SAE 6.A.01
NOTE DE CADRAGE

Lloret Florian
Filali-Zegzouti Sami
Muller Valentin

Sommaire

I. Introduction

1. Contexte et objectifs

Ce document a pour but d'établir une vision claire et exhaustive des choix technologiques à entreprendre pour le développement et l'évolution du serveur de jeu existant. Le projet consiste à faire évoluer un serveur de jeu basé sur Java et JBox2D, avec la possibilité de réécrire entièrement le serveur en utilisant d'autres technologies. L'objectif est de garantir que le serveur puisse continuer à répondre aux besoins fonctionnels des applications clientes et des bots déjà développés, tout en améliorant la qualité, la performance et la maintenabilité du code.

2. Enjeux de l'analyse technique

Le choix de la technologie sous-jacente à l'application revêt une importance capitale. Il doit assurer une expérience utilisateur optimale, une performance fiable, et une maintenance aisée, tout en prenant en considération des impératifs économiques. Cette analyse technique a pour ambition d'éclairer la décision en évaluant deux scénarios distincts : l'amélioration de la base de code existante et la réécriture complète du serveur dans un autre langage. Chaque scénario présente ses propres avantages et défis.

3. Orientation stratégique

Cette note de cadrage se veut être un guide pour l'orientation stratégique du projet. Elle évaluera les technologies sous plusieurs facettes, allant de la maturité et de la performance à la réutilisabilité du code, en passant par la flexibilité fonctionnelle et l'intégration harmonieuse avec les fonctionnalités natives des appareils mobiles. En fournissant une analyse détaillée des technologies et des méthodologies de développement, ce document vise à offrir une base solide pour des décisions éclairées tout au long du cycle de vie du projet.

4. Public cible

Destinée aux décideurs, chefs de projet, et équipes de développement, cette analyse technique a pour objectif de fournir une base solide pour la prise de décision. Elle se veut être un outil de référence tout au long du cycle de vie du projet, contribuant à une implémentation réussie et pérenne de l'application. En mettant en lumière les aspects cruciaux de la technologie et de l'architecture logicielle, cette note de cadrage vise à assurer la réussite technologique du projet.

II. Présentation de l'existant

1. Architecture actuelle

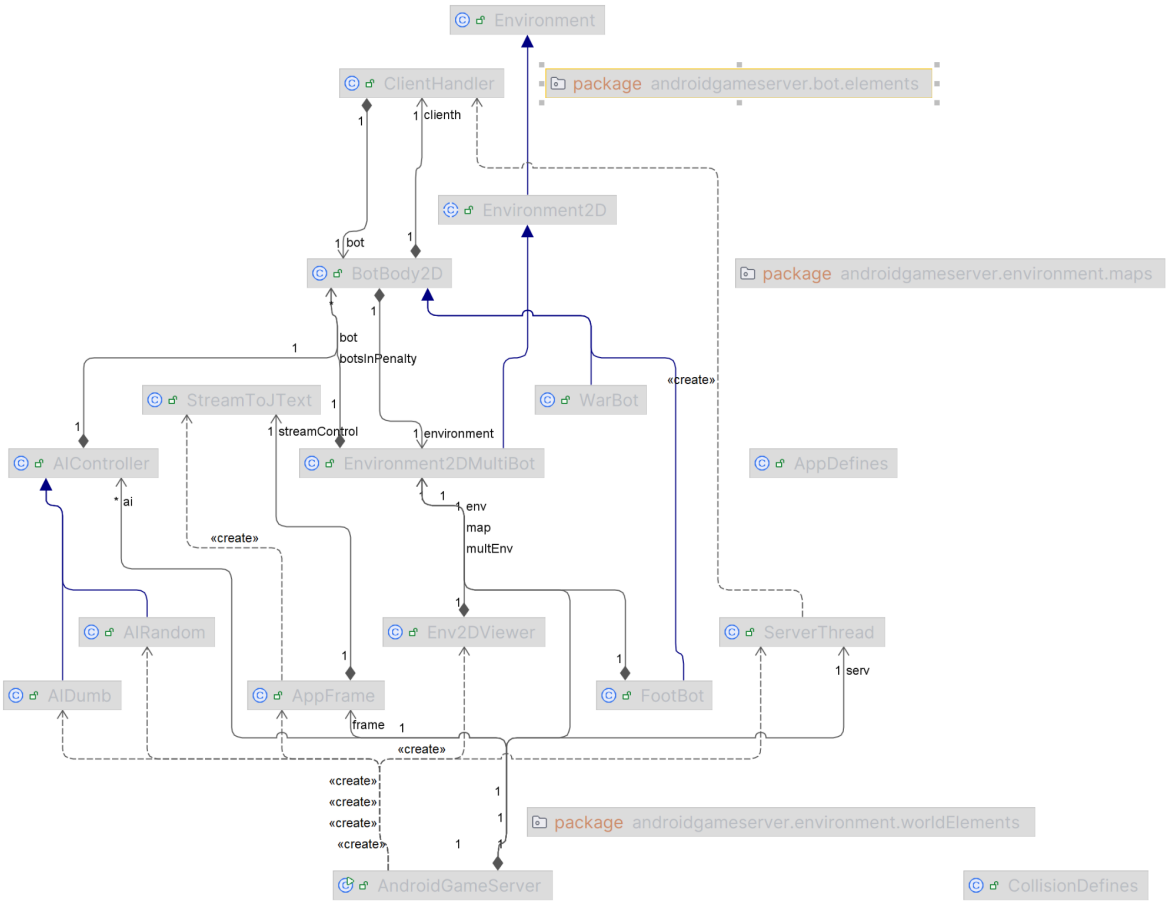

Diagramme de classe général du serveur

AndroidGameServer : Classe principale gérant l'interface graphique, le cycle de jeu, et les contrôleurs AI.

