IUT de Valence – Université Grenoble Alpes

DOCUMENT D'ANALYSE

SAÉ 6.1

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

1. État de l'existant sur le projet

1.1. Analyse du projet :

Ce projet est un serveur en Java, qui doit permettre la connexion à plusieurs joueurs qui pourront jouer sur une partie gérée par le serveur. La particularité de ce projet réside dans l'existence d'un code déjà fonctionnel, qui doit donc être modifié et amélioré. Parmi les différents points primordiaux qui restent à améliorer, le manque de documentation technique peut s'avérer un obstacle, tant pour nous d'un point de vue de reprise du code, mais aussi pour un développeur côté client, qui souhaiterait développer son propre bot.

De plus, l'interface "controlledElement" ne possède pas d'implémentation. Les objets isInstanceOf et controlledElement dans Environnement2D ont une écriture confuse, ce qui peut poser problème en cas de besoin de modification ou de mise à jour de ces fonctionnalités. Il serait donc préférable de les réécrire dans une manière plus propre et lisible.

Pour un souci de réutilisation, les bots et les joueurs sont à ce jour instanciés par la même structure "bot", mais il serait préférable de les séparer pour tracer un trait clair entre les deux.

Par ailleurs, beaucoup de code semble délaissé : il est soit en commentaire, soit non exploité. Si ces parties sont vraiment inutiles, il faudrait les supprimer pour gagner en organisation et en lisibilité.

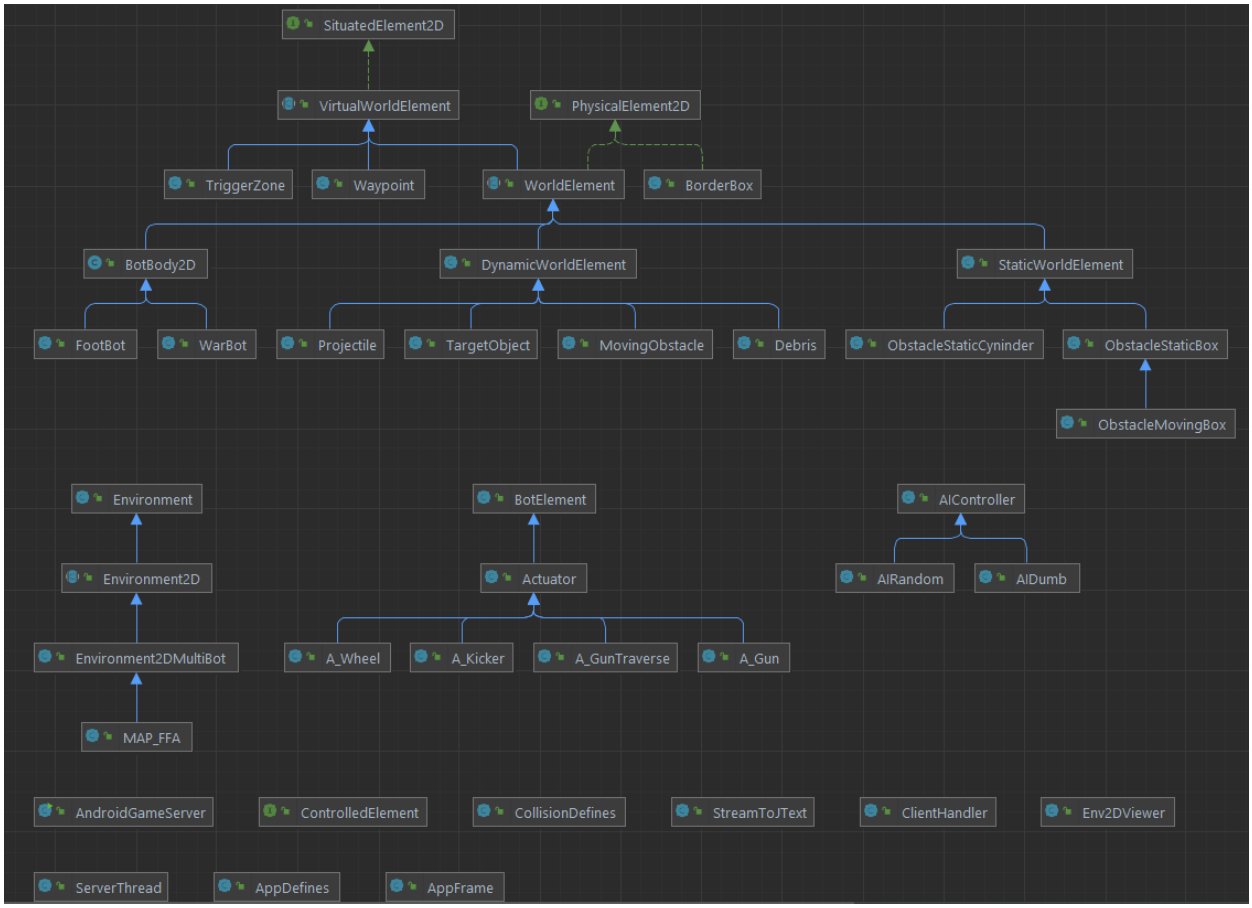
Le jeu possède à ce jour deux modes de jeu : l'un avec des tanks, l'autre avec des joueurs de football. Il semble par ailleurs que le système de zone de détection y soit déjà implémenté, ainsi qu'un système d'obstacle mouvant, qui n'est pour lors pas encore utilisé. Les obstacles peuvent, en revanche, prendre différentes formes.

