IUT de Valence – Université Grenoble Alpes

# COMPTE RENDU

SAÉ 6.1

BOURGUIGNEAU Ethan, CHAVANEL Yohann, GRAILLE Théo,
SAPET Alan
16/05/2024

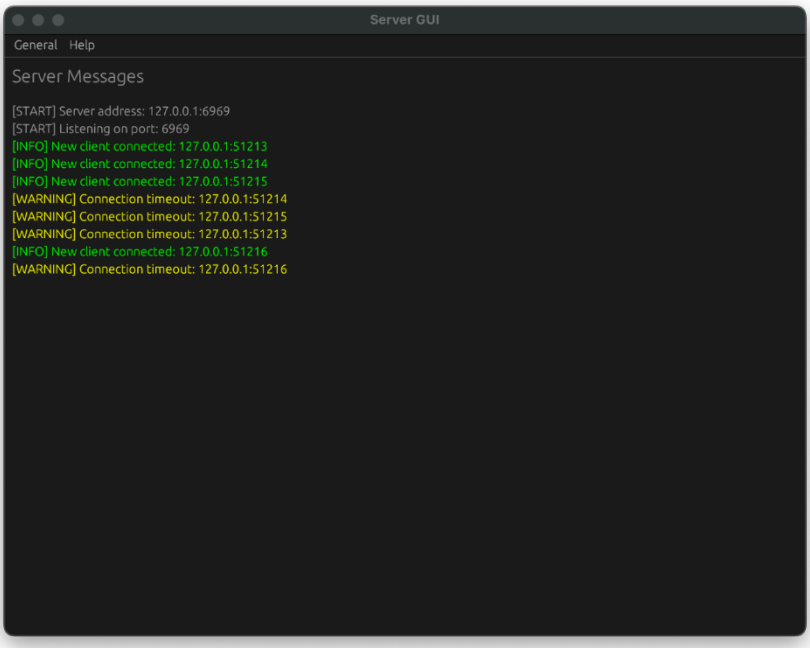
# 1. Fonctionnalités réalisées

## 1.1. Serveur Thread (Ethan) :

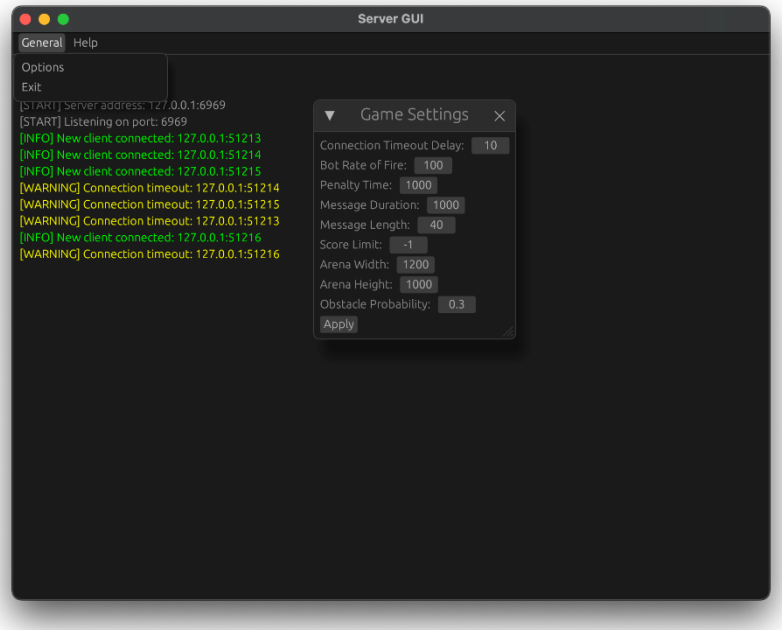
Dans cette fonctionnalité, j'ai dû faire en sorte d'instancier et de gérer la communication avec le serveur. C'est ici que je vais définir les différents paramètres du serveur à l'aide des variables globales de l'application contenue dans "App_defines". C'est aussi ici que je vais définir la structure du serveur, il en possède deux "ServerSettings" contenant tous les paramètres du serveur, et "ServerThread" qui va contenir le port, les messages et les settings. C'est aussi ici que je vais instancier l'implémentation c'est à dire avec la création d'un server et la fonction "start(&self)" et qui va permettre d'instancier le serveur sur une adresse et un port spécifier lors de la création du "ServerThread". Ensuite, le serveur va écouter en attendant qu'un client se connecte sur le serveur et si c'est le cas alors il va faire un "Arc::clone" pour les messages et les settings du serveur ce qui me sera très utile pour passer toutes ces informations à l'IU pour par exemple modifier les paramètres par défaut ou afficher les informations globales du serveur comme les connexions, les timeouts, les erreurs et les messages d'informations... Et c'est après tout cela qu'on instancie dans un nouveau thread, un "ClientHandler".