Hiérarchie de l'environnement : Environment → Environment2D → *Environment2DMultiBot* → *MAP_FFA* : Cette progression montre l'élaboration d'un environnement de jeu en deux dimensions qui supporte multiples bots, avec *MAP_FFA* représentant une configuration spécifique de la carte.

Composants de jeu : *BotElement*, *PhysicalElement2D*, *DynamicWorldElement*, et *StaticWorldElement* montrent la diversité des éléments manipulables dans le jeu, incluant des obstacles et des bots avec différentes capacités (représentés par *A_Gun*, *A_Wheel*, etc.).

Contrôleurs et interfaces utilisateur : *AppFrame* et *Env2DViewer* sont utilisés pour l'interface utilisateur, affichant le jeu et interagissant avec l'utilisateur.

La gestion réseaux : *ServerThread* et *ClientHandler*, ces deux classes gèrent les connexions TCP via un socket. Elles lisent les commandes envoyées par le client et y répondent, tout en gérant le temps d'inactivité pour déconnecter les clients inactifs ou ceux qui envoient la commande de sortie.
Le traitement des commandes est réalisé par la méthode *processCommand*, qui interprète et exécute les commandes sur le bot correspondant au client, y compris des ajustements tels que changer la couleur, le nom, ou exécuter des mouvements.

L'environnement 2D : La classe *Env2DViewer* est une extension de *JPanel* utilisée pour visualiser un environnement de jeu 2D dans le cadre du serveur de jeu *AndroidGameServer*. Elle gère l'affichage graphique de divers éléments dans un environnement multi-bot, y compris les bots, les obstacles, et d'autres éléments virtuels du monde.
La classe contient des attributs pour gérer le déplacement et le zoom de la vue, l'affichage des noms et des chemins des bots, et la configuration des couleurs via une classe interne *ColorTheme*.
Elle intègre des listeners pour les événements de la souris, permettant aux utilisateurs de déplacer la vue, zoomer et interagir avec l'environnement via le clic droit et gauche.
Elle utilise des objets *Graphics2D* pour le dessin, avec des configurations modifiables pour le tracé des lignes et le remplissage des formes.

IA : *AIController* avec ses sous-classes *AIRandom* et *AIDumb* gère l'intelligence artificielle des bots. *BotBody2D* maintient divers états comme active, name, label, et des compteurs de contacts qui sont essentiels pour gérer les interactions dans l'environnement de jeu. Les bots peuvent interagir physiquement avec l'environnement grâce à des définitions de corps (*BodyDef*) et de fixtures

(*FixtureDef*), qui sont utilisés pour définir les propriétés physiques comme la forme, la densité et les filtres de collision.
Le bot peut ajouter ou retirer des contacts avec des éléments physiques (*PhysicalElement2D*), influençant ainsi son comportement et ses interactions. Les actuators sont des composants qui modifient l'état du bot en fonction des valeurs contrôlées par les commandes reçues, comme la direction de déplacement ou des actions spécifiques. Chaque actuateur peut être normalisé et mis à jour à chaque étape de la simulation, permettant des réponses dynamiques aux commandes.

Les bots ont la capacité d'agir de manière autonome grâce à des actuators préprogrammés qui peuvent être ajustés en temps réel. La capacité de réagir aux commandes en temps réel via *ClientHandler* implique que l'IA peut être améliorée pour des stratégies adaptatives en fonction des conditions de jeu dynamiques.