Pour le système d'obstacle mouvant, la classe ObstacleMovingBox, qui désigne ce comportement, hérite de ObstacleStaticBox et non de movingElement. Il serait pourtant préférable de compter l'obstacle mouvant comme un élément qui se déplace et qui agit comme un obstacle plutôt qu'un obstacle statique qui peut se déplacer.

Le système d'équipes semble lui aussi déjà implémenté dans le code, bien que non utilisé. Cependant, parmi toutes ces fonctionnalités, bien qu'elles soient déjà implémentées mais non utilisées, elles nécessiteront une mise à jour, et tout particulièrement la façon dont le serveur gère les commandes reçues du client (clienthandler).

Enfin, même si certains noms de variables et de paramètres pourraient être changés/améliorés pour être plus clairs, le code respecte globalement les principes SOLID du développement.

1.2. Diagramme des classes existantes :



2. Propositions d’améliorations :

Après une analyse approfondie, nous avons identifié plusieurs pistes d'amélioration pour optimiser la qualité et les fonctionnalités de notre système. :

2.1. Améliorations de la qualité de développement :

2.1.1. Implémentation de tests unitaires et fonctionnels :

Actuellement, le code ne dispose d'aucun test unitaire ou fonctionnel, ce qui complique sa maintenabilité. La mise en place de ces tests permettra de détecter et de corriger les erreurs plus efficacement, assurant ainsi une meilleure stabilité du logiciel à long terme.

2.1.2 Documentation :

La documentation du code est actuellement insuffisante. Pour remédier à cela, nous allons utiliser CargoDoc pour générer une documentation complète et détaillée. Cela facilitera la compréhension et la maintenance du code par tous les développeurs qui travailleront sur ce projet à l'avenir.

2.1.3. Amélioration des noms de variables :

Les noms de variables utilisés dans le code sont parfois peu explicites, ce qui nuit à la lisibilité et maintenabilité sur le long terme. En adoptant des conventions de nommage plus claires et descriptives, nous améliorerons la lisibilité et la compréhension du code, ce qui facilitera le travail des développeurs.

2.1.4. Optimisation du code :

La robustesse de notre application peut être améliorée par une optimisation plus poussée du code. En révisant et en optimisant les algorithmes et les structures de données utilisés, nous pouvons augmenter les performances et la réactivité de notre système.

2.2. Nouvelles fonctionnalités et implémentations :

Etant donné que nous avons fait le choix de réécrire le serveur dans un langage de programmation différent du code existant, les fonctionnalités indiquées ci-dessous sont données à titre indicatif et seront réalisées uniquement s'il nous reste du temps.

