Documentation et développement de fonctionnalités pour un simulateur spatial en Rust

Rapport de stage – Licence 2 MPCI

Sarah KEGHIAN

Encadrant : François Brucker
Période : juin – juillet 2025

Table des matières

Remerciements

Je tiens à remercier mon encadrant de stage, François Brucker, enseignant-chercheur au LIS, de m'avoir accordé cette opportunité de stage, ainsi que pour son accompagnement, ses conseils tout au long du stage et pour le temps qu'il a consacré à réfléchir aux pistes d'évolution du projet.

Je remercie également Benjamin Lagosanto, ancien étudiant MPCI, dont le travail réalisé lors de son stage facultatif de L2 a constitué une base solide pour ce projet.

Présentation de l'organisme d'accueil

Le Laboratoire d'Informatique et des Systèmes (LIS) couvre divers domaines de recherche, organisés en quatre pôles : Calcul, Science des données, Analyse et Contrôle des systèmes, ainsi que Signal et Image. Le LIS se consacre à la fois à la recherche fondamentale dans ces domaines et à l'application concrète de ces connaissances dans divers secteurs industriels, notamment le transport, la santé, l'environnement et la défense. J'ai été encadrée par François Brucker de l'équipe ACRO (Algorithmique, Combinatoire et Recherche Opérationnelle) du pôle calcul.

Le LIS est fortement connecté au milieu socio-économique grâce à son implication dans divers pôles de compétitivité comme le Pôle Mer, le Pôle SCS, et le Pôle Eurobiomed. Il est aussi membre de l'institut Carnot STAR et participe à des initiatives nationales telles que l'Institut Langage, Communication et Cerveau (ILCB) et le Centre Turing des Systèmes Vivants (Centuri).

De plus, les chercheurs et enseignants-chercheurs du LIS sont impliqués dans la formation des étudiants à l'informatique, notamment avec la Faculté d'Aix-Marseille et avec l'école Centrale Méditerranée, partenaire du LIS. Bien que ce stage ne soit pas en lien direct avec les recherches de l'équipe ACRO, il fait tout de même partie de cette démarche d'éducation et professionnalisation des étudiants.

1 Introduction

Ce stage reprend le projet débuté par Benjamin Lagosanto pour son stage facultatif de L2, qui avait pour but de poser les bases d'un jeu d'exploration et de commerce spatial en développant des simulations physiques d'objets en mouvement dans le système solaire.

Outils de développement Ce jeu est entièrement codé en Rust [1], reconnu pour sa rapidité d'exécution (de par son caractère bas-niveau) mais aussi pour sa gestion des erreurs, le plus souvent détectées à la compilation. De plus, le code repose sur Bevy [3], un moteur de jeu open-source et minimaliste, sans interface graphique : tout y est fait directement en code. Pour ce qui est de la gestion des sources, nous travaillons avec Git et GitHub [2,4]. Largement utilisés dans l'industrie, ils permettent une passation rapide et efficace du projet aux prochains stagiaires tout en gardant les traces de son avancée.
J'avais déjà acquis une certaine familiarité avec Git et GitHub à travers mes précédents stages et projets. De plus, avant le début de ce stage, j'ai pris le temps d'étudier les bases du langage Rust afin de m'y préparer. En revanche, la découverte et l'apprentissage de Bevy se sont faits progressivement au cours du stage.

Sujet du stage L'apport de mon stage dans ce projet est double. Il s'agit d'une part de faciliter la passation de ce projet complexe aux futurs stagiaires, en expliquant son architecture ainsi que le fonctionnement de Bevy, un moteur qui n'est pas familier aux étudiants. D'autre part, mon stage contribue à l'avancement du jeu par le développement de nouvelles fonctionnalités.

La première partie du rapport porte donc sur l'analyse du code et l'explication de l'architecture du projet en lien avec le fonctionnement de Bevy. La deuxième détaille le développement de la fonctionnalité de planification d'actions pour les vaisseaux. Celle-ci m'a permis de mettre en pratique ma compréhension de Bevy et de son architecture ECS (Entity-Component-System). Enfin, la troisième partie présente la recherche et avancements réalisés sur la gestion des trajectoires des vaisseaux, ainsi que les perspectives d'évolution du projet.

2 Analyse du code existant

L'analyse du code a constitué une part importante de mon stage, en raison de la complexité du projet déjà bien avancé, de l'utilisation d'outils qui m'étaient alors peu familiers, mais aussi de l'importance de cet audit qui permettra aux futurs stagiaires de prendre en main le projet rapidement et efficacement même s'ils n'ont pas les mêmes bases. C'est pourquoi j'ai rédigé un résumé des fonctionnalités principales du moteur de jeu utilisé.

2.1 Bevy et le modèle ECS

Bevy est un moteur de jeu reposant sur le modèle *Entity-Component-System* (ECS), une architecture favorisant la modularité, la flexibilité et la séparation claire des responsabilités dans le code. Ce modèle se prête particulièrement bien à ce projet, où le développement de différentes fonctionnalités est réparti entre plusieurs stagiaires. Comme tout le code du projet s'appuie sur ce modèle, il est essentiel de bien comprendre son fonctionnement pour pouvoir travailler correctement.

L'architecture ECS se compose de trois concepts principaux :

— **Entity (Entité)** : Il s'agit d'un identifiant unique associé à un groupe de composants. Une entité ne contient pas de données en elle-même, mais est un conteneur abstrait auquel on attache des composants.

— **Component (Composant)** : Ce sont des structures de données (structs en Rust) qui détiennent les données associées à une entité. Chaque composant doit implémenter le trait `Component`.

— **System (Système)** : Ce sont des fonctions Rust qui manipulent souvent les composants. Un système peut lire ou modifie l'état des composants pour appliquer des actions, comme gérer la physique ou les comportements du jeu.