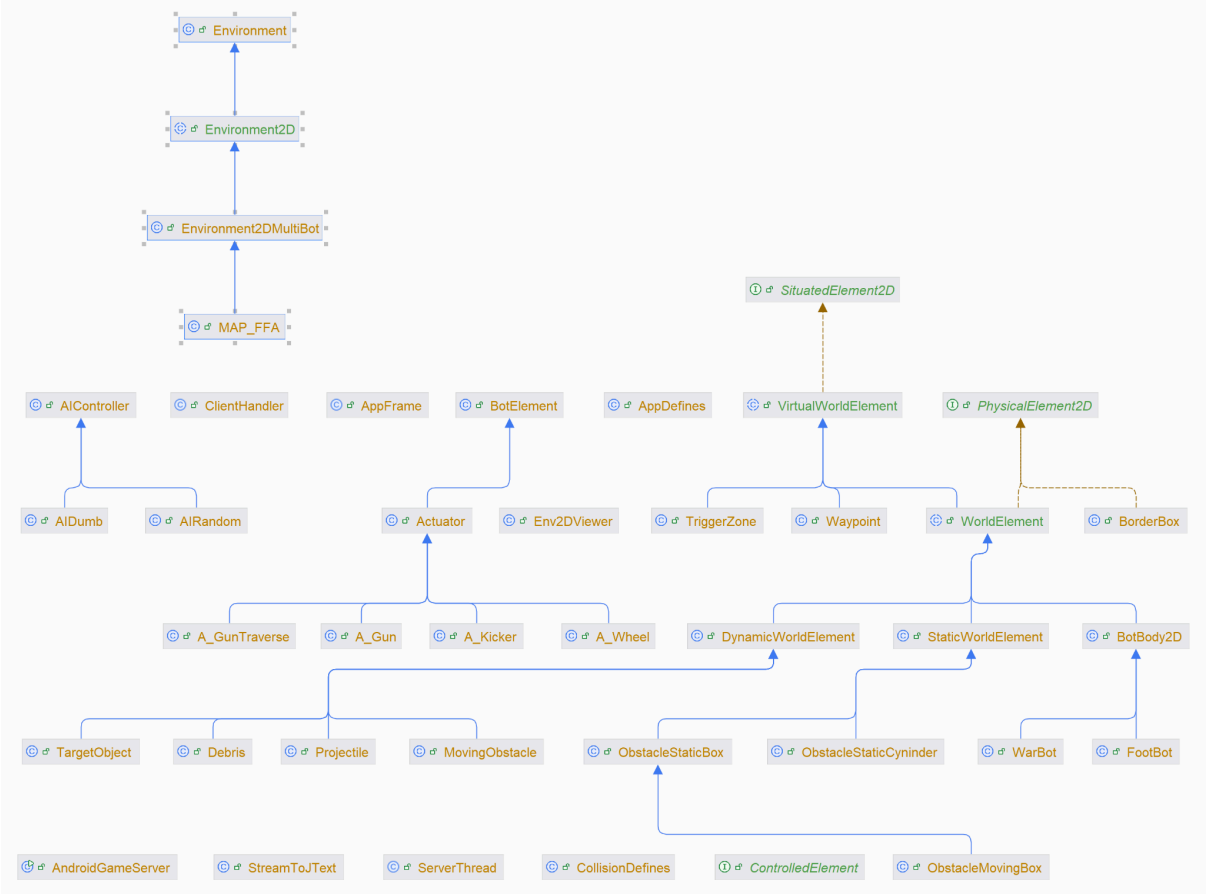

Diagramme de classe général détaillé du serveur

2. Technologies utilisées

Langage de Programmation

Java : Le cœur de l'application est écrit en Java, un langage de programmation orienté objet fortement typé, reconnu pour sa portabilité et sa robustesse. Java facilite la gestion des ressources réseau et offre des bibliothèques étendues pour le développement de systèmes complexes. Dans ce projet, Java est utilisé pour orchestrer la logique du jeu, la communication réseau, et l'interface utilisateur, tirant parti de la gestion automatique de la mémoire et du garbage collection pour maintenir des performances optimales.

Frameworks et Bibliothèques

JBox2D : Il s'agit d'une bibliothèque Java pour la simulation de la physique dans des environnements en deux dimensions. JBox2D est utilisé pour modéliser les interactions physiques entre les objets du jeu, comme les collisions et les dynamiques des bots. La bibliothèque permet de simuler des comportements physiques réalistes avec un minimum de configuration, facilitant l'implémentation de divers mécanismes de jeu tels que les mouvements des bots, les tirs de projectiles, et les interactions entre objets.

Java Swing : Employé pour construire l'interface graphique de l'application, Java Swing permet de créer des fenêtres, des boutons, des barres d'outils, et d'autres composants d'interface utilisateur. Dans notre contexte, Swing est utilisé pour rendre les éléments visuels du serveur de jeu, offrant une interaction directe avec l'utilisateur à travers des éléments graphiques interactifs et des visualisations en temps réel de l'état du jeu.

Architecture Logicielle

Architecture Orientée Objet (AOO) : L'application exploite pleinement les principes de l'AOO, ce qui est manifeste dans la structure hiérarchique des classes et les relations de composition. L'héritage est largement utilisé pour étendre les fonctionnalités des classes de base, tandis que les relations de composition sont employées pour créer des objets complexes à partir de sous-composants plus simples. Cette approche favorise la réutilisabilité du code, la modularité et

l'extensibilité de l'application, permettant une maintenance et une mise à niveau plus simples du système.

Gestion de la Concurrence

Threads et Synchronisation : Java offre un support robuste pour le multithreading, qui est utilisé dans l'application pour gérer simultanément plusieurs connexions de clients et exécuter la logique du jeu sans blocage. La synchronisation est gérée à travers des blocs synchronisés et des objets de verrouillage pour éviter les conditions de concurrence et assurer la cohérence des données entre les threads.