## 1.2. Serveur UI (Ethan) :

Cette fonctionnalité, va permettre d'afficher tous les logs sur serveur que ce soit lors de la création du serveur ou l'on peut voir sur quelle port et adresse écoute le serveur. Mais aussi, toutes les informations sur les clients, avec la connexion d'un nouveau client, mais aussi si un client n'envoi plus de message au serveur depuis un certain temps, il est donc shutdown/kick du serveur. Le serveur prend en charge la déconnexion d'un client si le client à envoyer le message "EXIT" qui est contenu dans "App_defines".



Cette image monte un exemple des logs que peuvent ressortir le serveur. On peut aussi voir les différents onglets avec général qui va permettre d'arrêter le serveur mais aussi, de changer les settings. De plus dans help il y a les mentions légales fictifs pour permettre d'avoir un code qui pourrait par la suite être publier et mis en production sans problème. Dans l'onglet général on va aussi pouvoir changer les paramètres du serveur grâce au Arc::clone qui permet de passer les informations du serveur à l'IU et inversement.




## 1.3. Client Handler (Ethan) :

Le client handler est vraiment très important car c'est lui qui va gérer toute la logique de connexion d'un client au serveur, mais aussi la gestion des messages envoyer au serveur et la réception des données renvoyer par le serveur. Pour la structure de clientHandler nous avons le socket (TcpStream), le buffer writer, le buffer reader permettant d'écrire et lire les messages au serveur. Il y a aussi le previous_time permettant de savoir si un utilisateur doit être timeout, les messages et settings permettant la gestion des informations entre toutes les interfaces et le serveur. En ce qui concerne l'implémentation du clientHandler nous avons la création d'un client, la fonction "run" très important car c'est lui qui va gérer tout le cycle de vie d'un client, c'est à dire la gestion du timeout, mais aussi des messages en les splittant au bon format pour pouvoir ensuite les traiter via la fonction "process_message", permettant la gestion des déconnexions, les commandes des joueurs et des informations que le serveur va pouvoir renvoyer au client. Toutes les commandes avec ce que peux faire un client sont contenu dans App_Defines.

Pour les tests le serveur va exceptionnellement print toutes les commandes qu'envoient les clients. On peut donc par exemple imaginer que le client envoi "NAME=Player1#COL=255=0=0#LIVE#MSG=Hello,server!#CBOT=0.2=143.5#CPROJ=0.2=143.5#NBOT=Player1" Et voici les print formatter et des messages reçus par le serveur pour les tests :

```
Processing message: "NAME=Player1"
Values: ["NAME", "Player1"]
Commande values: "NAME"


Processing message: "COL=255=0=0"
Values: ["COL", "255", "0", "0"]
Commande values: "COL"


Processing message: "LIVE"
Values: ["LIVE"]
Commande values: "LIVE"


Processing message: "MSG=Hello,server!"
Values: ["MSG", "Hello,server!"]
Commande values: "MSG"


Processing message: "CBOT=0.2=143.5"
Values: ["CBOT", "0.2", "143.5"]
Commande values: "CBOT"


Processing message: "CPROJ= 0.2=143.5"
Values: ["CPROJ", " 0.2", "143.5"]
Commande values: "CPROJ"


Processing message: "NBOT=Player1"
Values: ["NBOT", "Player1"]
```

On peut voir que les messages sont bien split et bien récupérer.