2.2.1. Refonte du serveur en Rust :

Nous prévoyons de migrer notre serveur de Java vers Rust. Cette transition nous permettra de bénéficier des performances et de la sécurité accrues offertes par Rust, tout en réduisant la consommation de ressources et en augmentant la stabilité du serveur.

2.2.2. Création de bases de données pour les classements :

Nous allons implémenter une base de données pour suivre les classements, à la fois globaux et individuels. Cela permettra aux joueurs de consulter leur progression et leurs performances au fil du temps, ajoutant une dimension compétitive et motivante au jeu.

2.2.3. Introduction de nouveaux types de munitions et d'entités :

Nous allons introduire de nouveaux types de munitions et d'entités jouables, comme des bateaux et des avions, chacun ayant des caractéristiques uniques. Cela diversifiera le gameplay et offrira aux joueurs de nouvelles stratégies et expériences de jeu.

2.2.4. Développement de nouveaux modes de jeu :

Nous pensons ajouter plusieurs nouveaux modes de jeu pour diversifier les options de gameplay, dans la mesure où le temps qui nous est imparti sera soit plus que suffisant. Les modes envisagés incluent :

- Mode Temps : Les joueurs doivent accomplir des objectifs dans un temps imparti.
- Mode Vie : Les joueurs ont un nombre limité de vies.
- Mode Équipe : Les joueurs s'affrontent en équipes.
- Mode Capture de drapeau : Les joueurs doivent capturer le drapeau de l'équipe adverse et le ramener à leur base.

2.2.5. Protection contre les attaques DDoS :

Pour assurer la stabilité du serveur, nous allons mettre en place des mécanismes de protection contre les attaques DDoS. En limitant le nombre de requêtes par utilisateur, nous empêcherons les tentatives de surcharge du serveur, garantissant ainsi une expérience de jeu fluide pour tous les utilisateurs.

2.3. Améliorations supplémentaires (si le temps le permet) :

2.3.1. Externalisation du rendu visuel :

Au lieu de traiter la partie visuelle sur le serveur, nous proposerons une API permettant à chaque client de gérer l'affichage lui-même. Cela allégera la charge sur le serveur et permettra aux clients de personnaliser leur rendu visuel, offrant ainsi une expérience plus flexible et personnalisée.

Ces améliorations, une fois mises en œuvre, renforceront la qualité et la robustesse de notre développement, tout en enrichissant l'expérience utilisateur grâce à de nouvelles fonctionnalités et modes de jeu.

3. Choix techniques :

Pour la refonte du serveur de jeu dans un nouveau langage de programmation, différents langages se présentaient à nous. Afin de pouvoir faire un choix, nous avons réalisé un SWOT des langages choisis.

3.1. C#

Forces :

Le C# est un langage développé par Microsoft, et qui reçoit encore aujourd'hui des mises à jour. Cela prouve que le langage ne compte pas être abandonné, et continuera d'évoluer avec son temps.

Étant un langage compilé, il est par définition plus puissant que d'autres langages comme Java ou Rust. A noter que les langages compilés sont plus sécurisés, car ils permettent d'envoyer au client seulement les fichiers compilés, les rendant donc difficilement décryptables.

Faiblesses :

Un des problèmes du C# est qu'il a une syntaxe et une structure plutôt compliquée, et il semble difficile d'apprendre, de coder, et d'avoir suffisamment de recul sur le développement en C# durant nos 60h de SAÉ pour arriver à un résultat concluant.

De plus, il n'est pas vraiment possible de générer de la documentation développeur, et elle devra donc être réalisée à la main. Cela devra donc s'ajouter à notre emploi du temps plus que chargé, en plus d'acquérir une maîtrise satisfaisante du langage.

Opportunités :

Le C# a une partie graphique native, n'obligeant pas à avoir appel à des librairies externes, ce qui est un plus par rapport à d'autres langages.