3. Analyse de l'existant

Qualité du code

Lisibilité : Bien que la structure du code soit logique, l'absence de commentaires détaillés et la documentation insuffisante sur les méthodes complexes et les décisions d'architecture entravent la compréhension immédiate du fonctionnement interne de l'application. Une amélioration significative serait d'ajouter des commentaires explicatifs et des descriptions de méthodes au niveau des interfaces et des classes clés.

Maintenabilité : Le code souffre d'une utilisation excessive de méthodes et de variables statiques, ce qui rend le système rigide et difficile à tester de manière isolée. Il est conseillé de réduire l'utilisation de composants statiques et d'encourager l'emploi de principes SOLID pour améliorer la modularité et la réusabilité du code.

Robustesse : Bien que des blocs try-catch soient présents pour la gestion des exceptions, la spécificité et la granularité de cette gestion sont insuffisantes pour certains scénarios critiques, comme la perte de connexion réseau ou les erreurs de synchronisation. Renforcer la gestion des exceptions par des mécanismes de reprise et de validation plus robustes serait bénéfique.

Architecture et Conception

Couplage : Le couplage fort entre *BotElement* et ses sous-classes limite la flexibilité du système. L'introduction de principes de conception comme l'inversion de contrôle (IoC) et l'injection de dépendances pourrait décompacter ce couplage et favoriser une meilleure extensibilité.

Complexité : L'architecture actuelle complexifie l'intégration de nouvelles fonctionnalités sans impacter les composants existants. Utiliser des patterns de conception comme le Mediator ou le Observer pour une meilleure séparation des préoccupations et une communication plus claire entre les composants peut alléger cette complexité.

Performance et gestion des ressources

Performance et Scalabilité : La performance de l'application est entravée par une utilisation intensive de polling et de délais manuels, ainsi que par une gestion des threads non optimisée. Refactoriser le système pour utiliser des modèles de concurrence plus avancés, tels que les acteurs ou les événements asynchrones, améliorera la performance et la scalabilité.

Gestion des Ressources : L'application consomme des ressources de manière intensive, en particulier dans la gestion des sockets. L'adoption de pools de sockets et la réutilisation de ces derniers pourraient réduire significativement la surcharge liée à la création fréquente de sockets.

Amélioration de l'IA

Algorithmes d'IA : Intégrer des techniques d'apprentissage automatique pour permettre aux bots d'apprendre et de s'adapter aux comportements des adversaires enrichirait l'expérience de jeu et augmenterait l'autonomie des agents.

Optimisation de la Performance : Améliorer l'efficacité des calculs relatifs aux interactions et aux déplacements des bots permettrait de gérer un plus grand nombre de bots simultanément avec une latence réduite, améliorant ainsi l'expérience utilisateur en temps réel.

III. Propositions d'amélioration

1. Améliorations de qualité

Pour améliorer la qualité du code, il serait utile d'ajouter davantage de commentaires, surtout autour des méthodes complexes et des choix techniques. Cela aiderait les développeurs à comprendre plus facilement ce que fait le code et pourquoi certaines décisions ont été prises. Éviter l'utilisation excessive de méthodes et de champs statiques en faveur de l'instanciation d'objets et de l'injection de dépendances rendrait le code plus modulaire et testable. Appliquer des principes comme la responsabilité unique et bien séparer les différentes parties du code le rendrait également plus facile à maintenir et à modifier.

En termes de robustesse, même si les blocs try-catch sont utilisés pour gérer les exceptions, il serait bénéfique de gérer les erreurs de manière plus spécifique. Mettre en place des gestionnaires d'exceptions adaptés à différents types de situations assurera que chaque erreur est traitée correctement. Ajouter des tests unitaires et des tests d'intégration pour couvrir divers scénarios d'erreur renforcera la solidité du système.

Le couplage fort entre certains composants, comme *BotElement* et ses sous-classes, limite la flexibilité du code. Pour améliorer cela, il serait avantageux de séparer ces composants en modules plus petits et indépendants, en utilisant des interfaces pour clarifier les relations entre eux. Des principes de conception comme la loi de Demeter et l'inversion des dépendances peuvent aider à réduire le couplage et à rendre le code plus modulaire.

Enfin, la complexité de l'architecture actuelle peut rendre difficile l'introduction de nouvelles fonctionnalités ou la modification des existantes. Pour simplifier, il serait judicieux d'adopter des modèles de conception éprouvés et une approche de développement orientée tests (TDD), ce qui obligerait à écrire du code plus simple et plus facile à tester. En clarifiant les responsabilités de chaque composant et en simplifiant la structure globale, le code devient plus facile à comprendre, à maintenir et à étendre.

2. Ajouts de fonctionnalités

Implémenter un journal de logs détaillant toutes les commandes reçues par le serveur permettrait une meilleure traçabilité et diagnostic des problèmes. Cela offrirait aux développeurs et administrateurs une vision claire des actions effectuées, facilitant ainsi le débogage et l'amélioration continue du système. Cette fonctionnalité pourrait inclure des filtres pour trier les logs par type de commande, utilisateur ou période.