En plus des composants, Bevy utilise les **Resources**, des structures globales et uniques qui stockent des données partagées dans tout le jeu. À la différence des composants, les ressources n'appartiennent donc pas à une entité particulière. Pour donner des exemples d'utilisation : on peut stocker le tick de simulation actuel dans une ressource nommée `GameTime`, ou encore avoir une autre ressource appelée `EditorContext`, permettant de stocker l'identifiant vaisseau sélectionné dans l'éditeur. Ces ressources implémentent le trait `Resource`.

Les systèmes sont enregistrés dans l'`App` de Bevy via des *Schedules* (ordonnancements) qui déterminent quand et dans quel ordre ils s'exécutent. Bevy propose plusieurs schedules de base, exécutés en boucle à chaque frame :

— **Startup** : s'exécute une seule fois au démarrage de l'application.

— **PreUpdate**, **Update**, **PostUpdate** : s'exécutent chaque frame, dans cet ordre, permettant de structurer les différentes phases de traitement (par exemple, logique du jeu, interface utilisateur, etc.).

Un schedule particulier, le **FixedUpdate**, s'exécute à intervalles réguliers indépendamment du taux de rafraîchissement de la machine, ce qui est utile pour les systèmes liés à la logique du jeu, comme le mouvement ou la physique des objets.

Il est aussi possible de créer des *sets*, (ou ensembles) de systèmes dans les schedules pour mieux organiser le code. Par exemple, un set `OrbitsUpdate` peut regrouper les systèmes et ressources mettant à jour le mouvement des objets orbitaux dans le `FixedUpdate`. Tandis que `ObjectsUpdate` met à jour l'état des objets selon les interactions du joueur dans `Update`. De plus, on peut également choisir de l'ordre d'exécution des systèmes ou des sets au sein d'un même schedule grâce à la méthode `.chain()`.

Pour la communication entre systèmes, Bevy propose un système d'**Events** (événements) :

— Un système utilisant `EventWriter` envoie des événements, par exemple lorsque la touche "z" est pressée, l'évènement `PlayerMovement::Up` est envoyé.

— Un système utilisant `EventReader` réagit à ces événements, par exemple, un système reçoit l'évènement `PlayerMovement::Forward` et ajoute 5 à la position x, du joueur pour le déplacer.

De plus, pour accéder aux données des entités, les systèmes utilisent des **Query**, qui spécifient quels composants doivent être présents sur les entités retournées, permettant un traitement ciblé et efficace.

Enfin, Bevy est conçu pour être modulaire grâce aux **Plugins**, qui permettent de regrouper des systèmes et leur ordonnancement ainsi que des ressources ayant un lien logique. Par exemple, un plugin physique (`PhysicsPlugin`) peut regrouper tous les systèmes (comme le set `OrbitsUpdate`) et ressources liés à la gestion de la physique, et être intégré dans l'`App` principale.

Maintenant que les bases de Bevy sont posées, nous pouvons nous pencher sur l'architecture du projet.

2.2 Architecture générale du projet

L'architecture du projet est construite autour d'un regroupement par fonctionnalités, plus que strictement par type Bevy (composants, systèmes, etc.). Chaque fonctionnalité ou sous-système du jeu dispose de son propre dossier, dans lequel sont regroupés ses éléments spécifiques : composants, systèmes Bevy, ressources et événements. Cette organisation découle naturellement du modèle ECS (Entity-Component-System) utilisé par Bevy, et favorise la séparation des responsabilités ainsi que la modularité du code.

Le point d'entrée de l'application est le fichier `src/bin/client.rs`, qui initialise une instance Bevy et enregistre les plugins `ClientPlugin`, `TuiPlugin` et `GuiPlugin` (fig. 1). Ces plugins constituent les briques principales de l'application côté client.