Le langage permet également d'utiliser le framework Microsoft .NET, qui est très populaire depuis de nombreuses années au sein de la communauté des développeurs.

Menaces :

Choisir le langage C# est un risque car il implique d'apprendre et de mettre à l'œuvre des compétences totalement nouvelles dans un langage qui nous est presque inconnu, et réduirait considérablement du temps qui aurait dû être attribuable à de la qualité de développement ou de l'ajout de fonctionnalités annexes et complexes.

3.2. Java

Forces :

Java dispose d'une vaste bibliothèque ainsi que d'une communauté active qui fournit une multitude de bibliothèques tierces, de frameworks et d'outils de développement. Cela facilite le développement d'applications complexes.

Par ailleurs, Java offre un large éventail de frameworks et d'outils pour les tests unitaires, d'intégration et de performance, ce qui facilite le développement de logiciels robustes et fiables.

De plus, la capacité d'intégrer une documentation détaillée directement dans le code source via Javadoc simplifie la maintenance et le partage des informations sur les API, améliorant ainsi la collaboration et la compréhension du code.

Enfin, la taille et la diversité de la communauté Java offrent un soutien significatif aux développeurs, avec des forums, des groupes de discussion, des tutoriels et des ressources en ligne abondantes, ce qui facilite l'apprentissage et le développement.

Faiblesses :

Tout d'abord, Java est souvent critiqué pour sa syntaxe délicate et sa complexité relative par rapport à certains langages plus modernes. Les développeurs peuvent trouver qu'il est plus compliqué d'apprendre Java plutôt qu'un autre langage.

Ensuite, bien que les performances de Java aient considérablement progressé, elles peuvent toujours être inférieures à celles de langages compilés dans certains cas d'utilisation spécifiques.

Enfin, étant un langage interprété, Java peut présenter des surcoûts de performances par rapport aux langages compilés, ce qui peut être un inconvénient pour les applications nécessitant une exécution très rapide ou une faible consommation de mémoire.

Opportunités :

Avec l'essor de l'Internet des objets, du Big Data, de l'intelligence artificielle et d'autres domaines technologiques émergents, Java a l'opportunité de se positionner comme un langage de choix pour le développement d'applications dans ces domaines.

Par ailleurs, Java est de plus en plus ouvert, avec des projets tels que OpenJDK, ce qui permet une plus grande participation de la communauté et une évolution plus rapide du langage.

Menaces :

Les langages comme Python, JavaScript, et Kotlin gagnent en popularité pour leur syntaxe plus concise et leur facilité d'utilisation. Avec le temps, il se peut que Java soit délaissé au profil de ces langages considérés plus simples.

Bien que Java ait une réputation de sécurité, les failles de sécurité et les vulnérabilités peuvent toujours poser problème. Les attaques ciblant la JVM ou les bibliothèques Java peuvent affecter la perception de la sécurité de la plateforme dans son ensemble.

3.3. Rust

Forces :

Rust offre des performances proches de celles du langage C/C++, mais avec des garanties de sécurité supplémentaires. Cela signifie que les serveurs fonctionnant en Rust peuvent gérer un grand nombre de connexions et de requêtes de manière efficace.

Côté sécurité, Rust assure une sécurité renforcée en garantissant la gestion sécurisée de la mémoire. Son modèle de possession et de prêt (ownership and borrowing) empêche les erreurs courantes comme les dépassements de tampon (buffer overflows), les références nulles (null references) et les conditions de course (race conditions). Ces caractéristiques sont intégrées directement dans le langage et vérifiées à la compilation, réduisant ainsi significativement les risques de vulnérabilités que l'on rencontre souvent dans d'autres langages comme C ou C++.

Le système de types avancé de Rust, couplé à ses règles strictes de gestion de la mémoire, permet de prévenir efficacement les erreurs de mémoire. Par exemple, Rust garantit que chaque morceau de mémoire est possédé par une seule variable à la fois, et les références à cette mémoire doivent respecter des règles strictes de durée de vie (lifetimes). Cela permet d'éviter les situations où plusieurs parties du programme pourraient essayer de modifier la même donnée en même temps, ce qui conduit souvent à des bugs difficiles à déboguer dans d'autres langages.