### 1.4. App Defines (Ethan) :

Pour configurer tout le serveur j'ai créé un fichier qui s'appelle App_define et qui contient toutes les variables globales par défaut du projet, cela va de la configuration par défaut du serveur, aux variables globales et aux codes que peut recevoir le serveur avec le split et les séparateurs de commandes.

### 1.5. Types (Ethan) :

J'ai créé un fichier rassemblent les différentes structures et implémentation globale du projet comme par exemple, le formatage des messages avec la configuration du style et de la gestion des différents types de messages à l'aide d'un enum pour définir si le message qui va être afficher (Info, Error, Warning, Debug, Default). Cependant, ce fichier pourra par la

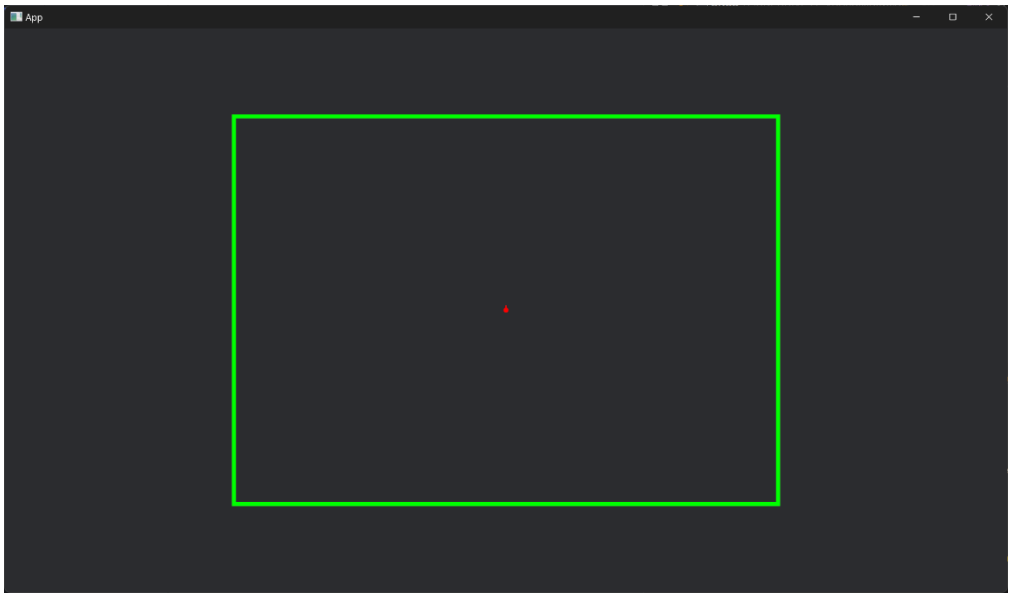
suite contenir d'autres méthodes et implémentation. Pour résumer, on pourrait comparer ce fichier à un fichier *utiles* contenant des méthodes globales réutilisables partout dans le code.

## 1.6. Game UI (Yohann) :

En ce qui concerne la partie visuelle du serveur (affichage du jeu en cours, des joueurs, de la carte, etc.), j'ai exploré diverses bibliothèques graphiques pour notre jeu. Pour cela, j'ai réalisé plusieurs prototypes en utilisant trois bibliothèques différentes : Bevy, Raylib et Egui.

### 1.6.1 Bevy

Pour ce qui concerne la première bibliothèque, Bevy, celle-ci n'est pas seulement une bibliothèque graphique mais également un moteur de jeu et un moteur physique. Ainsi, au lieu d'utiliser deux bibliothèques distinctes, une pour le moteur physique et une autre pour l'affichage graphique des calculs du moteur physique, Bevy permet de gérer les deux aspects simultanément. Cependant, j'ai trouvé la création de l'interface utilisateur (UI) plus complexe qu'avec Raylib ou Egui. Par exemple, organiser notre fenêtre avec un tableau de classement à gauche et l'affichage de la carte interactive à droite s'est avéré très difficile. En particulier, je n'ai pas réussi à coder les interactions avec la carte, comme le zoom, le dézoom et le déplacement de l'affichage (caméra), tandis qu'avec d'autres bibliothèques, ces fonctionnalités étaient plus faciles à implémenter.