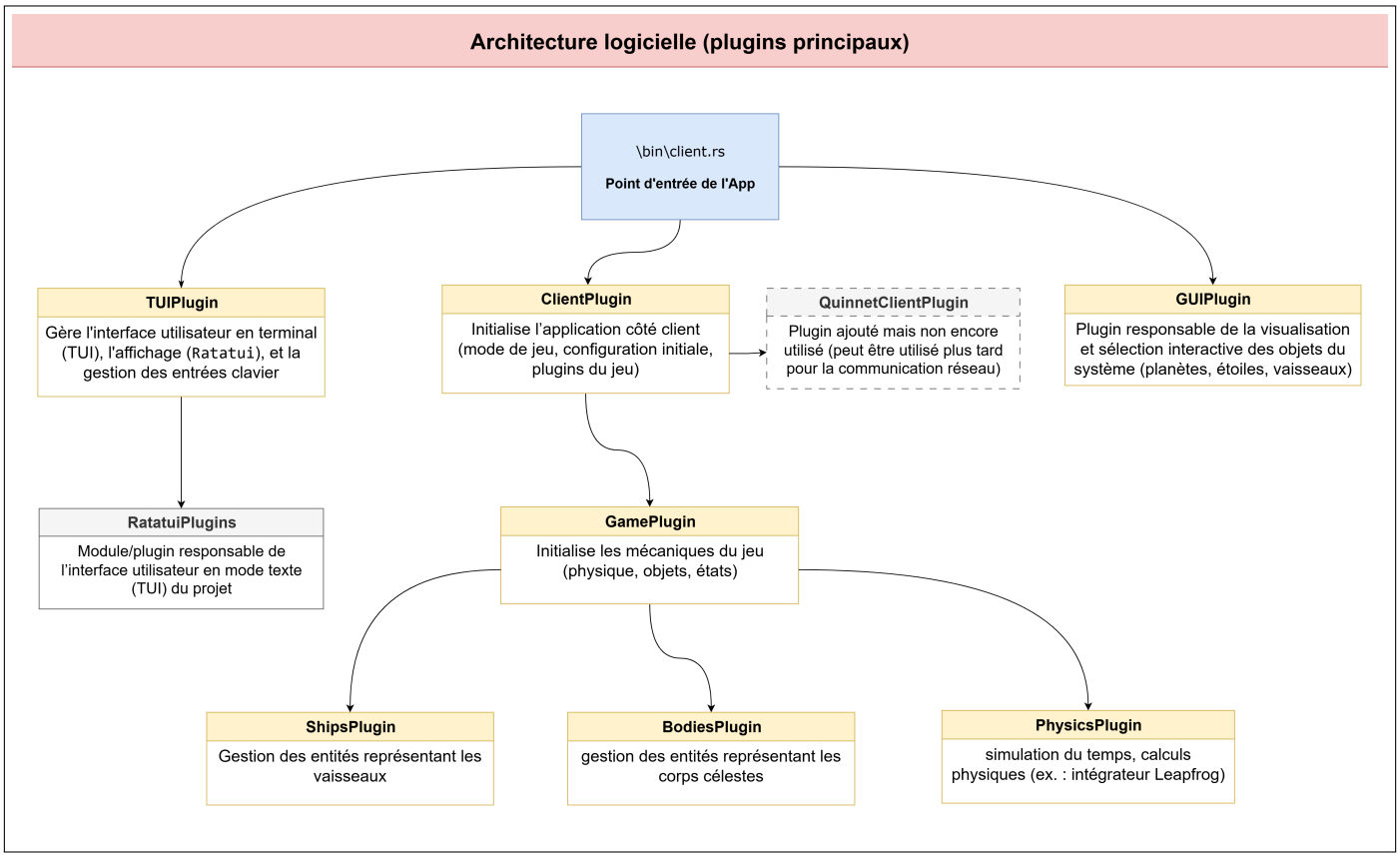

FIGURE 1 – Architecture des plugins principaux du projet

— `ClientPlugin` configure l'environnement de jeu en fonction du mode (solo, multijoueur, exploration). Actuellement, seul le mode solo est disponible.

— `TuiPlugin` initialise l'interface utilisateur du terminal à l'aide de la bibliothèque `bevy_ratatui`.
— `GuiPlugin` gère l'interface graphique du jeu sur lequel on peut éditer les trajectoires.

Le plugin `ClientPlugin` enregistre également le plugin central du jeu : `GamePlugin`. Ce dernier regroupe les systèmes principaux liés au cœur de la simulation via plusieurs sous-plugins thématiques :

— `PhysicsPlugin` : regroupe la simulation temporelle et physique du jeu (orbites, champs gravitationnels, intégration numérique, etc.).
— `BodiesPlugin` : prend en charge la création et le comportement des corps célestes (étoiles, planètes, lunes...).
— `ShipsPlugin` : de même pour le comportement des vaisseaux.

Cette organisation modulaire permet d'assurer une bonne évolutivité du projet. Chaque fonctionnalité est isolée dans son propre plugin, ce qui facilite les tests et l'ajout de nouveaux comportements.

De plus, cette architecture fondée sur les plugins se reflète également dans la structure des dossiers du projet. Le découpage du code source suit une organisation par domaines fonctionnels, où chaque dossier regroupe les éléments (systèmes, composants, ressources) nécessaires à une fonctionnalité donnée (voir figure 2). Cela permet de maintenir une cohérence entre la structure logique du code et sa structure physique sur disque.

Organisation des dossiers Chaque sous-dossier correspond à une sous-partie du jeu, généralement structurée autour d'un plugin. On y retrouve les composants, systèmes Bevy, ressources et événements propres à cette fonctionnalité. Cette séparation permet une meilleure lisibilité du code et facilite l'évolution du projet.

```
src/
├── bin/
│   └── client.rs        → Point d'entrée
├── objects/    → Entités du jeu
│   ├── ships.rs         → ShipsPlugin
│   ├── bodies.rs        → BodiesPlugin
│   ├── bodies/
│   └── ships/
│       ├── trajectories.rs
│       └── ...
├── physics/
│   ├── leapfrog.rs      → Intégration numérique
│   └── ...
├── ui/
│   ├── gui/           → Interface graphique
│   ├── screen/        → gestion des écrans du tui (editor, explorer, etc.)
│   └── ...
├── game.rs              → GamePlugin
├── physics.rs           → PhysicsPlugin
└── ...
```

FIGURE 2 – Organisation physique partielle des dossiers dans `src/` avec l'association des plugins Bevy correspondants

Par exemple, le dossier `objects/` contient les éléments liés aux entités du jeu. On y trouve notamment :

- `objects/ships/` : contient les fichiers liés à la gestion des vaisseaux.
- `objects/ships/trajectories.rs` : définit les systèmes lisant les modifications de trajectoire et qui, en conséquence, mettent à jour la vitesse des vaisseaux à chaque `FixedUpdate`.
- `objects/ships.rs` : contient les systèmes de gestion des événements comme la création d'un nouveau vaisseau, et les fonctions associées à l'instanciation ou modifications des entités vaisseaux dans le monde Bevy.

D'autres dossiers suivent la même logique, par exemple :

- `physics/` : contient les modules liés à la simulation physique (orbites, forces d'influence, intégration numérique...).
- `ui/` : regroupe les fichiers nécessaires à l'affichage pour l'interface utilisateur, en mode graphique et terminal.

Ce découpage permet une organisation claire, évolutive et cohérente avec le modèle de plugin Bevy : chaque fonctionnalité majeure dispose de son espace propre, tout en interagissant avec les autres via des ressources et événements partagés.

Maintenant que l'architecture du projet est claire, je vais présenter le travail effectué sur les fonctionnalités du jeu.

3 Développement du planificateur d'actions

Dans un jeu de type 4X, une partie évolue généralement de la manière suivante : le joueur commence avec peu de ressources et doit donc prévoir manuellement le trajet et les actions de ses vaisseaux afin de limiter la consommation de ces ressources. Au fur et à mesure de l'avancement de la partie, le joueur dispose de plus de ressources et de vaisseaux, et cherche alors à automatiser leur comportement, car il devient difficile de tous les suivre simultanément, surtout sur une durée prolongée. C'est à ce besoin que répond la fonctionnalité du planificateur d'actions (scheduler).