Rust est un langage fortement typé, ce qui améliore grandement la qualité du développement. Cela se traduit par une meilleure lisibilité du code, une maintenance facilitée et une gestion des erreurs plus efficace.

Faiblesses :

Le langage Rust n'est pas vraiment fait pour l'orienter objet, et par conséquent, il n'y a pas à proprement parler de notion d'interface ou de classe abstraite, ce qui peut complexifier le développement et le fonctionnement d'un serveur qui a une interface graphique.

Opportunités :

Le langage Rust est relativement jeune (moins de 10 ans) et devient de plus en plus populaire. Il peut être donc intéressant de regarder de près ce langage, qui commence bien assez à se faire une place sur la scène des langages de programmation.

Rust est très rigide, et bien que cela puisse être restrictif par moments (comme cité précédemment), cette rigidité peut nous forcer à prendre de bonnes habitudes dans notre façon de coder et dans notre sensibilité au code.

Menaces :

Puisqu'il est très jeune, les aides et forums liés à Rust sont moins fournis que pour des langages bien plus anciens et bien plus utilisés. En cas de problème, il sera plus difficile de trouver une réponse ou une aide en ligne en Rust.

Le manque de librairies de Rust peut grandement ralentir notre développement, de par l'absence de moteur physique, mais aussi par la structuration même du langage, qui demande de mettre en place des spécifications plus complexes que pour un langage plus commun.

3.4. Langages et librairies retenues

Nous avons décidé au vu des différentes réflexions que l'on a eu et une analyse approfondie des différents langages disponibles (voir Annexes pour le brainstorming), nous avons choisi Rust comme langage pour refaire le serveur. Rust nous semble être un excellent choix en raison de ses performances, de sa sécurité et de sa fiabilité.

Pour la gestion des connexions réseau, essentielle à la communication dans les applications serveur et client, nous avons opté pour les bibliothèques intégrées de Rust, à savoir **TcpListener** et **TcpStream**. **TcpListener** nous permettra d'écouter les connexions TCP entrantes sur une IP et un port spécifique, tandis que **TcpStream** facilitera la lecture et l'écriture des données sur ces connexions.

Pour la partie graphique, deux choix de libraries s'offraient à nous : raylib et egui. Raylib est une librarie écrite pour le langage de programmation c mais qui fonctionne également pour le rust. Cela dit, la documentation sommaire de cette librarie pour rust nous a fait choisir la librarie egui qui elle bénéficie d'une documentation complète avec des exemple concrets. L'autre avantage de egui c'est que cette librarie comporte des composants physiques tels que des boutons, des graphiques, des tableaux… qui sont plus difficiles à faire avec raylib.

Pour la partie moteur physique, nous avons choisi bevy, bevy_rapier2d

4. Planification des tâches

4.1. Structuration des tâches

Tâche 1 : Serveur côté logique :

Il faut pouvoir reproduire le comportement logique du serveur Java en Rust, pour qu'il puisse prendre en charge les mêmes fonctionnalités de base, à savoir :

- Créer une partie
    - Le serveur doit être capable d'ouvrir une partie, c'est à dire de lancer une session ouverte aux connexions clients, et être prêt à recevoir des connexions puis instructions de ces clients.
    - Priorité : MAXIMALE
    - Difficulté : 9/10

- Fermer une partie
    - Le serveur doit être capable de mettre fin à une partie, c'est à dire de fermer sa session aux connexions clients.

