projet Informatique « La Bataille Brestoise »

Alexandre Froehlich & Guillaume Leinen  
alexandre.froehlich@ensta-bretagne.org  
guillaume.leinen@ensta-bretagne.org

## Table des matières

<b>1</b>	<b>Description générale du problème</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Principe de fonctionnement du jeu . . . . .	1
1.3	Les différentes pistes envisagées . . . . .	2
1.4	Différentes hypothèses réductrices choisies . . . . .	2
<b>2</b>	<b>Application proprement dite</b>	<b>2</b>
2.1	Présentation générale du programme . . . . .	2
2.1.1	Le serveur . . . . .	2
2.1.2	Le client . . . . .	3
2.2	Description des principales classes/méthodes . . . . .	5
2.2.1	Classes de jeu . . . . .	5
2.2.2	Classe de l'IHM . . . . .	6
2.2.3	Classe du client/serveur . . . . .	7
<b>3</b>	<b>Conclusion</b>	<b>7</b>
3.1	Figures imposées . . . . .	7
3.2	Différents tests effectués . . . . .	9
3.3	Ajouts après la 1ère partie du projet . . . . .	9
3.4	Limitations . . . . .	11
3.5	Perspectives d'amélioration . . . . .	11
<b>A</b>	<b>Diagrammes de classe</b>	<b>13</b>

# 1 Description générale du problème

## 1.1 Introduction

Pour ce projet d'informatique, nous avons décidé de prendre un sujet autre que ceux proposés par nos enseignants. Ayant fait tous les deux une année zéro dans la marine nationale, un sujet proche de ce que l'on avait vécu semblait une solution évidente.

Nous sommes donc partis sur un jeu de type bataille navale, en temps réel et jouable en multijoueur que nous avons nommé :



FIGURE 1 – Logo du projet d'informatique

Pour travailler en collaboratif sur le code, nous avons utilisé un repository sur github, le code est accessible au lien suivant :

<https://github.com/NightlySide/La-Bataille-Brestoise>.

## 1.2 Principe de fonctionnement du jeu

Le jeu se base sur le fonctionnement du célèbre jeu [agar.io](https://agar.io). Le joueur commence la partie avec le navire de plus bas niveau (un bâtiment-école de niveau 1). Le joueur va devoir trouver puis tirer sur les ennemis afin d'engranger de l'expérience en endommageant des navires ennemis ou en les coulant (il obtient un bonus proportionnel au niveau(tier) du navire coulé).

Après un certain palier d'expérience il monte de niveau, afin de manoeuvrer un navire de plus en plus performant. Les niveaux vont de 1 à 5. Après 50000 exp le joueur est déclaré vainqueur. Le joueur possède un nombre de points de vie, et il peut mourir si ce nombre tombe à 0. Dans ce cas, le joueur réapparaît dans un bâtiment du niveau inférieur, et son exp est réinitialisé au palier inférieur. Le joueur peut notamment consulter son expérience actuelle via une commande dans une chatbox.

Le jeu permet donc plusieurs stratégies de victoire : s'attaquer à des ennemis plus faibles mais mettre du temps à gagner de l'expérience (farming), ou viser des ennemis plus imposants sous le risque de mourir. Les commandes sont détaillées dans le fichier *README.md*.

### 1.3 Les différentes pistes envisagées

Pour le choix de l'interface Homme-Machine (IHM) le choix s'est rapidement porté sur **PyQt5** pour sa polyvalence et sa programmation événementielle. Concernant la connexion entre le serveur et les clients, nous sommes tout d'abord partis sur la bibliothèque **sockets** qui permet de créer un tunnel simple entre deux clients. Nous avons finalement choisi **asyncio**, implémentée depuis python3, qui permet de gérer en asynchrone toutes les requêtes nécessaires au projet, et ce avec plusieurs clients en simultané.

### 1.4 Différentes hypothèses réductrices choisies

Pour réduire la taille du projet et surtout le rendre réalisable, nous avons choisi d'établir quelques réductions pour cette première partie :

- Le positionnement et le déplacement des joueurs et entités se fait sur une grille de  $8 \times 8$  pixels.
- Les joueurs ne sont pas visibles par les autres joueurs.
- Le chat ne peut accueillir que du texte, donc pas de commande (du type */help*).
- La vitesse de rafraichissement, ou le nombre de boucles de jeu par seconde est indépendant selon la classe.

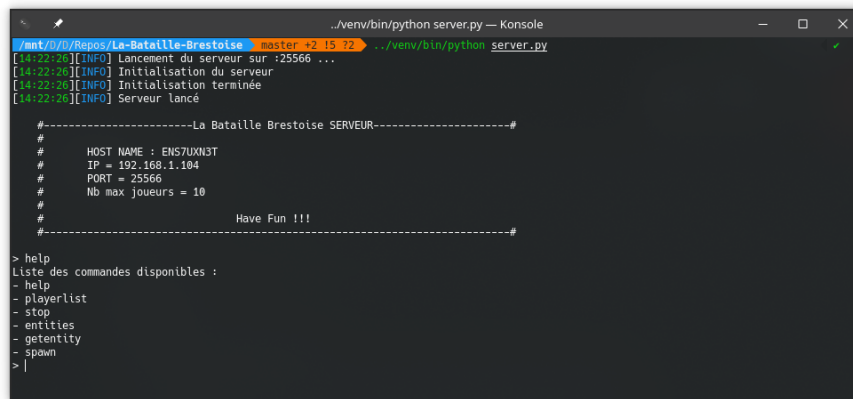
## 2 Application proprement dite

### 2.1 Présentation générale du programme

Le programme est scindé en deux parties : le client et le serveur avec chacun son script pour le lancer. Le script *test\_suite.py* permet de lancer tous les tests unitaires contenus dans le dossier *tests*.