Cette nouvelle fonctionnalité doit permettre de prévoir les actions d'un vaisseau sur le long terme et de les déclencher automatiquement le moment venu. Par exemple, on doit pouvoir ordonner au vaisseau 1 d'activer une poussée de vecteur $(30, 30, 0)$ au tick 515 (un tick correspond à une unité de temps), puis d'activer son radar pour analyser son environnement au tick 1024. Cette deuxième action n'existe pas encore, mais le planificateur doit être conçu de manière à pouvoir intégrer facilement de nouvelles actions dans les futures versions du jeu.

Dans le cadre de mon stage, je n'ai pas travaillé sur l'interface graphique qui permet l'ajout des actions dans le planificateur, mais bien sur la mise en place du fonctionnement interne du planificateur.

Pour commencer, il était nécessaire de créer un fichier afin de bien séparer les responsabilités, puisque cette fonctionnalité est nouvelle. C'est dans le dossier `ships` que j'ai placé ce fichier `scheduler.rs`, car ce planificateur est un outil directement lié au comportement des vaisseaux.

La fonctionnalité repose sur une idée simple : une liste d'actions à effectuer, propre à chaque vaisseau. Il faut donc à la fois stocker cette liste, avec les ticks auxquels les actions doivent être déclenchées, et lier cette liste à un vaisseau en particulier. Pour cela, j'ai défini un struct `ShipSchedule`, contenant un vecteur de tuples de la forme (tick, type d'action). Comme ce struct est ajouté en tant que composant aux entités vaisseaux, il n'est pas nécessaire de stocker l'identifiant associé car on y accède grâce au composant `ShipInfo` du vaisseau.

Pour ce qui est de la gestion des actions, elle doit rester efficace même si un grand nombre de vaisseaux et d'actions sont présents simultanément dans la partie. Comme les ticks sont traités à une fréquence élevée, il est important d'éviter tout coût ou parcours inutile à chaque mise à jour.

Dans un premier temps, j'ai initialement envisagé d'utiliser un type générique paramétré sur l'action, par exemple ShipSchedule<T> où T représenterait un type concret d'action. Mais cette approche aurait imposé qu'un même planning ne contienne qu'un seul type d'action, et donc à maintenir des listes séparées et à dupliquer une partie de la logique de traitement. Cela aurait alourdi la conception et rendu plus difficile l'ajout de nouvelles actions. J'ai donc opté pour un enum ShipActionKind, qui recense l'ensemble des actions planifiables, avec les données nécessaires à leur exécution (par exemple un nœud de manœuvre pour `AddNode`). Pour l'instant, seule l'action `AddNode` est implémentée, mais cette énumération centralise toutes les actions possibles dans une même structure, ce qui facilite leur ajout futur et rend le code plus lisible.

L'activation des actions est déclenchée par un système Bevy `handle_schedules`, qui, à chaque tick du jeu, parcourt tous les `ShipSchedule` via une Query Bevy [6] et vérifie si des actions doivent être activées. Si c'est le cas, l'action est retirée de la liste, puis transmise à une fonction `convert_kind` qui transforme la donnée du type `ShipActionKind` en un événement Bevy concret, comme `TrajectoryEvent::AddNode` afin de l'envoyer. Cet évènement sera ensuite traité par un système `EventReader` dédié, qui existe déjà dans la partie du code lié à la trajectoire des vaisseaux.

Pour ajouter une action planifiable, il suffit alors de deux étapes :
- Ajouter un `ShipActionKind` correspondant à la nouvelle action dans l'enum.
- Ajouter une branche au match de la fonction `convert_kind`, qui envoie l'évènement de la nouvelle action lorsque que le `ShipActionKind` correspond.

Ensuite, il restait à créer l'évènement indiquant l'ajout d'une action ainsi que le système qui écoute cet évènement et ajoute l'action dans le planificateur du bon vaisseau.

Concernant l'ajout du composant `ShipSchedule`, j'avais choisi à l'origine d'ajouter dynamiquement un composant `ShipSchedule` aux vaisseaux uniquement lorsque le joueur exprimait l'intention de planifier des actions. Cette opération était déclenchée par un événement `CreateSchedule`, capté par un système Bevy dédié. L'idée derrière ce choix était d'éviter d'attacher inutilement un composant à un vaisseau si celui-ci ne devait jamais être planifié. Cependant, dans Bevy, le moteur organise les entités en groupes appelés archétypes, en fonction des composants qu'elles possèdent. Deux vaisseaux identiques, dont l'un possède un `ShipSchedule` et l'autre non, appartiennent donc à deux archétypes différents. D'une part, cela peut rendre certaines opérations (comme parcourir tous les vaisseaux ou leur appliquer un traitement commun) un peu plus lourdes ou plus complexes à écrire. D'autre part, cela crée deux archétypes distincts pour des objets de même nature, ce qui n'est pas idéal d'un point de vue conceptuel.

Pour cette raison, j'ai décidé d'ajouter un `ShipSchedule` vide directement lors de la création de chaque vaisseau. Cela évite un système d'événements supplémentaire, et garantit que tous les vaisseaux sont planifiables dès le départ. Ce choix rend le code plus lisible et plus robuste, sans coût notable en mémoire ou en performance.

Enfin, j'ai également implémenté la gestion d'une touche clavier (`"s"`) qui permet de basculer l'affichage de l'application vers l'écran du planificateur d'actions d'un vaisseau. Cela signifie que les futurs développeurs de l'interface graphique ou tex-

tuelle pourront se concentrer uniquement sur le fichier
`screen/scheduler_screen.rs` pour coder le terminal utilisateur (TUI) et le système envoyant l'évènement `AddAction`, sans avoir à modifier la logique sous-jacente de gestion du planificateur.