Une fonctionnalité permettant de visualiser le gameplay du serveur en direct, sans avoir à regarder l'écran du serveur, serait bénéfique. Cela pourrait être réalisé via un client léger ou une interface web permettant aux utilisateurs de suivre l'action en temps réel. Cette fonctionnalité améliorerait non seulement l'accessibilité, mais aussi l'engagement des utilisateurs.

L'ajout d'une fonctionnalité permettant d'ajouter plusieurs BOTS en même temps améliorerait considérablement l'efficacité et la flexibilité du système. Cela serait particulièrement utile lors de la configuration initiale ou lors de l'organisation de sessions de jeu impliquant de nombreux participants. Une commande simple pour spécifier le nombre de BOTS à ajouter et leurs paramètres respectifs pourraient être intégrée à l'interface utilisateur.

L'introduction de nouvelles commandes de jeu pourrait enrichir l'expérience des utilisateurs. Par exemple, une commande permettant aux joueurs de connaître la position de leurs alliés en temps réel renforcerait la coordination et la stratégie en équipe. De plus, l'ajout d'un mode "Team" permettrait de structurer les parties en équipes, ajoutant une nouvelle dimension de compétition et de coopération.

Ajouter de nouveaux modes de jeu tels que Match à Mort en Équipe (MME), Search and Destroy (SND), Infecté et Capture the Flag (CTF) offrirait une variété accrue et attirerait une base d'utilisateurs plus large. Chaque mode de jeu pourrait avoir ses propres règles et objectifs, ce qui rendrait le système plus dynamique et intéressant pour les utilisateurs.

Permettre aux utilisateurs de sauvegarder une configuration de serveur par défaut simplifierait grandement le processus de lancement de nouvelles sessions de

jeu. Cette fonctionnalité pourrait inclure des options pour sauvegarder les paramètres de jeu, la liste des BOTS, et d'autres préférences spécifiques, offrant ainsi une expérience utilisateur plus fluide et personnalisée.

IV. Analyse technologique

1. Option de réutilisation de l'existant

Avantages de continuer avec Java et JBox2D

Familiarité et expertise : Les membres de l'équipe sont déjà familiers avec Java, ce qui réduit la courbe d'apprentissage et permet de se concentrer sur les améliorations à apporter plutôt que sur l'apprentissage de nouvelles technologies.

Support et documentation : Java bénéficie d'une vaste communauté de développeurs et de ressources documentaires. JBox2D, en tant que port Java de la célèbre bibliothèque Box2D, possède également une documentation robuste et des exemples d'utilisation variés.
Performance et Efficacité

Performance éprouvée : Java, bien qu'interprété, offre des performances proches des langages compilés grâce à la JVM (Java Virtual Machine) et à des optimisations telles que le JIT (Just-In-Time) compilation.

Gestion de la physique : JBox2D est optimisé pour les simulations physiques en 2D, offrant des fonctionnalités robustes et performantes pour la gestion des interactions entre objets.

Maintenabilité et extensibilité : La base de code actuelle peut être améliorée progressivement, ce qui permet une transition en douceur et évite les interruptions de service. L'architecture orientée objet de Java facilite la modularisation du code, permettant une isolation des fonctionnalités et une extensibilité accrue.

Écosystème et intégration : Java peut s'intégrer facilement avec d'autres technologies grâce à son vaste écosystème de bibliothèques et frameworks. De nombreux outils de développement, de test et de déploiement sont disponibles pour Java, tels que Maven, Gradle, JUnit, et d'autres, facilitant le cycle de vie du développement logiciel.

Inconvénients de continuer avec Java et JBox2D

Performance et scalabilité : Bien que la JVM soit performante, elle peut présenter des limitations en termes de gestion fine des ressources, particulièrement dans des environnements de jeu nécessitant des performances très élevées et une faible latence. Le garbage collector de Java, bien qu'efficace, peut introduire des pauses imprévisibles, ce qui peut être problématique pour les applications de temps réel telles que les jeux.

Dette Technique : Le code hérité peut contenir des implémentations obsolètes ou mal optimisées, ce qui peut nécessiter des efforts significatifs de refactoring pour atteindre les standards de qualité modernes.
Le couplage fort entre certaines classes et la complexité accrue peuvent rendre difficile l'introduction de nouvelles fonctionnalités ou l'amélioration des fonctionnalités existantes.

Limitations de JBox2D : JBox2D, bien que robuste, peut ne pas fournir toutes les fonctionnalités spécifiques nécessaires pour certaines évolutions avancées du jeu, nécessitant des développements supplémentaires ou l'intégration d'autres bibliothèques.

Innovation et adaptabilité : En restant sur Java et JBox2D, l'application pourrait manquer l'opportunité d'intégrer des technologies émergentes qui pourraient offrir des avantages significatifs en termes de performance, de flexibilité et de capacités fonctionnelles.
Les nouvelles technologies, comme les moteurs de jeu plus modernes ou les langages offrant des paradigmes différents (comme Rust pour la sécurité et la performance), pourraient offrir des bénéfices non atteignables avec Java et JBox2D.