#### 2.1.1 Le serveur

Lancer le serveur avec le script *serveur.py*, crée un serveur sur l'hôte *localhost* (127.0.0.1 ou 0.0.0.0) sur le port 25566. Le serveur va ensuite accueillir les nouveaux joueurs, organiser le chat et gérer toutes les entités du jeu. Une fois lancé et opérationnel le terminal de commande ressemble à cela :



```
..venv/bin/python server.py — Konsole
[14:22:40][INFO] Lancement du serveur sur 25566 ...
[14:22:40][INFO] Initialisation du serveur
[14:22:40][INFO] Initialisation terminée
[14:22:40][INFO] Serveur lancé

#-----La Bataille Brestoise SERVEUR-----#
#
# HOST NAME : ENS7UXN3T
# IP = 192.168.1.104
# PORT = 25566
# Nb max joueurs = 10
#
# Have Fun !!!
#-----#

> help
Liste des commandes disponibles :
- help
- playlist
- stop
- entities
- getentity
- spawn
> |
```

FIGURE 2 – Ecran de fonctionnement du serveur

Une fois démarré le serveur affichera les informations utiles telles que le nom de la machine sur laquelle il est exécuté, son IP locale, le port de connexion et le nombre de joueurs maximum (pas encore implémenté).

### 2.1.2 Le client

Une fois le serveur lancé en local, on peut soit se connecter à ce dernier, soit se connecter à un serveur que nous laisserons ouvert chez nous, pré-enregistré dans la liste des serveurs disponibles.

En démarrant le client avec le script *client.py*, on se trouve devant l'écran de connexion du projet. (cf. 3)

Sur celui-ci l'utilisateur pourra retrouver les serveurs pré-enregistrés ainsi qu'une boîte pour indiquer son pseudonyme, un bouton pour se connecter au serveur sélectionné.

Enfin l'utilisateur pourra appuyer sur le bouton "Serveur local" pour entrer manuellement les informations de connexion au serveur (cf. 4)



FIGURE 3 – Ecran de connexion client

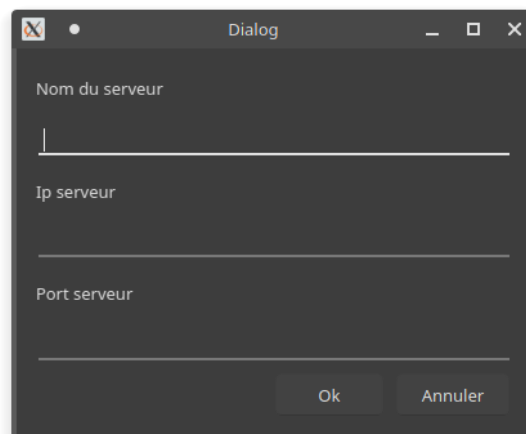


FIGURE 4 – Boite de dialogue de connexion manuelle à un serveur

Une fois connecté au serveur, l'utilisateur est face à l'écran de jeu (cf. 5) qui comporte une surface de jeu, que nous appellerons *canvas*, sur laquelle sera dessinée la carte, les entités et les joueurs. On y retrouve un radar permettant de détecter les entités aux alentours (attention pas entièrement fonctionnel pour l'instant), et une chatbox pour communiquer avec les autres joueurs présents sur le serveur.

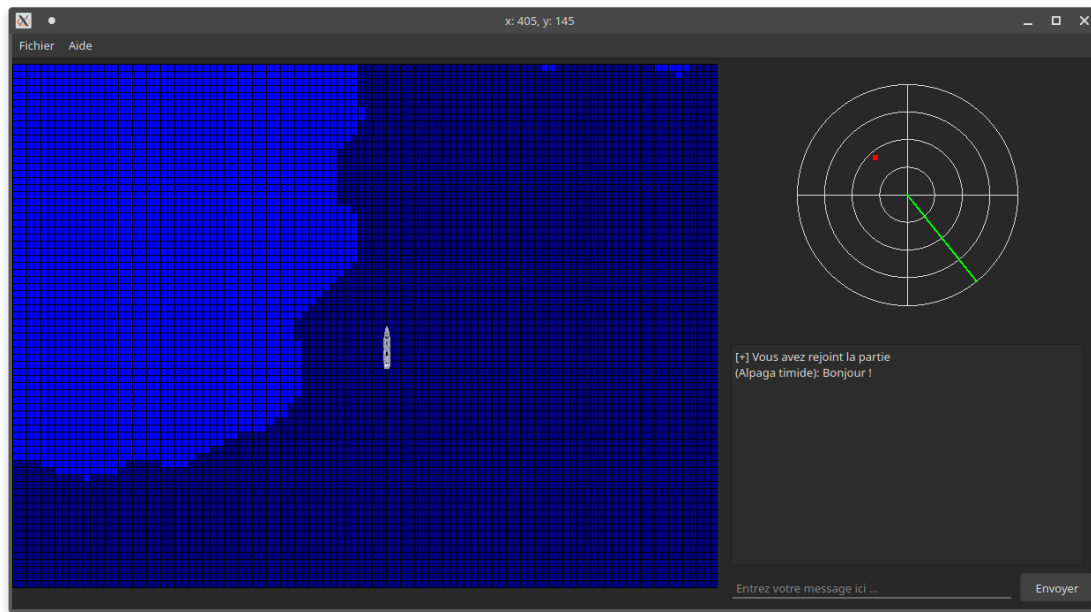


FIGURE 5 – Ecran de jeu du projet

Pour cette première phase, nous nous sommes concentrés sur la structure du jeu. Par conséquent les contrôles joueurs sont limités. Pour se déplacer l'utilisateur pourra utiliser les flèches du clavier dès que le focus est sur le Canvas (un click souris sur le Canvas suffit à récupérer le focus).