Ainsi, la mise en place de cette fonctionnalité m'a permis de mieux comprendre les principes fondamentaux de Bevy, comme le système ECS, les événements, et les archétypes, et d'appliquer ces concepts dans un contexte concret.

4 Recherches sur la gestion des trajectoires

Dans la suite de mon travail, je me suis interrogée sur les différences entre les trajectoires des planètes et celles des vaisseaux. Les premières sont prévisibles et déterminées à l'avance car elles obéissent aux lois gravitationnelles et ne peuvent pas être modifiées. Tandis que les secondes peuvent être modifiées dynamiquement grâce à nœuds de manœuvres, qui permettent d'effectuer des poussées en fonction des besoins et des envies du joueur.

C'est pourquoi, dans le jeu tel que je l'ai repris, les trajectoires des vaisseaux sont calculées en résolvant numériquement les équations différentielles du mouvement à l'aide de la méthode *Leapfrog* [7]. Celle-ci est particulièrement adaptée pour simuler des déplacements soumis à des forces gravitationnelles et à des variations de poussée, car il s'agit d'un intégrateur symplectique. Ce type d'intégrateur conserve l'énergie et la stabilité des orbites sur le long terme, tout en restant efficace numériquement. Il permet ainsi de modéliser correctement l'évolution des vaisseaux, même lorsque des manœuvres viennent modifier leur trajectoire.

Les trajectoires des planètes, quant à elles, ne sont pas intégrées numériquement à chaque pas de temps, mais déduites à partir de leurs éléments orbitaux [9] (excentricité, demi-grand axe, inclinaison, etc.). Ceux-ci sont recensés dans la base de donnée du jeu, tirée de "l'Open Data du système solaire" [8]. Le mouvement des planètes est calculé grâce au modèle analytique basé sur les lois de Kepler et l'équation de Kepler. En effet, à partir de ces paramètres, la position et la vitesse sont recalculées en fonction du temps, ce qui permet d'obtenir des orbites stables et déterministes tout en réduisant la charge de calcul.

Cependant, une situation fréquente et désirable dans le jeu consiste à placer un vaisseau en orbite stable autour d'un corps céleste, par exemple pour récupérer des ressources ou réaliser d'autres opérations. Dans une telle situation, la trajectoire du vaisseau présente alors de fortes similitudes avec celle d'une planète ou d'une lune : les deux suivent une orbite prédictible autour d'un astre.

Cela amène naturellement une question : qu'est-ce qui distingue réellement un vaisseau en orbite stable d'un corps céleste ? Et ne pourrait-on pas calculer la trajectoire orbitale du vaisseau à l'aide du même modèle képlérien que celui utilisé pour les planètes ?

En poussant l'idée plus loin, on pourrait même considérer la possibilité d'envoyer des vaisseaux en orbite autour d'autres vaisseaux eux-même en orbite stable.

C'est sur ces réflexions que j'ai orienté la suite de mon travail.

4.1 Généralisation des composants pour la gestion des entités orbitales

Pour réfléchir correctement à ces idées, il faut d'abord comprendre comment les entités « vaisseaux » et « corps célestes » sont définies dans le code.

La figure 3 présente les composants qui composent ces entités, en excluant ceux utilisés uniquement pour l'affichage.

Ship	**Body**
ShipInfo : - id: ShipID - spawn_pos: DVec3 - spawn_speed: DVec3	**BodyInfo** (BodyData)
	Mass(f64)
Acceleration : - current: DVec3 - previous: DVec3	**EllipticalOrbit :** - eccentricity: f64 - semimajor_axis: f64 - inclination: f64 - long_asc_node: f64 - arg_periapsis: f64 - initial_mean_anomaly: f64 - revolution_period: f64 - ...
Influenced : - main_influencer: Option<Entity> - influencers: Vec<Entity>	
Position (DVec3)	**HillRadius** (f64)
Velocity (DVec3)	**Position** (DVec3)
ShipSchedule (Vec<(u64,ShipActionKind)>)	**Velocity** (DVec3)

FIGURE 3 – Liste des composants des entités vaisseaux et corps célestes

Dans le code, on repère les composants qui différencient le traitement du mouvement :

- Le mouvement **képlérien** utilise les composants `Position`, `Velocity`, `EllipticalOrbit` et `BodyInfo`.
- Le mouvement **calculé numériquement** utilise `Position`, `Velocity`, `Acceleration` et `Influenced`.

Avec le fonctionnement actuel, pour qu'un vaisseau soit traité comme une planète, il faut lui ajouter :

1. un composant `EllipticalOrbit`, obtenu en calculant ses éléments orbitaux [9],
2. un composant `BodyInfo`.

Ce dernier est spécifique aux planètes, car il contient une structure `BodyData` qui regroupe des informations issues de la base de données, comme le rayon moyen du corps.

En regardant le code de plus près, on constate pourtant que la seule donnée réellement utilisée dans `BodyData` est le champ `orbiting_bodies`, qui recense les objets enfants (lunes, satellites, etc.). Ce champ est utilisé pour la mise à jour des positions globales, effectuée lors du parcours en largeur de l'arbre des orbites. À chaque étape, la position globale d'un objet enfant est calculée en ajoutant sa position relative à celle de son parent.

Pour permettre à un vaisseau de voir sa position mise à jour comme celle d'un corps céleste, un *refactoring* est nécessaire : on extrait le champ `orbiting_bodies` et le transforme en un composant `OrbitingObjects`. Ce composant peut être attaché

aussi bien à un corps céleste qu'à un vaisseau, et contenir des corps ou bien des vaisseaux.

Ajouter un composant à une entité modifie son *archetype* dans l'ECS, ce qui implique de déplacer l'entité vers un nouvel emplacement mémoire correspondant à la nouvelle combinaison de composants. Dans ce cas, c'est cohérent car le nouvel *archetype* correspond à un nouveau type d'entité, une sorte de « vaisseau-station ». De plus, ce processus peut être relativement coûteux en termes de performance, mais ce n'est pas un problème ici. En effet, contrairement à la mise à jour de position, ces ajouts n'interviennent pas à chaque tick, mais plutôt rarement, lorsqu'un vaisseau arrive en orbite.