    - Priorité : MAXIMALE
    - Difficulté : 6/10

- Ouvrir une connexion avec un joueur
    - Le serveur doit être capable de laisser un joueur se connecter.
    - Priorité : Haute
    - Difficulté : 8/10

- Fermer une connexion avec un joueur
    - Le serveur doit être capable de mettre à terme la connexion avec un joueur.
    - Priorité : Haute
    - Difficulté : 6/10

- Recevoir les actions du joueur et les appliquer :
    - Changer son nom
        - Le serveur doit être capable, dans un premier temps, de pouvoir recevoir une chaîne de caractères et de l'attribuer au joueur qui est en est source, et dans un second temps, de l'utiliser dans la partie graphique pour l'afficher au-dessus du joueur et dans le tableau des scores. La donnée est envoyée par le client par la fonction NAME qui renvoie une chaine de caractères.
        - Priorité : Faible
        - Difficulté : 3/10

    - Changer sa couleur (par un hexadécimal OU RBG)
        - Le serveur doit être capable de recevoir un hexadécimal ou un RGB de la part du client dans un premier temps, pour pouvoir mettre à jour sa couleur dans la partie graphique dans un second temps. La donnée est envoyée par le client par la fonction COL qui renvoie 3 int (RGB) ou 1 int (hexadécimal).
        - Priorité : Faible
        - Difficulté : 3/10

    - Déconnecter le joueur
        - Le serveur doit être capable de recevoir l'information de déconnexion de la part du client, et de mettre un terme à sa connexion. La donnée est envoyée par le client par la fonction EXIT.
        - Priorité : Moyenne
        - Difficulté : 4/10

    - Recevoir le statut de connexion du joueur

        - Le serveur doit être capable de recevoir le statut de connexion du joueur. Ceci n'a aucune influence sur la partie en cours. La donnée est envoyée par le client par la fonction LIVE.
        - Priorité : Moyenne
        - Difficulté : 4/10

    - Puissance des moteurs, et calculer la direction en fonction
        - Le serveur doit être capable de recevoir des informations sur les moteurs du client, et de calculer la direction du joueur en fonction. Les données sont transmises par le client par les fonctions MotL, MotR. Norme de donnée : 0 pleine vitesse arrière, 1 pleine vitesse avant, 0.5 stop.
        - Priorité : Haute
        - Difficulté : 6/10

    - Direction du canon pour viser
        - Le serveur doit être capable de recevoir des informations sur l'orientation du canon du joueur, et de l'orienter en fonction.
        - Priorité : Moyenne
        - Difficulté : 7/10

    - Action de tir
        - Le serveur doit être capable de recevoir une instruction de tir de la part du client, et d'effectuer l'action.
        - Priorité : Moyenne
        - Difficulté : 3/10

    - Afficher un message pendant une courte période sous le vaisseau du joueur
        - Le serveur doit être capable de recevoir un message de statut d'un joueur dans un premier temps, et ensuite de le transmettre à la partie graphique sous le véhicule du joueur.
        - Priorité : Basse
        - Difficulté : 2/10

- Envoyer des informations au joueur
    - Renvoyer l'orientation du joueur
        - Le serveur doit être capable de renvoyer au client l'orientation ABSOLUE de son avatar. La valeur renvoyée doit être un float compris entre 1 et 2PI.
        - Priorité : Basse
        - Difficulté : 4/10

    - Renvoyer les informations de l'adversaire le plus proche du joueur

        - Le serveur doit être capable de renvoyer au client les informations sur l'adversaire le plus proche au format : "orientation"/"distance" (ex : 0.2/143.5).
        - Priorité : Moyenne
        - Difficulté : 5/10

    - Renvoyer les informations sur le projectile le plus proche
        - Le serveur doit être capable de renvoyer au client les informations sur le projectile le plus proche au format "orientation"/"distance" (ex : 0.2/143.5).
        - Priorité : Basse
        - Difficulté : 5/10

    - Renvoyer la liste des joueurs connectés
        - Le serveur doit être capable de renvoyer au client la liste des joueurs connectés. Les noms de chaque joueur doit être séparé par "=".
        - Priorité : Basse
        - Difficulté : 5/10

- Calculer le déplacement d'un vaisseau et d'un tir
    - A partir de son orientation et d'une vitesse fixe, le serveur doit être capable de calculer la direction que prend un vaisseau et un tir, et de savoir s'il y a collision avec un autre élément du jeu.
    - Priorité : Moyenne
    - Difficulté : 7/10