## 2.2 Description des principales classes/méthodes

Pour une meilleure lisibilité, vous pouvez retrouver les diagrammes de classes en annexe (cf. 13)

### 2.2.1 Classes de jeu

En premier lieu, il convient de décrire les classes directement liées au fonctionnement du jeu. (cf. 15)

La classe "Entité" est la base d'un joueur, qu'il soit réel ou contrôlé par la machine. Elle va notamment avoir des attributs de points de vie, de position sur la carte ou encore d'arme actuellement sélectionné par le joueur.

Une entité peut être "joueur" ou "IA". Une entité interagit avec les autres entités par le biais du client de jeu, les entité "IA" sont contrôlées par le serveur. Une entité

à un instant  $t$  contrôle un bâtiment et équipe une arme disponible sur ce bâtiment. La classe "Bâtiment" contient toutes les statistiques et données sur un bâtiment, cela peut être son niveau, les points de vie maximaux, ou encore la portée de détection. La classe "Armes" quant à elle définit les données des différentes armes disponibles. (cf. 16)

### 2.2.2 Classe de l'IHM

Dans l'IHM, nous avons intégré plusieurs éléments qui nécessitent leur propre classe.

La classe "Radar" est une classe héritée de `QWidget`. Elle gère le radar de détection des autres entités ennemies.

La règle du jeu veut que si un ennemi se trouve dans notre portée de détection et que cet ennemi a une dissimulation (implémentée plus tard) supérieure à la distance qui nous sépare d'elle, cet ennemi sera affiché par un point rouge sur le radar.

Le radar balaye l'espace à  $90^\circ/s$ . On garde les traces d'un ennemi détecté en rouge puis en gris lorsqu'elles s'apprêtent à disparaître du radar.

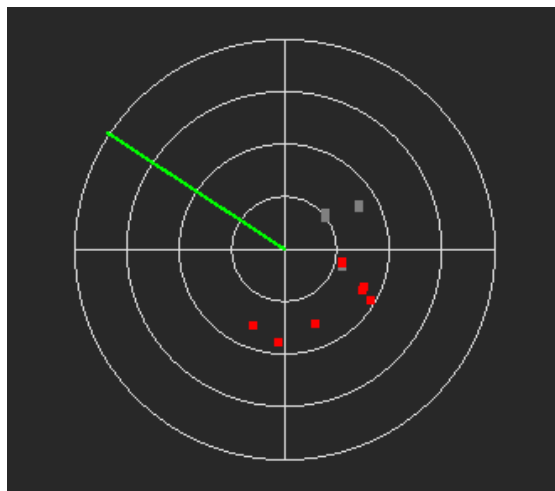


FIGURE 6 – Aperçu du Radar

La classe "Chatbox" comme son nom l'indique renvoie à la chat box en bas à droite du canevas de jeu. Elle est héritée de `QTextBrowser`. Cette classe existe car on doit pouvoir écrire sur le chat pendant que le jeu tourne, il est donc nécessaire de créer un autre thread, sur lequel tourne la chatbox, et dans lequel Qt va piocher les informations lorsque nécessaire.

Elle contient une liste de lignes qui est mise à jour avec les messages des autres joueurs. Un signal Qt permet d'envoyer un message en appuyant sur la touche Entrée.



### 2.2.3 Classe du client/serveur

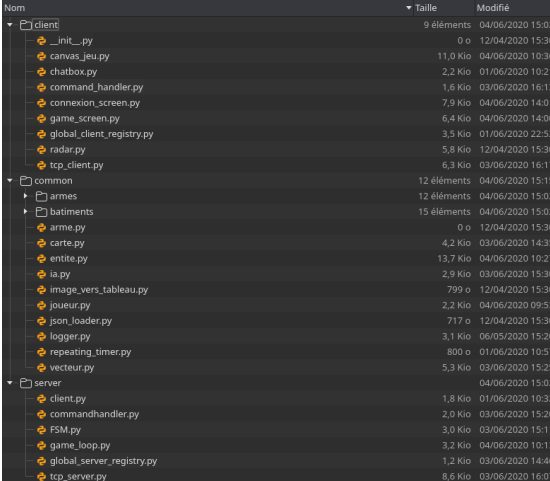
Pour gérer la connexion asynchrone entre le client et le serveur il est nécessaire d'avoir deux classes supplémentaires gérant ces différents aspects. La classe 'TCPClientProtocol' (cf. 15) hérite du protocole de la bibliothèque *asyncio* et qui fonctionne en évènementiel. La classe 'TCPServer' du côté serveur (cf. 14), possède un fonctionnement similaire. On retrouve ainsi dans les deux classes une fonction pour prendre en charge l'ouverture du tunnel entre le client et le serveur, une fonction pour prendre en charge la déconnexion et enfin deux fonctions pour l'envoi et la réception de données.