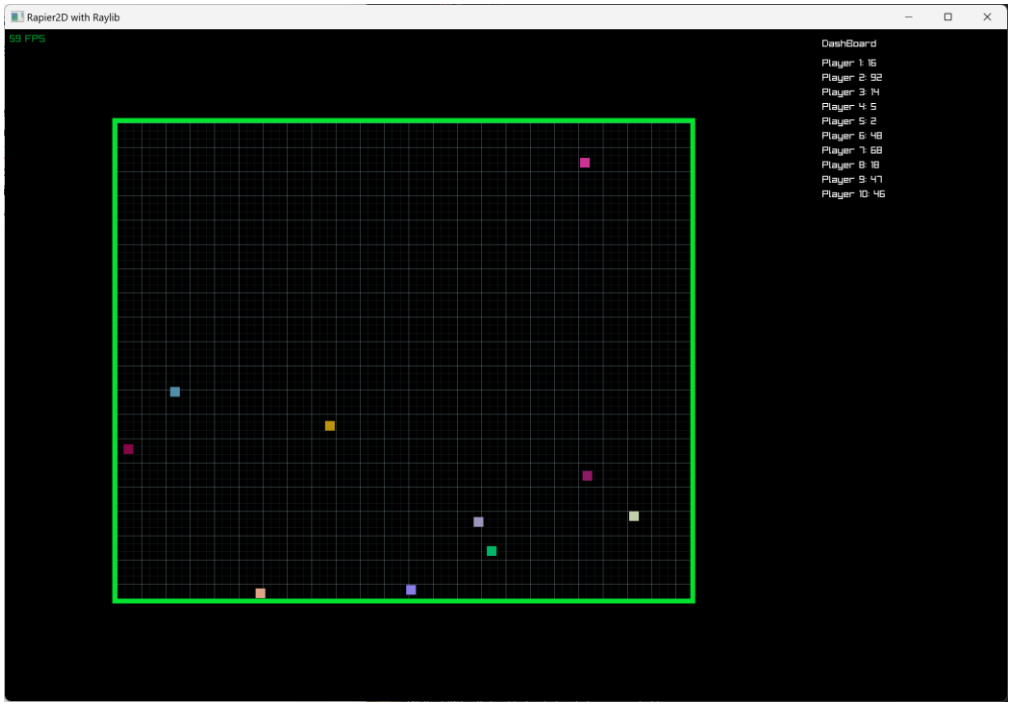



Exemple d'interface réalisée en Bevy avec en vert les bordures du monde et en rouge un joueur.

### 1.6.2 Raylib & Rapier2D

Une autre bibliothèque que j'ai explorée est RayLib. Contrairement à Bevy, RayLib se concentre uniquement sur l'affichage graphique et ne gère pas les calculs physiques. Pour cela, j'ai utilisé la bibliothèque Rapier2D. RayLib permet de réaliser des « dessins » sur la fenêtre, ce qui est extrêmement pratique pour représenter la carte et les joueurs. Comme le dessin est vectoriel, nous pouvons zoomer et dézoomer sans perte de qualité grâce à une gestion des caméras facilitée. Cependant, cette approche présente des contraintes. Par exemple, pour le tableau de bord, il serait plus pertinent d'utiliser un composant tableau pour afficher les scores des joueurs, plutôt que de le dessiner. Il en va de même pour les boutons d'interaction avec le jeu. RayLib ne dispose pas de composants boutons intégrés, il faut donc redessiner chaque bouton et gérer les collisions avec la souris manuellement.

Un autre inconvénient majeur de cette bibliothèque est qu'elle a été initialement conçue pour le langage de programmation C. La quasi-totalité des exemples et de la documentation sont en C et non en Rust. Bien que la documentation pour la version Rust de Raylib existe, elle reste très sommaire.