- Ajouter des bots ne pouvant que tirer ("Dumb bot")
    - Le serveur doit être capable d'instancier et d'ajouter un ou plusieurs "dumb bot", c'est à dire des vaisseaux dont la seule action est de tirer.
    - Priorité : Moyenne
    - Difficulté : 6/10

- Ajouter des bots agissants aléatoirement ("Random bot")
    - Le serveur doit être capable d'instancier et d'ajouter un ou plusieurs "random bot", qui se déplacent et tirent de manière aléatoire.
    - Priorité : Moyenne
    - Difficulté : 7/10

- Supprimer les bots
    - Le serveur doit être capable de supprimer tous les bots présents dans la partie, c'est à dire retirer tous les vaisseaux qui ne sont pas des joueurs.
    - Priorité : Basse

    - Difficulté : 5/10

- Interpréter plusieurs commandes à la chaine dans un seul message
    - Lorsqu'un joueur envoie un message pour effectuer une action, il est possible qu'il envoie un message contenant plusieurs actions succinctes à effectuer. Il faut dans ce cas être capable d'interpréter et d'effectuer ces actions à la chaine. Les requêtes successives sont séparées par le caractère "#".
    - Priorité : Basse
    - Difficulté : 4/10

- Créer des obstacles
    - Le serveur doit être capable de créer des obstacles positionnés sur la carte, et qui doivent bloquer les joueurs au contact.
    - Priorité : Moyenne
    - Difficulté : 7/10

- Envoyer diverses informations à la partie graphique
    - Lorsqu'un joueur effectue une action, ou qu'un événement se produit, il est important que les bonnes informations soient envoyées à la partie graphique pour que l'affichage soit rafraichi.
    - Priorité : Haute
    - Difficulté : 7/10

Tâche 2 : Serveur côté graphique :

La partie graphique du serveur doit être capable d'afficher diverses informations :

- La position du joueur
    - La position du joueur ainsi que son orientation doivent être mis à jour à chaque déplacement pour être en raccord avec sa position/orientation logique.
    - Priorité : Haute
    - Difficulté : 7/10

- La position du tir du joueur
    - La position du tir du joueur ainsi que son orientation doivent être mis à jour en fonction de sa vitesse, pour l'afficher en temps réel.
    - Priorité : Haute
    - Difficulté : 7/10

- La couleur du joueur
    - La couleur d'un joueur doit pouvoir être mise à jour et être appliqué graphiquement.
    - Priorité : Basse
    - Difficulté : 4/10

- La liste des joueurs
    - La liste des joueurs connectés doit pouvoir être affichée sur la partie.
    - Priorité : Haute
    - Difficulté : 5/10

- Le score du joueur
    - Le score du joueur doit être affiché dans la liste des joueurs.
    - Priorité : Moyenne
    - Difficulté : 3/10

- Les obstacles
    - Les obstacles doivent apparaître graphiquement.
    - Priorité : Haute
    - Difficulté : 6/10

Tâche 3 : Nouvelles fonctionnalités possibles (priorité extrêmement basse) :

- Nouvelles entités et nouveaux types de munitions
    - Ajouter de nouveaux types de munitions, ou de vaisseaux à piloter.
    - Difficulté : 9/10

- Classer l'affichage selon le score
    - Trier l'affichage des joueurs en fonction de leur score.
    - Difficulté : 2/10

- Ajouter un compteur de morts
    - Ajouter une liste avec le nombre de morts des joueurs.
    - Difficulté : 2/10

- Limiter le nombre de vies à trois
    - Compter le nombre de morts, et si égal à trois, on empêche le joueur de réapparaitre.
    - Difficulté : 2/10