## 3 Conclusion

### 3.1 Figures imposées

Pour ce projet, 3 figures étaient imposées. On a alors pu choisir parmi les figures restantes pour en respecter au minimum 7. Voici les figures imposées ainsi que l'endroit de leur implémentation dans ce projet :

**Factorisation du code** : Le projet est relativement conséquent. Garder l'intégralité du code dans un unique fichier est impossible car trop difficile à maintenir. Ce projet est alors divisé en 3 grandes parties dans le dossier *lib*: *client* (TCP et IHM), *serveur* (TCP et mécaniques de jeu) et *common* (Commun au client et au serveur).



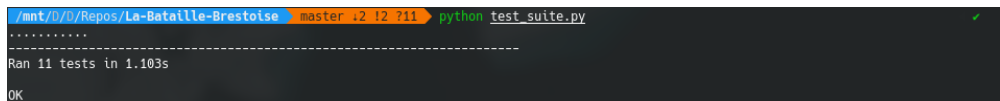
Nom	Taille	Modifié
client	9 éléments	04/06/2020 15:02
__init__.py	0 o	12/04/2020 15:30
canvas_jeu.py	11,0 Kio	04/06/2020 10:36
chatbox.py	2,2 Kio	01/06/2020 10:21
command_handler.py	1,6 Kio	03/06/2020 16:13
connexion_screen.py	7,9 Kio	04/06/2020 14:01
game_screen.py	6,4 Kio	04/06/2020 14:00
global_client_registry.py	3,5 Kio	01/06/2020 22:52
radar.py	5,8 Kio	12/04/2020 15:30
tcp_client.py	6,3 Kio	03/06/2020 16:17
common	12 éléments	04/06/2020 15:15
armes	12 éléments	04/06/2020 15:02
batiments	15 éléments	04/06/2020 15:02
__init__.py	0 o	12/04/2020 15:30
arme.py	4,2 Kio	03/06/2020 14:35
carte.py	13,7 Kio	04/06/2020 10:27
entite.py	2,9 Kio	03/06/2020 15:30
ia.py	799 o	12/04/2020 15:30
image_vers_tableau.py	2,2 Kio	04/06/2020 09:53
joueur.py	717 o	12/04/2020 15:30
json_loader.py	3,1 Kio	06/05/2020 15:20
logger.py	800 o	01/06/2020 10:57
repeating_timer.py	5,3 Kio	03/06/2020 15:25
vecteur.py		04/06/2020 15:02
server		04/06/2020 15:02
client.py	1,8 Kio	01/06/2020 10:32
commandhandler.py	2,0 Kio	03/06/2020 15:20
FSM.py	3,0 Kio	03/06/2020 15:11
game_loop.py	3,2 Kio	04/06/2020 10:13
global_server_registry.py	1,2 Kio	03/06/2020 14:46
tcp_server.py	8,6 Kio	03/06/2020 16:07

FIGURE 7 – Structure des fichiers du projet.

**Documentation et commentaires du code** : On a décidé de commenter le code à l'aide des docstrings et en suivant la nomenclature Google ([https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example\\_google.html](https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html)) qui nous correspondait le mieux en termes de lisibilité et de place prise. La documentation

est à la fois présente en local (compilé avec Sphinx) dans le dossier `/doc/build` ou bien en ligne sur ReadTheDocs à l'adresse suivante : <https://la-bataille-brestoise.readthedocs.io/fr/latest/>.

**Tests unitaires** : les tests unitaires sont utiles pour vérifier tout au long de la conception du projet que le code de « base » est toujours opérationnel. Pour se faire nous avons utilisé la bibliothèque *unittest*. Pour lancer la batterie de tests unitaires (comportant des tests sur les entités, sur l'IHM et sur la classe personnalisée vecteur), il suffit de lancer le fichier *test\_suite.py*.



```

/mnt/D:/Repos/La-Bataille-Brestoise master i2 i2 ?11 > python test_suite.py
.....
Ran 11 tests in 1.103s
OK

```

FIGURE 8 – Exemple de lancement du fichier *test\_suite.py*.

**Création d'un type objet (classe)** : afin de rendre le code exploitable et surtout de pouvoir le maintenir, nous avons créé des classes représentant divers objets. Que ce soit les entités ou bien leurs classes héritées, que ce soit les classe contenant l'IHM, ce projet utilise bien la Programmation Orientée Objet.

**Héritage au moins entre deux types créés** : Nous implémentons cette figure avec notamment les classes *Batiment* (`lib.common.batiments.batiment`) et *Armes* (`lib.common.armes.arme`) qui sont les classes de base des navires et des armes du jeu. Grâce à ces classes on peut spécifier les différents types de navires et d'armes de notre jeu en conservant un nombre faible de fonctions grâce au polymorphisme.

**Héritage depuis un type intégré (hors IHM)** : La carte du jeu (`lib.common.carte`), chargée à partir d'une image en noir et blanc, est directement héritée d'un array numpy afin de pouvoir profiter des fonctions avancées de slicing pour optimiser l'exécution du programme.

**Structure de données dynamique (autre que celles intégrées à Python)** : La carte répond à cette figure, car il s'agit d'une structure de données qui varie en fonction de son utilisation dans client ou serveur.

**Lecture/ écriture de fichiers** : dans ce projet, on charge des images, on charge du son et de la musique dans le `canvas_jeu` (`lib.client.canvas_jeu`) mais aussi on charge les paramètres du serveur stocké dans un fichier de type JSON (Javascript Object) *serveur\_config.json* et les serveurs connus sont enregistrés sont de même dans le fichier *known\_servers.json*. Ces deux derniers fichiers sont chargés à l'aide de la classe *JsonLoader* (`lib.common.json_loader`).