Pour ce nouveau composant `OrbitingObjects`, j'ai créé l'énumération `OrbitalObjID`, permettant de référencer aussi bien un corps céleste (`BodyID`) qu'un vaisseau (`ShipID`), et le nouveau composant `OrbitingObjects` stocke une liste de ces `OrbitalObjID`. De plus, j'ai implémenté le trait `From<&MainBodyData>`, pour permettre de convertir automatiquement les données existantes (`orbiting_bodies`) en ce nouveau format, afin de conserver la compatibilité avec les corps célestes.
Ce changement rend la gestion des orbites plus générique et modulable.

Pour utiliser ce changement, un *refactoring* était nécessaire pour adapter le code afin de prendre en compte ces deux types d'objets possibles dans les traitements, alors qu'un seul n'était considéré précédemment. On pouvait le résoudre notamment en effectuant des match selon qu'il s'agissait d'un corps céleste ou d'un vaisseau, mais ce ne sont pas les seuls problèmes qui se sont posés.
Ce travail m'a fait comprendre à quel point un changement à première vue simple pouvait en réalité impliquer des ajustements à de très nombreux endroits du code. Notamment avec des effets en cascade qui impactent des systèmes ou des fichiers qui n'avaient pas l'air impactés au départ.
Ce travail de généralisation des composants rend le code plus modulaire et plus souple, en permettant de traiter de manière uniforme différents types d'objets du jeu.

4.2 Gestion des transitions entre mouvement libre et orbite elliptique

Dans le cadre de la gestion des vaisseaux, j'ai ajouté deux événements internes (`ShipEvents`) : `SwitchToFreeMotion` et `SwitchToOrbital`. Ces événements sont internes car ils ne peuvent pas être déclenchés directement par le joueur, ils sont générés et utilisés automatiquement par l'application lorsqu'un vaisseau doit changer le type de calcul du mouvement. À leur déclenchement, leur rôle est d'ajouter et retirer les composants d'une entité afin de changer son *archetype*, pour passer d'un vaisseau « classique » à un vaisseau-station, ou inversement. Initialement, j'avais codé cela dans la fonction de traitement des événements liés aux vaisseaux (`handle_ship_events`).
Cependant, au fur et à mesure, je me suis aperçue que la lisibilité de ce fichier se dégradait. La fonction grossissait rapidement : près de 90 lignes de code, de longs `match` sur les `ShipEvents`, et la présence de logiques qui n'avaient finalement pas grand-chose à voir les unes avec les autres. Cela rendait aussi les signatures de fonctions plus lourdes, avec de nombreux arguments pas toujours utiles selon le cas.

C'est pourquoi j'ai réfléchi à un *refactoring*. Même si cela prend du temps, ce temps est largement compensé par la lisibilité et la maintenabilité à long terme.

Ma première idée a été de déplacer le contenu des différentes branches dans des fonctions dédiées. Mais rapidement, je me suis dit que cette approche ne faisait que déplacer le problème : la fonction principale resterait trop lourde et continuerait à recevoir trop d'arguments.

J'ai donc envisagé une solution plus complète : utiliser des systèmes séparés pour chaque type d'événement `ShipEvents`. Avant de commencer, j'ai pesé les avantages et les inconvénients de cette solution.

Avantages :
- Clarté et lisibilité : chaque système gère une seule responsabilité.
- Maintenance facilitée : corriger ou modifier un comportement précis ne risque pas d'impacter les autres.
- Meilleure testabilité : chaque système peut être testé indépendamment.
- Gestion des paramètres simplifiée : un système ne reçoit que ce qui est pertinent pour l'événement qu'il traite.
- Parallélisation possible : Bevy peut exécuter plusieurs petits systèmes en parallèle s'ils n'ont pas de conflits.

Inconvénients :
- Plus de systèmes à gérer dans le même ordonnancement.
- Duplication de la lecture de l'`EventReader<ShipEvents>` dans plusieurs systèmes.
- Léger surcoût : chaque système doit parcourir tous les événements `ShipEvents` et filtrer ceux qui le concernent.

Finalement, ces inconvénients sont mineurs : la lecture d'un `EventReader` est très peu coûteuse, le nombre d'événements par boucle reste modéré, et Bevy est conçu pour gérer efficacement des centaines de systèmes. La clarté et la modularité du code priment largement sur ce faible coût supplémentaire, et en cas de problème de performance, il sera toujours possible de cibler les parties réellement les plus coûteuses.

Au final, c'était un raisonnement assez classique : repérer une fonction qui enfreint le principe de responsabilité unique, réfléchir à plusieurs manières de la découper, et choisir celle qui rend le code plus clair et facile à maintenir, sans impacter les performances. Ce n'était pas particulièrement difficile à mettre en place, mais cela a demandé un certain temps de réflexion.

Après avoir mis en place cette structure plus claire, il reste à déterminer à quel moment envoyer ces évènements pour gérer les transformations des vaisseaux. Pour `SwitchToFreeMotion`, la mise en œuvre reste relativement simple. Dans le système qui suit les trajectoires des vaisseaux, il suffit de vérifier si un vaisseau est une station lorsqu'un nœud de manœuvre est atteint. Si c'est le cas, on déclenche l'événement `SwitchToFreeMotion`, qui retire les composants liés à l'orbite et ajoute ceux nécessaires au mouvement libre, comme `Acceleration` et `Influenced`, puis on traite la poussée.

En revanche, savoir quand déclencher `SwitchToOrbital` est plus délicat. Cela soulève une question physique : à partir de quel instant peut-on considérer qu'un corps est réellement en orbite stable ? Et, côté informatique, comment le vérifier dans le contexte du jeu ?