Exemple d'interface réalisée avec Raylib. En vert les bordures du monde et les petits carrés colorés sont les joueurs.

### 1.6.3 Egui & Rapier2D

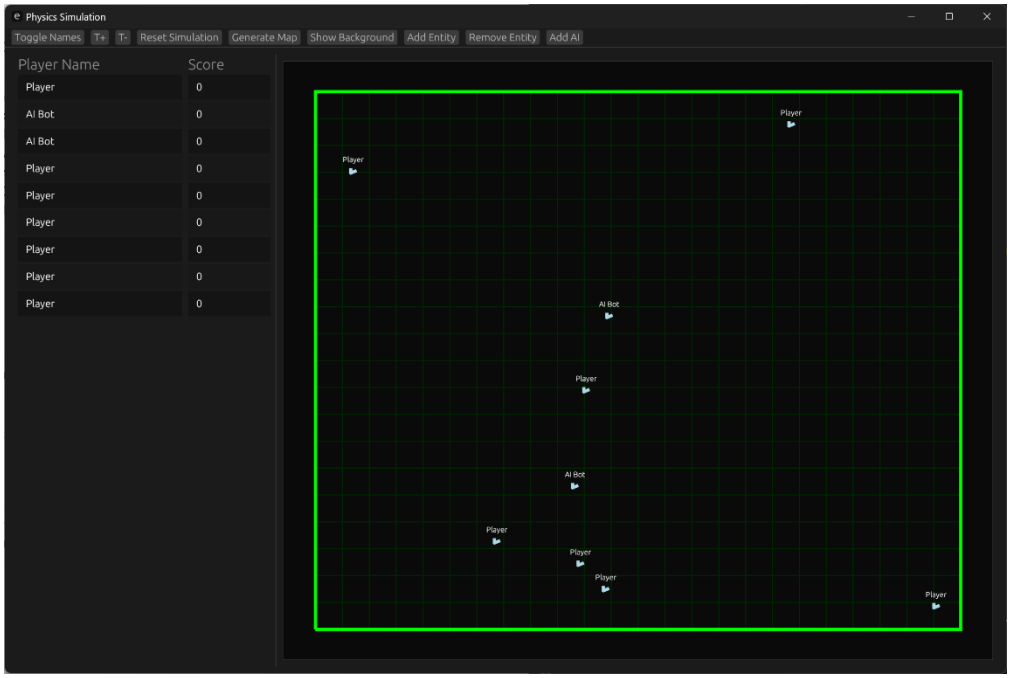
La troisième et dernière bibliothèque que j'ai explorée est Egui. Tout comme Raylib, cette bibliothèque ne gère pas l'aspect physique du jeu. Pour cela, j'ai utilisé Rapier2D lors de mes tests. Cependant, Egui se distingue en permettant la création de véritables applications et interfaces utilisateurs complètes. Contrairement à Raylib, Egui propose des composants intégrés tels que des boutons, des tableaux, et bien d'autres.

La création d'une interface utilisateur complexe a été grandement facilitée par Egui. J'ai pu, non sans difficultés, créer un panneau latéral pour afficher le tableau de bord, ainsi que des boutons en haut de la fenêtre pour interagir avec le jeu. Egui simplifie l'ajout de ces éléments grâce à ses composants prêts à l'emploi, ce qui réduit considérablement le temps de développement par rapport à Raylib.

Cependant, Egui ne dispose pas de fonctionnalités de dessin vectoriel comme Raylib. Pour contourner cette limitation, j'ai utilisé Egui-Plot, une extension d'Egui. Egui-Plot permet de créer des graphiques interactifs similaires à ceux que l'on peut réaliser avec GeoGebra. Sur une grille, il est possible de dessiner des traits, des points, des cercles, etc. Egui-Plot gère nativement le zoom, le dézoom, et le déplacement de l'affichage de la carte.