Au total ce sont près de 8 figures que nous respectons dans ce projet informatique. *NB: Devant le nombre conséquent de classes et de portions de code implémentant les figures imposées, nous avons fait le choix de ne pas expliciter de façon exhaustive ces dernières. Nous avons cependant cité quelques exemples.*

### 3.2 Différents tests effectués

Pour les tests unitaires, le script *test\_suite.py* permet d'effectuer les tests provenant des classes de *unittest* présentes dans le dossier 'tests'. Ces classes de test vont tester les éléments de l'IHM ainsi que certaines mécaniques du jeu.

Pour la connexion client/serveur, nous avons tout d'abord essayé la bibliothèque *sockets* afin de créer la connexion entre le client et le serveur. Cependant cette dernière ne gère que difficilement les connexions multiples du côté du serveur. Il est vrai que l'on aurait pu utiliser comme avec python 2.7 le module *selectors*, cependant cette méthode, comme *asyncore* que nous avons aussi essayé, semble disparaître avec les versions après 3.6 au profit de *asyncio*.

Pour suivre l'évolution des tests et à fins de debug, nous avons implémenté un *Logger* dans notre projet. Il va gérer l'affichage sur le terminal de commande en fonction de la sévérité du message envoyé par le programme, il peut de plus, enregistrer ces logs dans un fichier *.log*.

### 3.3 Ajouts après la 1ère partie du projet

Un des ajouts principaux consiste en l'implémentation d'une IA, dont les comportements sont décrits dans la classe *FSM*, pour Final State Machine. l'IA peut ainsi "Aggresser", comprendre suivre, un joueur "réel" (*Joueur*), tirer à intervalles réguliers ou fuir devant un ennemi.

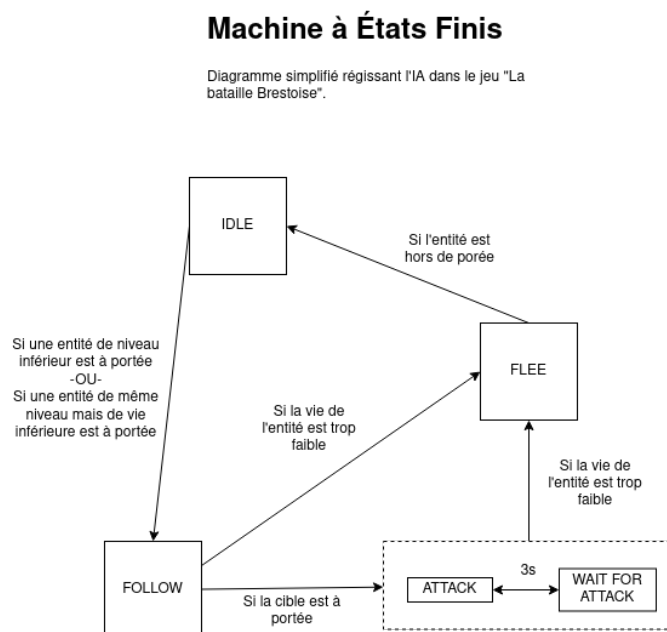


FIGURE 9 – Diagramme de comportement de l'IA (FSM)

En plus de cela, nous avons d'une manière plus générale relié la partie client et

serveur du jeu (l'affichage et les requêtes) à l'aspect "mécaniques de jeu", notamment l'objet *entite*. Il est maintenant possible pour les joueurs et les IA d'interagir. De plus les différentes statistiques des navires et des armes (points de vie, dégâts par secondes, vitesse, ...) ont fait l'objet d'un équilibrage afin de ne pas rendre un navire trop ou trop peu performant par rapport aux autres.

arme	DPS		tps equipement (s)	portée (km)	
calibre50	1000		1	1	
20mm	2000		3	2	
76mm	4250		4	6	
100mm	4000		5	8	
mistral	4500		5	5	
exocet	5000		8	15	
torpille légère	5000		10	20	
torpille lourde	6000		10	30	
rafale	6000		10	30	

bâtiments	PV	portée détection (km)	dissimulation (km)	vitesse (km/s)	niveau
BE	10000	2	1,75	1	1
BIN	7500	2	1,5	1	1
AVISO	15000	3	3	1	2
CMT	10000	4	2,25	0,9	2
BH	15000	5	3	1,25	2
FS	20000	10	4	1,15	3
F70	30000	8	5	1,1	3
FREMM	30000	30	8	1,3	4
FDA	40000	23	10	1,35	4
SNA	15000	15	3	1,1	4
PA	50000	23	15	1,4	5
SNLE	30000	18	5	1,2	5

FIGURE 10 – Statistiques des navires, la dissimulation ,la portée de détection ou encore l'équipement ne sont pas encore implémentés

Concernant l'aspect interface, nous avons ajouté un certain nombre d'éléments graphique comme des icônes dynamiques pour les bâtiments, une barre de vie coloré en vert pour le joueur, en rouge pour un joueur ennemi, ou encore le tier du bâtiment actuel à gauche de la barre de vie.