4.2.1 Détection de l'entrée en orbite via l'énergie spécifique

Nous pouvons trouver réponse à cette question en réfléchissant à l'énergie.

En effet, l'énergie potentielle de gravitation est définie comme suit (équation 1).

$$E_p = -\mathcal{G}\frac{mm_0}{r} \tag{1}$$

Avec m_0 la masse de l'attracteur, m la masse de l'objet céleste, r la distance entre eux et $\mathcal{G}$ la constante de gravitation universelle. D'après sa formule, l'énergie potentielle est croissante par rapport à r, c'est-à-dire que l'énergie potentielle est plus élevée lorsque l'objet est loin du centre attracteur.

Quant à l'énergie mécanique (conservée lorsque seule la force de gravitation est en jeu), sa formule est la suivante (équation 2).

$$E_m = E_c + E_p = \frac{1}{2}mv^2 - \mathcal{G}\frac{mm_0}{r} \tag{2}$$

.

De plus, on sait qu'une énergie mécanique strictement négative implique qu'un objet est dans un état lié (c'est-à-dire qu'il ne peut s'éloigner qu'à une distance finie du centre attracteur).

En effet, supposons par l'absurde que ce n'est pas le cas. L'objet peut s'éloigner à une distance infinie c'est-à-dire $r = \infty$. Cela implique $E_p = 0$, or comme l'énergie cinétique est toujours positive, on aurait $E_m \geq 0$, ce qui contredit l'hypothèse. Cela nous donne donc un critère nous permettant de distinguer lorsqu'un vaisseau est en orbite.

Cependant, dans la simulation, nous ne connaissons pas à priori la masse des vaisseaux. Nous n'avons donc à disposition que leur position et leur vitesse, c'est pourquoi on ne peut pas appliquer la formule de l'énergie mécanique telle quelle. C'est là qu'intervient l'énergie spécifique.

L'énergie spécifique est définie comme l'énergie mécanique par unité de masse :

$$\varepsilon = \frac{E_m}{m} = \frac{v^2}{2} - \frac{\mathcal{G}m_0}{r}, \tag{3}$$

où v est la vitesse du corps et r sa distance au centre attracteur. Cette grandeur ne dépend donc plus de la masse de l'objet testé et elle conserve la même propriété que l'énergie mécanique : si $\epsilon < 0$, le corps est lié au centre attracteur et suit une orbite fermée. Si $\epsilon \geq 0$, il peut s'échapper vers l'infini. En effet, comme $m > 0$, diviser $E_m < 0$ par m ne change pas le signe. Nous obtenons donc un critère pour assurer qu'un objet est en orbite fermée et qui n'utilise pas la masse du vaisseau.

Toutefois, il faut connaître la masse du corps attracteur, donc pour trouver ce corps sans tous les essayer un par un, nous prenons l'influenceur principal du vaisseau (ou `main_influencer`). Celui-ci est calculé et enregistré dans le composant `Influenced` à chaque tick de simulation. L'influenceur principal est le corps dont la sphère de Hill contient l'objet et qui a le rayon le plus petit parmi ces corps. Dans la simulation, nous utilisons les *sphères de Hill* pour éviter de devoir résoudre un problème à n corps en le simplifiant en un problème à deux corps. La sphère de Hill d'un corps en orbite autour d'un autre corps est approximativement donnée par la formule :

$$R_H \approx a(1-e)\left(\frac{m}{3(M+m)}\right)^{1/3}$$

où :

— a est le demi-grand axe de l'orbite,
— e est l'excentricité de l'orbite,
— m est la masse du corps considéré,
— M est la masse du corps central.

Cette sphère représente la région autour du corps dans laquelle sa gravitation domine par rapport à celle du corps central (ici le soleil), et permet ainsi de déterminer quel corps influence principalement le mouvement de l'objet.

Pour la partie informatique, il reste à écrire 2 fonctions. La première (`check_ship_orbits`), s'exécute à chaque tick dans le PostUpdate pour afin de prendre en compte les changements de position et vitesse du tick en cours. Celle-ci, parcourt l'ensemble des vaisseaux non-orbitaux pour calculer leur énergie spécifique. Si elle est négative, le système envoie l'évènement `SwitchToOrbital` pour le vaisseau concerné.

La seconde, appelée par le système `handle_switch_to_orbital`, calcule les éléments orbitaux permettant de décrire l'orbite du vaisseau à partir de sa position, de sa vitesse et de la masse du corps hôte puis les enregistre dans un composant `EllipticalOrbit` (voir figure 4).

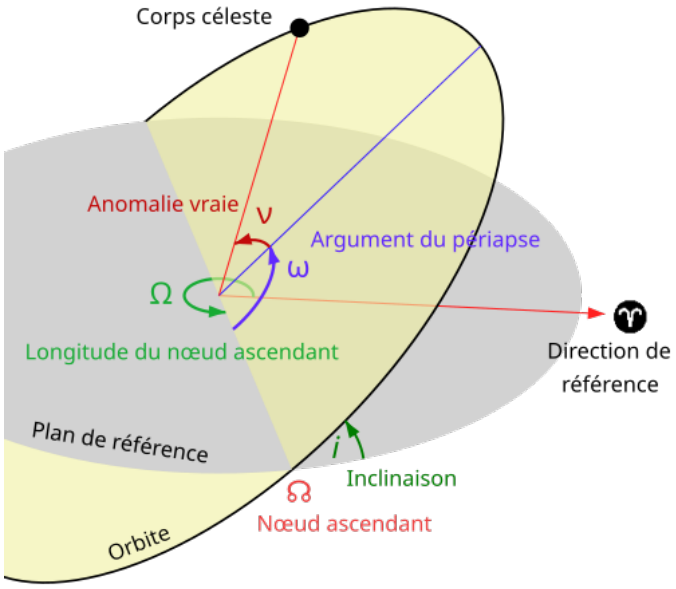

FIGURE 4 – Schéma des éléments orbitaux

4.2.2 Problèmes rencontrés et pistes d'amélioration

Cependant lors des premiers tests, le système ne se comporte pas entièrement comme attendu. Lorsqu'un vaisseau est initialisé avec une position et une vitesse correspondant à une orbite autour d'un corps, le mécanisme fonctionne correctement : le vaisseau est bien détecté comme orbital et l'ellipse correspondante est calculée et affichée. Grâce à la modification du fichier gui.rs, les ellipse apparaissent et les vaisseaux orbitaux apparaissent en rouge.