- Limiter le nombre de munitions

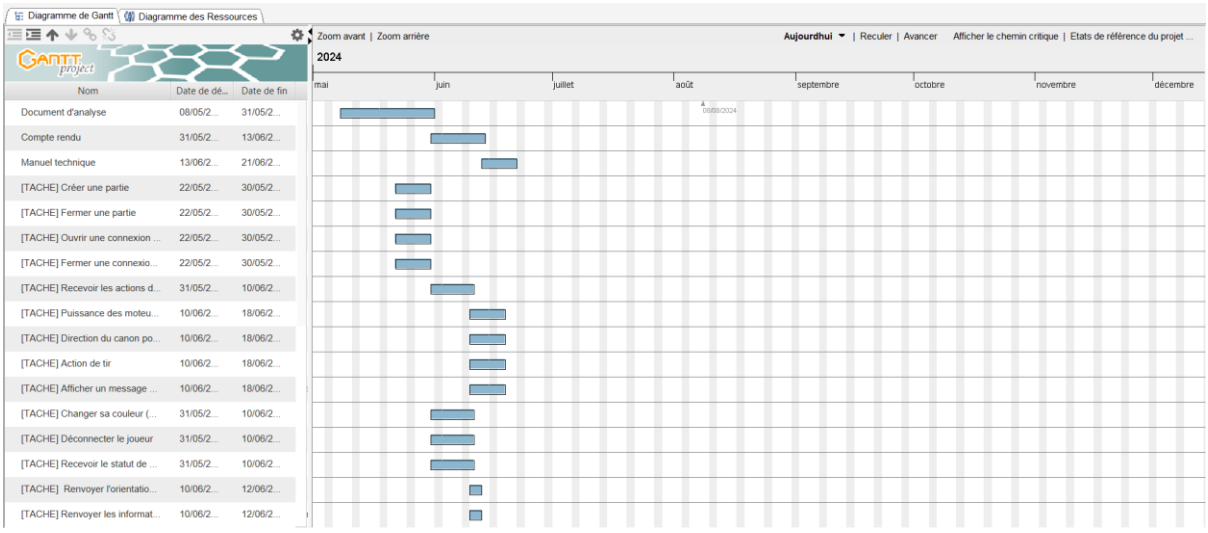
- Limiter le nombre de munitions à X munitions, et mettre un temps de recharge avant de regagner ses munitions.
- Difficulté : 5/10

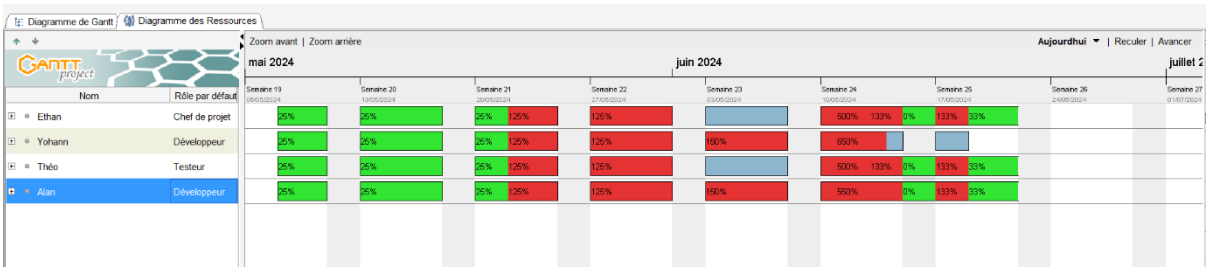
- Détecter les projectiles autres que les siens
  - Faire en sorte qu'un joueur puisse détecter les projectiles qui ne sont pas les siens.
  - Difficulté : 6/10

- Lorsqu'un véhicule est "détruit", il n'est plus perçu par les autres véhicules
  - Ajouter un statut "indétectable" qui fait ignorer aux autres vaisseaux un vaisseau qui vient de mourir.
  - Difficulté : 9/10

- Ajout de la gestion d'équipes
  - Retirer la possibilité de toucher ses coéquipiers.
  - Difficulté : 8/10

4.2. Diagramme de Gantt





6. Annexes

Images du brainstroming en amont pour les choix techniques et fonctionnalités à développer :