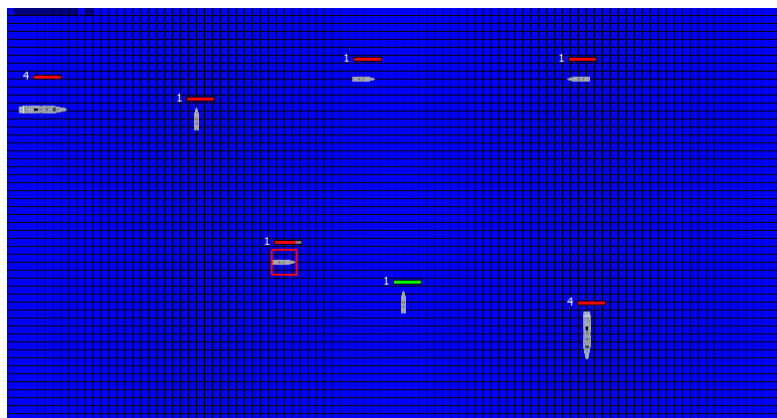


FIGURE 11 – Nouveaux éléments graphiques implémentés

Nous avons également implémenté un objet *QMediaPlayer* afin de jouer pendant la partie un son d'ambiance et un son de tir. D'ailleurs nous avons ajouté la mécanique de tir (lié à la pression de la barre espace).

### 3.4 Limitations

Nous avons rencontré quelques limitations durant ce projet.

En premier lieu, l'usage d'une mode multijoueur implique une utilisation extensive de la bande passante, ce qui nécessite d'optimiser les requêtes afin de ne pas surcharger un réseau.

Ensuite le fait que Pickle ne gère pas directement QT, en raison du fait qu'il ne gère pas les objets codés en C. Or, nos messages sont sérialisés à l'aide de *pickle* et envoyés sur le réseau. Il faut donc se restreindre à de l'envoi d'objets de 'bases' reconnus par pickle lors de la communication client/serveur.

Une dernière limitation est le fait que la bibliothèque *pyQt*, n'accepte pas que l'on interagisse avec ses objets à partir d'un thread différent. La solution fut de passer par des classes annexes comme "ChatBox" afin de contourner cette limitation, en ayant d'un côté le thread TCP qui peuple le contenu de la ChatBox, et de l'autre PyQt qui récupère ces informations pour l'afficher sur l'IHM (comme expliqué précédemment et cf. 12)

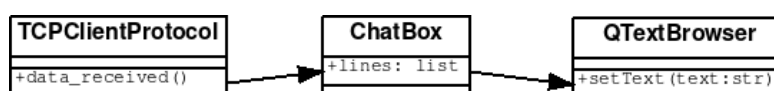


FIGURE 12 – Principe de la classe intermédiaire entre deux Threads

A l'issue de la 2ème partie, nous avons pu contourner la plupart de ces limitations, notamment au niveau des requêtes serveurs (notamment à l'aide d'un *Command\_Handler*) et les seuls réels limitations de notre projet sont actuellement plus de l'ordre de la performance que du fonctionnement.

### 3.5 Perspectives d'amélioration

On pourrait améliorer la dynamique des bâtiments, en leur donnant des caractéristiques d'inertie ou encore de rayon de giration, ce qui aurait été plus réaliste (on ne manoeuvre pas un porte-avions comme un BIN).

On pourrait encore implémenter un concept de dégât critique, avec des équipements importants du bâtiment qui pourrait être mis temporairement hors d'usage après un coup bien placé (armement, radar, propulsion).

Pour améliorer le temps de calcul on pourra par exemple réfléchir à un moyen de passer en calcul vectoriel si possible.

Au niveau des fonctionnalités au coeur du projet, il était prévu de pouvoir dynamiquement changer d'arme équipé en partie à l'aide d'une roue d'équipement (en utilisant un objet de type *QtPieChart* par exemple). Nous avons été pris par le temps et nous n'avons pas pu réaliser le widget, cependant la structure de la classe *batiment* et certaines méthodes comme la méthode *equiper()* d'un objet *Entite* sont

déjà implémentés, ce qui permettra au final de facilement ajouter la fonctionnalité

Enfin on pourra implémenter des sauvegarde de score ou tout simplement des sauvegardes de paramètres utilisateurs dans des fichiers internes au projet, probablement en JSON ou en YAML.