Grâce à Egui-Plot, le dessin des différents objets, des joueurs, et des limites du jeu a été facilité. Il suffisait de dessiner des traits colorés avec une certaine épaisseur pour représenter ces éléments. Cette approche a permis de maintenir une qualité visuelle élevée tout en offrant une grande flexibilité pour les interactions utilisateur.

En résumé, Egui offre une solution robuste pour la création d'interfaces utilisateurs complètes, bien que nécessitant des extensions comme Egui-Plot pour les fonctionnalités de dessin interactif. Cela en fait une option intéressante pour des projets nécessitant une interface utilisateur riche et interactive, même si l'intégration de calculs physiques nécessite l'utilisation de bibliothèques supplémentaires comme Rapier2D.



## 1.7. Physique du jeu & concepts d'objets (Alan & Théo) :

Notre mission durant cette SAE est de reproduire la physique et les concepts d'objets qui existaient déjà dans le serveur Java, et de les adapter pour qu'elles soient compatibles avec les nouvelles technologies utilisées par notre projet.

### 1.7.1 Adaptation au choix de librairies (Alan)

Mon code et ma gestion de la physique a dû évoluer au fur et à mesure du développement, pour être en adéquation avec les bibliothèques choisies par le groupe. Dans un premier temps, l'utilisation de Raylib, qui ne gère que de l'affichage, impliquait qu'absolument tous les calculs physiques devaient être effectués à part et en parallèle, et que chaque calcul effectué devait mettre à jour manuellement l'information à l'écran.

Ensuite, lorsque nous avons voulu tester la bibliothèque Bevy, je me suis rendu compte que cette ressource possédait un affichage des calculs physiques intégré, ce qui pourrait faciliter et alléger mon travail précédent.

Finalement, comme nous avons décidé d'utiliser Egui et Rapier2D, j'ai dû basculer à nouveau vers un calcul complet de la physique, mais tester différentes bibliothèques successivement m'a pu permettre de comprendre et de tester la flexibilité de mon code et ma logique.

### 1.7.2 Joueurs et bots (Alan)

Afin de garder une rétrocompatibilité avec les bots qui avaient été conçu sur l'ancien serveur, beaucoup de principes quant à la structure d'un vaisseau ont dû être conservées. Par exemple, afin de pouvoir recevoir le format de données basé sur le principe de moteur de droite et de moteur de gauche (ici appelé des roues), j'ai dû conserver ces éléments et les réadapter pour retransmettre ces informations à la partie graphique du serveur.

```rust
#[derive(Component)]
pub(crate) struct ship{
    pub(crate) body: MovingObject,
    pub(crate) right_wheel: f32,
    pub(crate) left_wheel: f32,
    pub(crate) gun_orientation: [f32; 2],
}
```

### 1.7.3 Déplacements du vaisseau (Alan)

J'ai dû, pour recréer les déplacements en fonction de l'orientation du vaisseau, calculer la rotation du vaisseau en fonction de l'activation de ses moteurs de gauche et de droite. Bien que je me sois basé sur ce qui était déjà fait, de nombreux ajustements ont dû être faits pour coller aux bibliothèques choisies.

```rust
if let Ok((mut ship :Mut<ship> , mut transform :Mut<Transform> )) = query.get_single_mut() {
    ship.update_wheels(keyboard_input);

    let delta_seconds :f32 = time.delta_seconds();
    let rotation_speed :f32 = std::f32::consts::PI;
    let forward_speed :f32 = 500.0;

    if ship.left_wheel == 1.0 && ship.right_wheel == 1.0 {
        // Move forward
        let forward :Vec3 = transform.rotation * Vec3::Y;
        transform.translation += forward * forward_speed * delta_seconds;
    } else if ship.left_wheel == 0.0 && ship.right_wheel == 0.0 {
        // Move backward
        let backward :Vec3 = transform.rotation * Vec3::Y;
        transform.translation -= backward * forward_speed * delta_seconds;
    } else if ship.left_wheel == 1.0 && ship.right_wheel == 0.5 {
        // Turn right
        transform.rotate(Quat::from_rotation_z( angle: -rotation_speed * delta_seconds));
    } else if ship.right_wheel == 1.0 && ship.left_wheel == 0.5 {
        // Turn left
        transform.rotate(Quat::from_rotation_z( angle: rotation_speed * delta_seconds));
    }
}
```

### 1.7.5 Gestion des limites du terrain (Théo)

Pour empêcher les vaisseaux de sortir des limites du terrain, j'ai dû créer des bordures physiques à l'aide de Bevy, qui empêche tout objet physique de les traverser. Cela permet d'éviter que les vaisseaux puissent sortir des limites de la carte, ou qu'ils puissent se placer hors d'atteinte des autres joueurs.