Cependant, dès le lancement de la progression du temps, c'est-à-dire dès le premier calcul de position à l'aide des formules de Kepler, le vaisseau se "téléporte" à une autre position sur l'ellipse. Cela ne se produit qu'au lancement initial du temps, et pas lorsqu'on le met en pause et qu'on le fait reprendre. Cette anomalie est donc

probablement liée au calcul initial de l'anomalie vraie. L'anomalie vraie est l'angle qui décrit la position du vaisseau sur l'orbite à un instant t (voir figure 4). Le stage touchant à sa fin au moment de ces tests, je n'ai pas eu l'opportunité d'approfondir ma compréhension de la théorie de la mécanique céleste et des calculs de position orbitale. Cette erreur n'a donc pas pu être corrigée.

Un autre point à considérer concerne l'application d'une poussée pour modifier l'orbite, par exemple afin d'augmenter le périapse de l'ellipse. Dans ce cas, le comportement devient problématique. L'intégrateur Leapfrog est correctement utilisé pendant un tick, mais la fonction `check_ship_orbits` détecte immédiatement que les nouvelles conditions de position et de vitesse correspondent à une orbite fermée et déclenche aussitôt l'évènement `SwitchToOrbital`. Le vaisseau est alors projeté instantanément sur la nouvelle ellipse calculée, sans avoir suivi la trajectoire physique qui devait normalement le conduire à cette orbite. Visuellement, cela se traduit par une "téléportation" du vaisseau, qui ne reflète pas l'évolution réelle attendue.

Pour corriger ce comportement, il serait nécessaire d'introduire une temporisation de l'envoi de l'évènement. Concrètement, on pourrait :

- créer un nouvel évènement `TransitionToOrbital` qui marque le début de la phase de transition après l'application d'une poussée
- modifier `check_ship_orbits` pour qu'au lieu de déclencher immédiatement `SwitchToOrbital`, il envoie d'abord `TransitionToOrbital` et démarre un suivi de position du vaisseau
- ne déclencher `SwitchToOrbital` qu'une fois que le vaisseau a parcouru un segment stable et cohérent de sa nouvelle orbite, ou qu'un délai minimal (`MIN_TRANSITION_TIME`) est écoulé
- stocker l'état de transition en tant que composant du vaisseau, de sorte que `check_ship_orbits` sache si le vaisseau est déjà en phase de transition et ne réapplique pas l'évènement prématurément.

Cette approche permettrait de conserver la cohérence physique de la trajectoire, d'éviter les téléportations visuelles, tout en maintenant l'intégration Leapfrog pour la simulation du mouvement orbital.

Toutefois, je n'ai pas eu le temps de mettre en œuvre cette fonctionnalité au cours de mon stage. L'idée reste donc théorique et pourrait être explorée dans le futur.

5 Conclusion

Pour conclure, mon stage m'a permis de travailler sur plusieurs aspects du projet et de poser des bases solides pour les futurs contributeurs. J'ai documenté l'architecture du projet et sa prise en main dans une documentation détaillée (disponible dans le dossier `docs` du GitHub [10]), afin de faciliter la compréhension pour les prochaines personnes intervenant sur le code. Bien que je n'aie pas pu mener à terme certaines idées pendant le stage, j'ai veillé à ne laisser sur le dépôt GitHub que du code propre et fonctionnel, accompagné de guides et d'instructions dans la documentation et les issues.

J'ai également contribué directement à l'avancement de l'application en ajoutant une fonctionnalité de planificateur d'actions pour les vaisseaux. Parallèlement, j'ai mené des travaux de recherche, d'essais et de refactoring pour poser les bases d'une fonctionnalité de calcul analytique du mouvement des vaisseaux orbitaux.

Par ailleurs, bien que ce point n'ait pas été développé dans le rapport pour des raisons de cohérence et de concision, je me suis intéressée à la création de Mesh en Bevy. J'ai notamment créé à la main une pyramide représentant les vaisseaux. Celle-ci est intégrée au jeu dans une branche de développement. Cette branche pourrait à l'avenir permettre de différencier une vue technique (zoom très faible) d'une vue plus détaillée, avec des vaisseaux et planètes plus réalistes ou éventuellement pour un passage en 3D.

Ainsi, ce stage m'a offert l'opportunité de découvrir de nouvelles technologies, notamment Rust et Bevy, tout en contribuant à un projet collaboratif d'envergure, bien plus ambitieux que ceux auxquels j'avais participé jusqu'alors.

Références

[1] Rust Programming Language, en-US, `https://www.rust-lang.org/`

[2] Git - Reference, `https://git-scm.com/docs`.

[3] Bevy Engine, en, `https://bevyengine.org/`.

[4] GitHub, `https://github.com/`

[5] Rust Standard Library : `Vec<T>`, en, `https://doc.rust-lang.org/std/vec/struct.Vec.html`

[6] Bevy Documentation – Query Struct, en, `https://docs.rs/bevy/latest/bevy/prelude/struct.Query.html`

[7] Leapfrog Integration, en, `https://fr.wikipedia.org/wiki/M%C3%A9thode_saute-mouton`

[8] L'OpenData du Système solaire, fr, `https://api.le-systeme-solaire.net/`

[9] Orbital elements, en, `https://en.wikipedia.org/wiki/Orbital_elements`

[10] Page GitHub du projet Solar4X, `https://github.com/Sarah-Keghian/solar4x/`