## A Diagrammes de classe

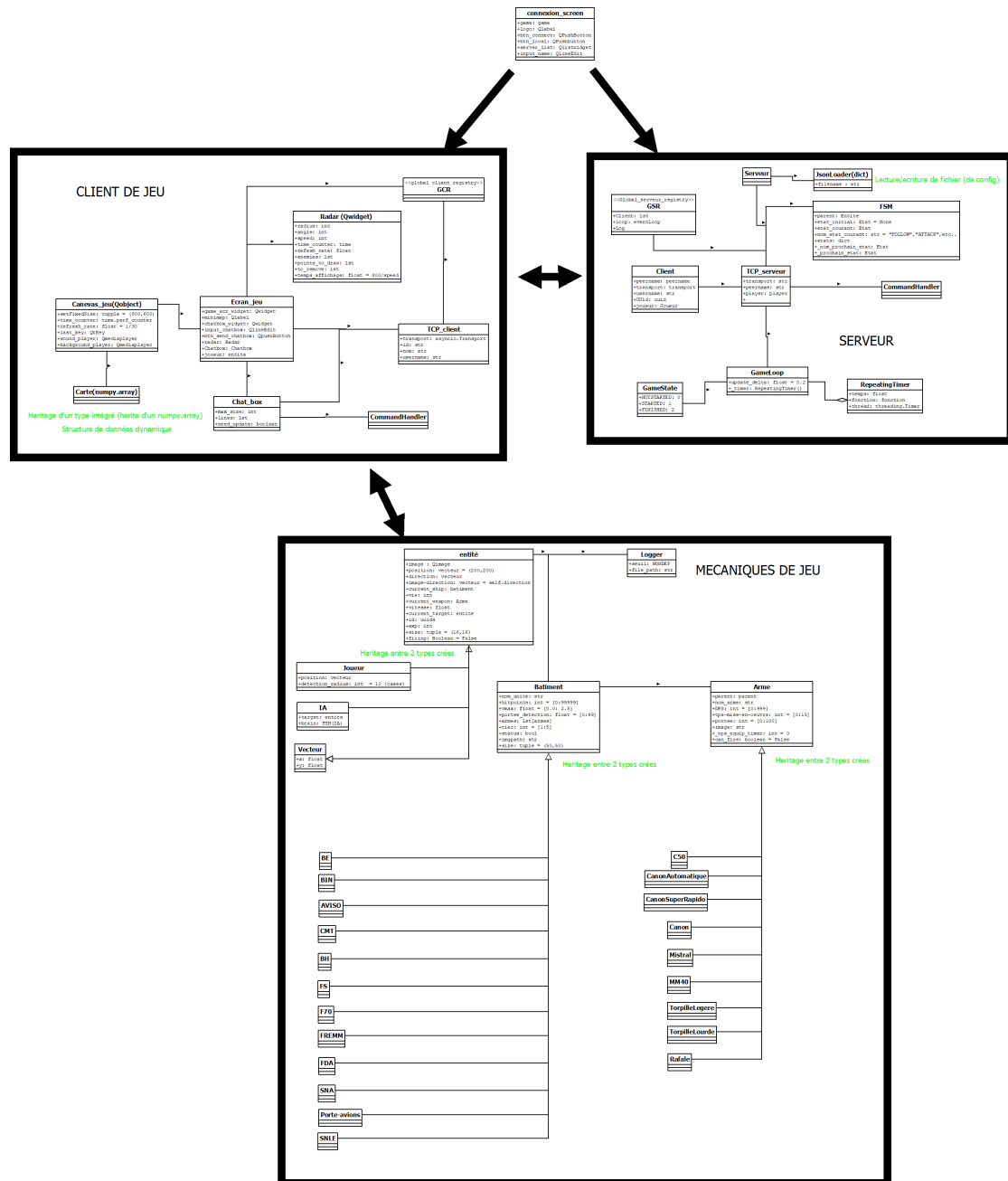


FIGURE 13 – Diagramme de classes complet

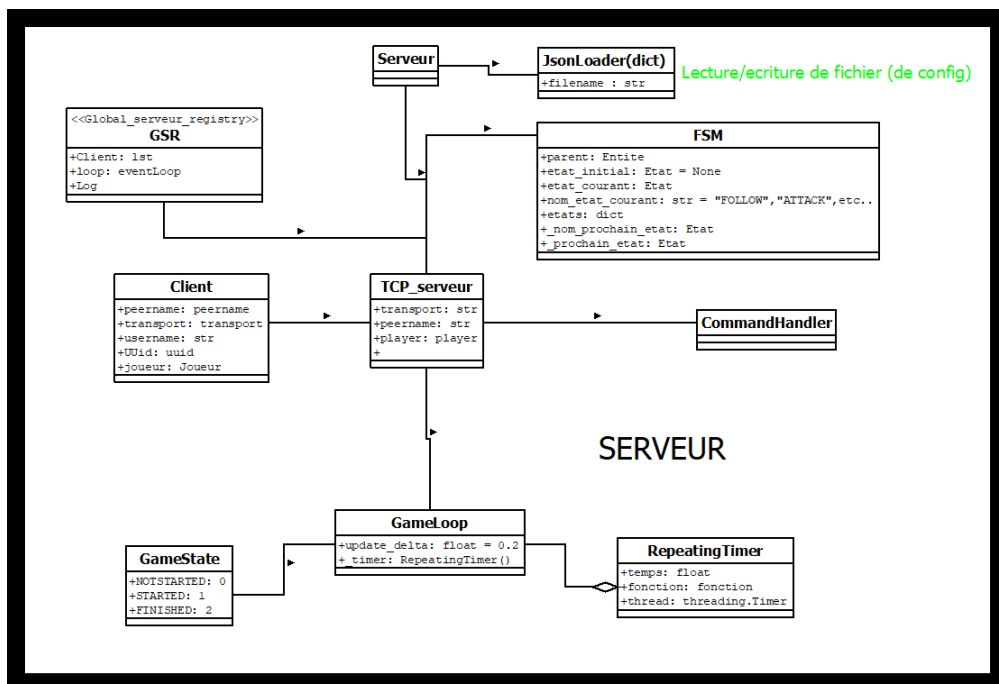


FIGURE 14 – Diagramme de classes côté serveur