### 1.7.6 Notions de forces de propulsion et de collision (Théo)

Grâce à la bibliothèque de Bevy, j'ai pu tester différentes façons d'implémenter des calculs physiques sur des objets, notamment des forces de propulsion, qui réagissent lors de l'impact d'un objet. Grâce à cela, il serait par exemple possible d'émettre une force de recul sur les objets se percutant, comme deux vaisseaux ou un vaisseau et un obstacle. Ces principes n'étant pas dans le projet initial, je trouvais intéressant de pouvoir les implémenter pour ajouter une touche de réalisme au jeu.

# 2. Structuration du projet

## 2.1. Organisation du code

Pour l'organisation du code, nous avons décidé de le répartir en deux catégories principales : la partie serveur et la partie interface utilisateur (UI).

Partie Serveur :

Cette partie contient tout le code relatif à la gestion du serveur, y compris l'instanciation du serveur, la gestion des messages reçus et à envoyer, la gestion de la physique (collisions et déplacements), et les entités avec la logique qui les compose. Voici la structure des dossiers et fichiers pour la partie serveur :

server_thread : Le module server_thread est responsable de gérer les connexions réseau pour le serveur. Il écoute sur un port spécifié, accepte les connexions entrantes et délègue la gestion de chaque connexion à un ClientHandler.

client_handler : Gère toute la logique de connexion d'un client au serveur, ainsi que la gestion des messages envoyés au serveur et la réception des données renvoyées par le serveur. Module principal pour client_handler, qui contient la création d'un client, la fonction run (qui gère le cycle de vie d'un client, le timeout, les messages, etc.), et la fonction process_message (qui traite les commandes des joueurs et les informations renvoyées au client).

Game_logic : permet de gérer toute la logique de jeu à savoir le moteur physique, les entités sur la map, les tirs des joueurs. Game logic permet de réinitialiser la map, d'ajouter des entités, gérer les IA…

Partie Interface Utilisateur (UI) :

Cette partie contient tout le code relatif à l'interface utilisateur, divisée en deux sous-catégories : l'interface utilisateur du jeu et l'interface utilisateur du serveur.

game_ui/ : Contient l'interface utilisateur du jeu.

server_ui/ : Contient l'interface utilisateur du serveur avec la gestion des logs, la possibilité de modifier les options du serveur.

Autres Fichiers :

app_defines.rs : Contient les variables globales dans l'ensemble du projet, avec les paramètres par défaut du serveur ou les messages que l'on peut envoyer.

main.rs : Point d'entrée de l'application, contenant la fonction main qui va lancer le serveur et également les UI du jeu et du serveur.

types.rs : Fichier dédié aux types globaux utilisés dans le projet, il contient par exemple les implémentations des messages personnalisé.

Configuration Cargo :

Cargo.toml : Fichier de configuration de Cargo pour gérer les dépendances et les informations sur le projet.

Cargo.lock : Fichier de verrouillage de Cargo, assurant que les mêmes versions de dépendances sont utilisées.

Cette structuration claire et hiérarchisée du code facilite la maintenance et l'évolution du projet en séparant les différentes responsabilités en modules logiques bien définis.

## 2.2. Choix de conception :

Pour la conception de notre projet, nous avons dû prendre plusieurs décisions cruciales afin d'assurer une architecture robuste et maintenable. Voici les principaux choix de conception que nous avons faits :

Utilisation de Rust comme langage de programmation

Sécurité de la mémoire : Rust offre une gestion de la mémoire très sûre grâce à son système de possession (ownership) et d'emprunt (borrowing). Cela élimine pratiquement les risques de fuites de mémoire et de segmentation.