Conclusion

Après une analyse approfondie, bien que continuer avec Java et JBox2D présente certains avantages liés à la familiarité et à la modularité, les limitations en termes de performance, de scalabilité et de potentiel d'innovation sont significatives. La complexité et le couplage élevé du code existant posent également des défis importants pour l'ajout de nouvelles fonctionnalités et l'amélioration de l'application.

De plus, pour répondre aux exigences de notre projet d'évolution du serveur de jeu, incluant l'ajout de nombreuses fonctionnalités avancées telles que de nouveaux modes de jeu, des capacités de gestion d'équipes, et des options de personnalisation de véhicules, il devient évident que la réutilisation du code existant en Java et JBox2D pourrait freiner notre progression.

En considérant l'ensemble des facteurs, il est stratégique de réécrire le serveur en utilisant des technologies plus modernes et performantes. Cette réécriture permettra de créer une architecture plus flexible et scalable, d'intégrer des technologies émergentes pour optimiser les performances et d'assurer une plus grande facilité de maintenance et d'extensibilité. Cette approche garantira que notre serveur de jeu peut non seulement répondre aux besoins actuels, mais aussi évoluer de manière fluide pour intégrer de nouvelles fonctionnalités futures.

2. Nouvelles technologies envisagées

Dans le cadre de ce projet, nous envisageons deux principales approches technologiques : l'utilisation de Rust en combinaison avec une bibliothèque comme Macroquad, ou l'adoption de moteurs de jeu bien établis tels que Godot et Unity. Ces deux approches offrent des perspectives distinctes en termes de performance, communauté, et facilité de développement.

Performance : Rust, grâce à sa gestion fine de la mémoire et son contrôle précis des ressources, est reconnu pour ses performances exceptionnelles. En revanche, bien que les moteurs de jeu comme Godot et Unity puissent présenter une légère diminution en termes de gestion des ressources, ils compensent par des outils intégrés optimisant les performances pour des projets de petite à moyenne envergure.

Communauté : Rust bénéficie d'une communauté croissante et dynamique, particulièrement intéressée par la performance et la sécurité. Cependant, les moteurs de jeu comme Godot et Unity ont des communautés beaucoup plus larges et bien établies, offrant une abondance de ressources, de plugins, et de support technique.

Facilité de développement : Rust, bien qu 'offrant un contrôle fin et des performances élevées, peut être complexe à maîtriser et nécessite un effort de développement important. En revanche, les moteurs de jeu comme Godot et Unity simplifient le développement avec des outils prêts à l'emploi, des interfaces intuitives, et un support multi-langage, ce qui réduit considérablement le temps de développement.

Scénario 1 : Projet sur Godot

Langage de programmation : Nous utiliserons GDScript et C# pour bénéficier à la fois de la simplicité de GDScript, conçu pour être intuitif dans l'environnement Godot, et de la puissance de C#, qui offre une performance élevée et une vaste gamme de fonctionnalités.

Architecture de l'application : L'architecture de notre application s'articulera autour du système de scène et de nœud de Godot. Chaque élément du jeu, qu'il s'agisse de personnages, d'objets ou de niveaux, sera représenté par des nœuds, tandis que les scènes structurent ces éléments de manière modulaire et hiérarchique.

Communication réseaux : Pour la communication en direct, Godot offre des solutions intégrées pour la mise en réseau, permettant de créer facilement des jeux multijoueurs. Les protocoles TCP/UDP peuvent être utilisés via GDScript ou C# pour gérer les connexions entre les joueurs de manière efficace.

Développement et tests : Godot facilite le développement et les tests avec son éditeur intégré qui permet de tester et de déboguer en temps réel. Les tests unitaires et fonctionnels peuvent être automatisés à l'aide de diverses bibliothèques de test disponibles pour GDScript, telles que GUT et WAT.

Scénario 2 : Projet en utilisant Rust et Macroquad

Langage de programmation : Nous opterons pour Rust en raison de ses performances élevées et de sa gestion fine de la mémoire, en utilisant Macroquad comme bibliothèque de support pour le rendu graphique, la gestion des entrées utilisateurs, et la physique.

Architecture de l'application : L'architecture de l'application en Rust sera construite autour des modules et des fonctionnalités offertes par Macroquad. La gestion de la mémoire et des ressources sera optimisée pour garantir des performances élevées et une faible latence.

Communication réseaux : Pour la communication en direct, Rust offre des bibliothèques comme Tokio et async-std pour la gestion des connexions réseau asynchrones. Ces bibliothèques permettent de créer des serveurs et des clients robustes pour les jeux multijoueurs en temps réel.

Développement et tests : Le développement en Rust peut être plus complexe, mais offre des outils puissants comme Cargo pour la gestion des dépendances et les tests. Les tests unitaires et d'intégration peuvent être automatisés pour assurer la qualité et la fiabilité du code.

Scénario 3 : Projet sur Unity

Langage de programmation : Nous utiliserons C# pour exploiter la performance et la flexibilité offertes par Unity. C# est parfaitement intégré à Unity, offrant une syntaxe puissante et des fonctionnalités avancées pour le développement de jeux.