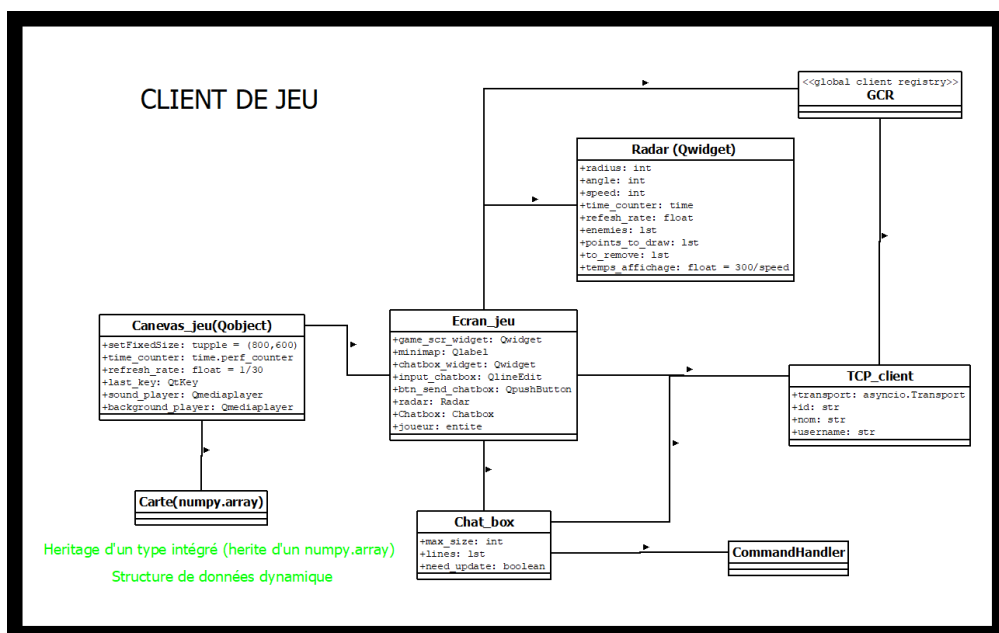


FIGURE 15 – Diagramme de classes côté client



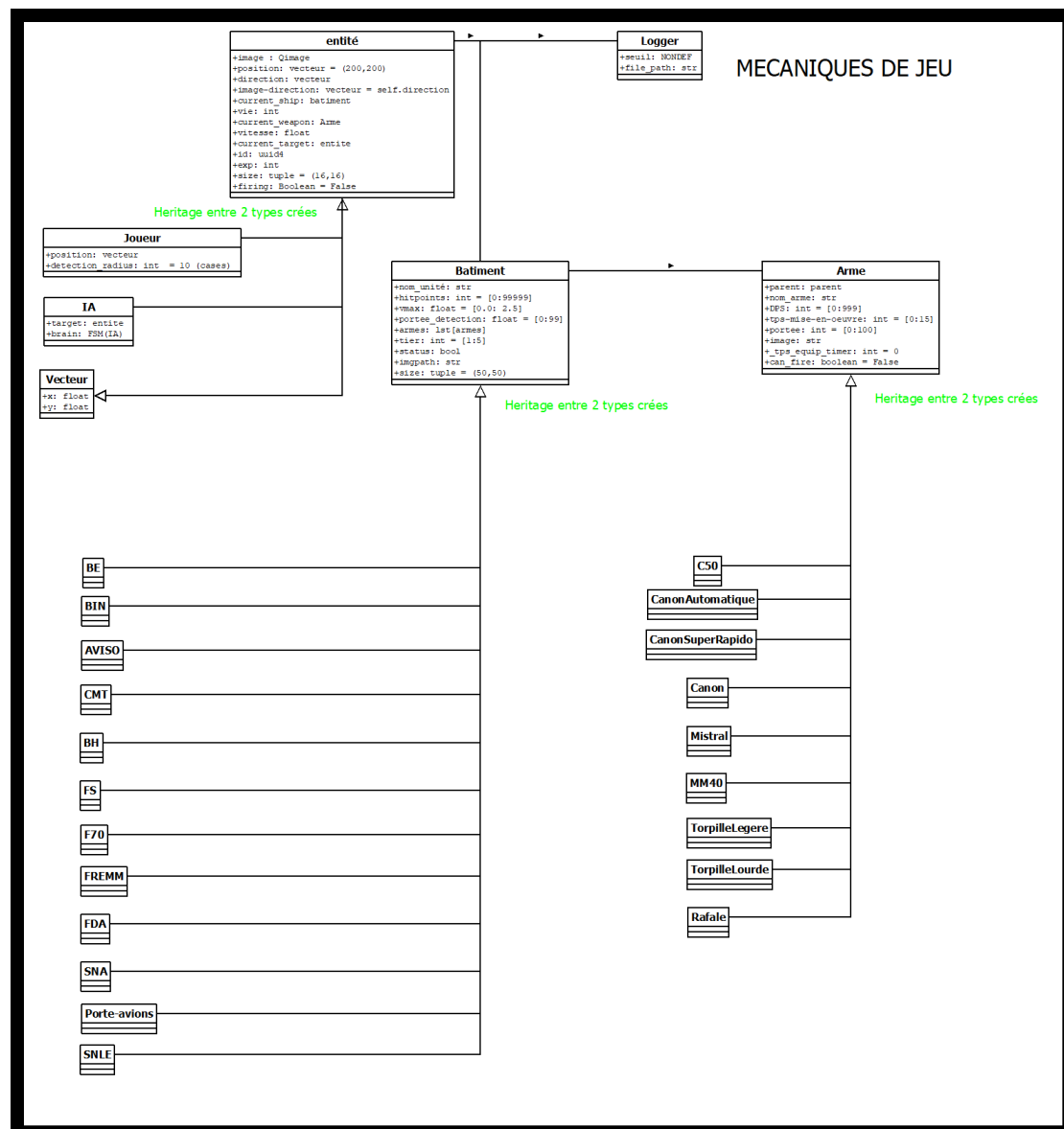


FIGURE 16 – Diagramme de classes des mécaniques de jeu