Performance : Rust est en langage compilé, offrant des performances proches de celles de C/C++ tout en garantissant une sécurité mémoire.

Concurrence : Rust facilite l'écriture de code concurrent sans erreurs grâce à ses garanties de sécurité à la compilation.

Bonnes pratiques de programmation : Rust encourage les bonnes pratiques telles que l'immutabilité par défaut et la gestion explicite des erreurs, son fort typage et la gestion des erreurs que ce soit à la compilation ou bien en créant des enum d’erreur, les matchs, les results et les options.

Choix des libraries Egui & Rapier2D

Egui permet de créer des interfaces utilisateurs complètes avec des composants intégrés comme des boutons et des tableaux. Egui-Plot, une extension d'Egui, permet de dessiner des graphiques interactifs, simplifiant l'affichage des objets de jeu comme la carte ou les vaisseaux.

Rapier2D est une bibliothèque de simulation physi que 2D très performante et précise, qui gère les collisions, les forces, et les interactions entre objets. Elle permet de

simuler des dynamiques complexes de manière efficace, ce qui est essentiel pour notre jeu. L'intégration de Rapier2D dans notre projet offre une gestion robuste des mouvements et des interactions physiques des vaisseaux, garantissant un comportement réaliste et cohérent des objets dans le jeu.

En conclusion, nous avons opté pour la combinaison Egui & Rapier2D pour sa facilité de création d'interfaces utilisateurs complexes avec Egui et sa flexibilité pour les interactions physiques grâce à Rapier2D. Cette combinaison nous permet de bénéficier des avantages d'une interface utilisateur riche et interactive tout en assurant des simulations physiques précises et efficaces, bien que l'utilisation d'Egui-Plot soit nécessaire pour certaines fonctionnalités de dessin interactif.

## 3. Les difficultés rencontrées

Lors du développement, nous avons rencontré plusieurs difficultés, notamment lors du choix et de l'analyse des différentes bibliothèques possibles, que ce soit pour la partie moteur physique ou l'aspect graphique. En particulier, le fait de lancer la partie graphique et le serveur dans deux threads différents nous a posé beaucoup de problèmes. En effet, de par sa nature bloquante, l'interface egui ne permet pas d'avoir un autre thread en parallèle.

Le temps a également été un autre point de difficulté. En effet, nous pensions que Rust était une meilleure alternative au Java en raison de ses nombreux avantages. Tout d'abord, Rust offre une gestion de la mémoire plus sûre grâce à son système de possession (ownership) et d'emprunt (borrowing), ce qui élimine les risques de fuites de mémoire et de segmentation. Ensuite, Rust propose une compilation optimisée qui permet des performances proches du C/C++, tout en garantissant une sécurité mémoire. De plus, Rust bénéficie d'une communauté active et de bibliothèques modernes et performantes. Enfin, le langage encourage les bonnes pratiques de programmation, telles que l'immutabilité par défaut et la gestion explicite des erreurs avec le type Result.

Cela dit, même si en théorie Rust est plus performant, le temps imparti pour réaliser ce projet ne nous a pas permis de réécrire le serveur en Rust et d'implémenter toutes les fonctionnalités que nous souhaitions. La courbe d'apprentissage de Rust, bien que justifiée par ses avantages, est également un facteur à considérer dans la gestion du temps de développement.

# 4. Répartition des tâches

| Tache | Alan | Ethan | Théo | Yohann |
|---|---|---|---|---|
| **Serveur** | 0% | 100% | 0% | 0% |
| **Interface serveur** | 0% | 100% | 0% | 0% |
| **Interface jeu** | 0% | 0% | 0% | 100% |
| **Physique** | 60% | 0% | 40% | 0% |
| **Cahier des charges** | 76% | 8% | 8% | 8% |
| **Compte Rendu** | 30% | 30% | 10% | 30% |
| **Globalité du projet** | 26% | 26% | 22% | 26% |