Architecture de l'application : L'architecture de l'application en Unity sera organisée autour des GameObjects et des composants. Chaque élément du jeu sera représenté par un GameObject, et les composants seront utilisés pour ajouter des comportements et des fonctionnalités aux objets.

Communication réseaux : Pour ce projet, nous utiliserons FishNet, une bibliothèque réseau puissante et flexible pour Unity. FishNet est conçue pour faciliter le développement de jeux multijoueurs, offrant des fonctionnalités avancées pour la gestion des connexions, la synchronisation des objets, et le contrôle de la latence. Elle permet une gestion efficace des connexions réseau et une synchronisation précise des états entre les joueurs, ce qui est essentiel pour les jeux multijoueurs en temps réel. Grâce à ses API conviviales et son intégration transparente avec Unity, FishNet simplifie la mise en place des fonctionnalités multijoueurs, assurant ainsi une expérience fluide et réactive pour les joueurs.

Développement et tests : Unity propose un environnement de développement complet avec des outils de test intégrés. Les tests unitaires et fonctionnels peuvent être automatisés à l'aide de frameworks comme UTF, permettant de maintenir une haute qualité de code et de déboguer efficacement.

3. Comparatif des scénarios

Critère (note /10)	Scénario 1 (Godot)	Scénario 2 (Rust)	Scénario 3 (Unity)

Couverture fonctionnelle front-office	9	7	8
Couverture fonctionnelle back-office	8	7	8
Technique	9	9	8
Maturité de la technologie	8	8	9
Ouverture	8	8	9
Pérennité	9	7	9
Fiabilité	7	9	8
Nombre d'acteurs	8	6	10
Niveaux de compétences requis	9	8	6
Maîtrise interne	8	7	5
Rapidité de mise en œuvre	8	7	9
Performance	8	9	9
Expérience Utilisateur (UX)	9	7	8
Flexibilité des Fonctionnalités	9	8	9
Intégration avec les Fonctionnalités Natives	9	8	9
Coût de Développement	10	8	7
Maintenance et Évolutivité	9	8	8
Temps de Développement Initial	9	7	8
Réutilisabilité du Code	9	8	8
Sécurité	7	9	7
Écosystème de Développement	9	7	10
Total	188	170	181

4. Conclusion

Après avoir examiné en détail les différents scénarios de développement pour notre projet, il est clair que le choix du bon langage de programmation et du moteur de jeu est essentiel pour atteindre nos objectifs de manière efficace et efficiente. Bien que Java présente certaines qualités, telles que sa polyvalence, pour notre application axée sur les jeux, il est évident que des options plus spécialisées offrent des avantages décisifs.

Le choix entre Rust, Unity et Godot s'est avéré être un exercice minutieux, tenant compte de nombreux critères, allant de la performance à la facilité de développement en passant par la communauté et la pérennité technologique. Alors que Rust offre des performances exceptionnelles et une gestion précise des ressources, son niveau de complexité et les efforts de développement nécessaires le placent dans une catégorie distincte, mieux adaptée aux besoins spécifiques des projets où la performance brute est primordiale et où une expertise technique approfondie est disponible.

D'autre part, Unity, avec son écosystème mature et ses performances solides, reste une option attrayante, notamment pour les jeux nécessitant une intégration étroite avec des fonctionnalités natives et une flexibilité maximale. Cependant, son coût initial et sa dépendance à des plugins tiers pour certaines fonctionnalités peuvent être des inconvénients pour les projets avec des contraintes budgétaires serrées.

C'est dans ce contexte que Godot se distingue comme une solution optimale pour notre projet. Son architecture flexible, sa communauté active offrent une combinaison unique d'avantages : rapidité de développement, coût réduit et puissance fonctionnelle. En outre, sa bibliothèque intégrée pour la gestion des jeux multijoueurs, ENet, représente un avantage significatif pour notre application (UDP), qui inclut des fonctionnalités multijoueurs. Nous pouvons également utiliser les librairies existantes pour le TCP avec Godot C#.

En prenant en compte la maturité technologique, l'expérience utilisateur, l'intégration avec les fonctionnalités natives, la réutilisabilité du code, la performance, l'écosystème de développement et le coût, il est évident que Godot se démarque comme le choix privilégié pour notre projet. Il offre une solution complète et équilibrée, répondant à nos besoins fonctionnels tout en optimisant les aspects techniques, économiques et expérientiels du développement de notre application de jeux. En conséquence, nous sommes confiants que l'adoption de Godot comme plateforme de développement nous permettra d'atteindre nos objectifs de manière efficace et de livrer un produit final de haute qualité.

V. Planification de la réalisation

1. Gestion de projet

Priorisation des améliorations

Lors du développement du projet, nous prioriserons les tâches suivantes : la création d'un journal de logs des commandes effectuées, l'affichage du score pendant la partie, et la mise en place d'un moyen d'enregistrer un modèle de serveur par défaut. Ensuite, si nous avons encore du temps de développement, nous ajouterons des modes de jeu et des éléments de progression.

Nous pensons que ces tâches sont réfléchies en termes d'efficience et de gain de temps de développement, tout en apportant des améliorations significatives au projet. Nous priorisons l'amélioration de l'expérience de développement avec notre serveur avant l'ajout de nouvelles fonctionnalités de gameplay.

Découpage en tâches

Liste des tâches à réaliser pour chaque amélioration proposée. Trello

https://trello.com/invite/b/UnhjO9uV/ATTId35cc3db98d1fa801ef57edbd01dc735B2A7B47E/sae

2. Utilisation de Git

Branche Master

La branche master est la branche principale du dépôt Git. Elle contient la version stable et fonctionnelle de l'application.

La branche master est utilisée pour déployer des versions du logiciel qui ont été testées, validées et considérées comme prêtes pour une utilisation en production. Elle représente la version la plus stable et est souvent liée à des déploiements en production.

Branche Develop

La branche develop est une branche de travail intermédiaire où les fonctionnalités sont intégrées avant d'être fusionnées dans la branche master.

Cette branche permet de rassembler les différentes fonctionnalités développées par les membres de l'équipe. Cela offre un environnement central pour tester l'intégration de nouvelles fonctionnalités avant de les livrer dans la version stable (master).

Branches de Fonctionnalités

Pour chaque nouvelle fonctionnalité ou résolution de ticket, une nouvelle branche de fonctionnalité est créée à partir de la branche develop.

Cette approche permet d'isoler le développement de chaque fonctionnalité, ce qui facilite la gestion des modifications et des retours. Chaque branche de fonctionnalité est indépendante des autres, ce qui évite les conflits potentiels entre les développements simultanés.

Merge Request, Review et Merge dans Develop

Une fois qu'une fonctionnalité est développée et testée dans sa branche de fonctionnalité, une merge request est créée pour fusionner les modifications dans la branche develop. La fusion est précédée d'une revue de code.

Les merge requests offrent un mécanisme structuré pour examiner les modifications, s'assurer de la qualité du code et garantir qu'il répond aux normes de l'équipe. Cela permet également de résoudre les éventuels conflits avant l'intégration dans la branche develop.

Fusion de Develop dans Master (Version Stable)

Une fois que la branche develop contient un ensemble stable de fonctionnalités, elle est fusionnée dans la branche master, marquant une nouvelle version stable de l'application.

Cette approche garantit que seules les fonctionnalités approuvées et testées sont intégrées dans la version stable de l'application. La branche master représente ainsi la version la plus récente et fonctionnelle de l'application.

Conclusion

L'utilisation de Git avec une branche master principale, une branche develop et des branches de fonctionnalités, suivie de merge requests, de revues de code et de fusions sélectives, offre une méthodologie de développement robuste et efficace. Cela permet une gestion structurée des versions, une collaboration transparente au sein de l'équipe, et garantit la stabilité de la version stable de l'application.

VI. Conclusion

1. Synthèse des points clés

Point sur l'existant : Le serveur de jeu actuel est basé sur Java et JBox2D, avec une architecture orientée objet, mais présente des limitations en termes de performance et de scalabilité.

Pas de reprise de Java : Après une analyse détaillée, la décision a été prise de ne pas continuer avec Java et JBox2D en raison des défis posés par la performance et la complexité du code existant.

Choix de Godot : Godot a été choisi comme la nouvelle plateforme de développement en raison de son architecture flexible, de sa communauté active, de son modèle open-source, et de ses capacités intégrées pour la gestion des jeux multijoueurs.

Planification avec Trello : La planification et la gestion de projet seront réalisées en utilisant Trello, permettant une organisation structurée des tâches et une priorisation efficace des améliorations.

2. Perspectives futures :

Intégrer plus de protocoles réseau : Améliorer la flexibilité et la robustesse de la communication réseau en intégrant des protocoles supplémentaires.

Intégrer des inputs clavier, manette : Étendre le support des périphériques d'entrée pour inclure les claviers et les manettes, offrant ainsi une meilleure expérience utilisateur.

Étendre les modes de jeux : Ajouter de nouveaux modes de jeu pour diversifier l'expérience utilisateur et attirer une base d'utilisateurs plus large.

VII. Annexes

Sources pour rust : https://macroquad.rs/

Documentation commande Jeu existant :
https://raievskc.gricad-pages.univ-grenoble-alpes.fr/enseignement/BUTInfo/AndroidBUTInfo/WifiGameController/docs/server.html

Documentation JBox2d :
http://www.jbox2d.org/

Sources pour godot :

Network godot :
https://docs.godotengine.org/en/stable/tutorials/networking/high_level_multiplayer.html
https://docs.godotengine.org/fr/4.x/classes/class_enetconnection.html
https://docs.godotengine.org/fr/4.x/classes/class_streampeertcp.html

Test sur godot
https://medium.com/@JustAnotherRandomGuy/writing-tests-in-godot-a9a5ed279b2c
https://github.com/watplugin/wat
https://github.com/bitwes/Gut

Source pour unity :

Network sur unity :
https://forum.unity.com/threads/free-networking-solutions-comparison-mirror-vs-fishnet-vs-netick.1574884/

Test sur unity :
https://docs.unity3d.com/Packages/com.unity.test-framework@1.4/manual/index.html