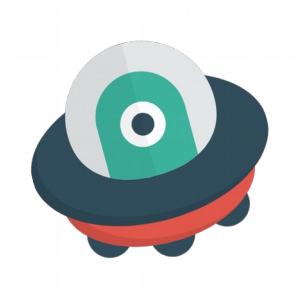
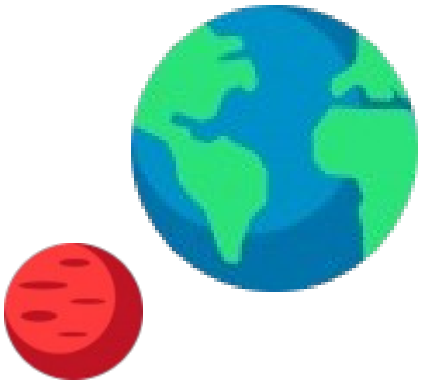
THIVEND Sarah, TANGUY Enorine, POIROT Nicolas, JAVERLIAT Charles



GRAVITY ROCKET

Compte rendu du projet d'informatique de 2ème année

INSA Lyon 2018 – 2019



THIVEND Sarah, TANGUY Enorine, POIROT Nicolas, JAVERLIAT Charles

Cahier des charges

Le projet Gravity Rocket a pour but de nous faire développer un programme informatique, qui nous permettrait de mettre en pratique les connaissances vues en cours jusqu'à aujourd'hui (Programmation Orientée Objet ; Interfaces Graphiques ; etc.).

Notre programme est un jeu, qui met en scène une fusée devant voyager d'une planète à une autre, et devant braver la gravité et autres éléments pour pouvoir se poser.

- **Initialisation** : Sur le menu de sélection des niveaux, le joueur peut choisir lequel lancer. La difficulté augmentant, la fusée devra faire face à de plus en plus d'obstacles (astéroides, aliens) ou de mouvements compliqués à réaliser (virage en utilisant la gravité des planètes). Il faudra économiser son fuel pour réussir à se poser/atteindre la zone.
- **Action** : Le jeu commence en partant d'une planète. Il y a deux types de niveaux, ceux où la fusée doit atterrir sur une planète, ou atteindre une constellation. La gravité permet d'attirer la fusée vers les planètes (pour des raisons de jouabilité, les planètes ne s'attirent pas entre elles). Le joueur devra éviter les astéroïdes et les tirs lasers des aliens pour atteindre la planète/zone cible, sinon elle peut exploser au contact de ces derniers. Dans le cas des niveaux où la fusée doit se poser sur une planète cible, il est important que la fusée ait une vitesse et un angle correct relativement à la planète, sinon elle se crash.
- **Game Over** : Une partie peut prendre fin pour plusieurs raisons : un mauvais atterrissage sur l'une des planètes (crash), une collision avec un obstacle dans l'espace, une dérive du vaisseaux dans les tréfonds de l'espace, ou un tir bien placé de la part des petits hommes verts.

Problèmes posés, solutions employées

- *Comment gérer la force gravitationnelle entre les entités du niveau ?*

Le principe du jeu est basé sur la force gravitationnelle telle que décrite par Isaac Newton. Pour mettre en œuvre ce principe de forces entre les différents corps, nous avons opté pour une méthode dynamique/discrète, plutôt que prévisionnelle/pré-calculée. En d'autres termes, à chaque mise à jour du jeu (60 fois par seconde), on recalcule et on applique les accélérations liées aux forces gravitationnelles à chaque entité, plutôt que de calculer au préalable les équations de trajectoires.

Pour ce faire on utilise simplement la relation de Newton :

$$\sum F_{ext} = m \cdot \vec{a} \Rightarrow \vec{a} = \frac{\sum F_{ext}}{m}$$

Sachant que chaque entité applique une force gravitationnelle sur les autres entités, on somme donc des forces de la forme :

$$\vec{F_{B/A}} = G \cdot \frac{m_A \cdot m_B}{d^2} \cdot \frac{\vec{AB}}{||\vec{AB}||}$$

- *Comment gérer les collisions entre les entités du niveau ?*

Pour développer la fonctionnalité où la fusion peut se crasher sur les planètes, ou entrer en collision avec d'autres entités mobiles (Aliens, lasers, …), il nous a fallut réfléchir à un système de collision adapté. La plupart de nos entités étant approximativement rectangulaires (fusée, alien, lasers, …) ou rondes (planètes, sattelites, …), il semblait pertinent d'utiliser ces deux formes comme boites de collision. Le problème se pose alors, comment vérifier que deux boites de collision rentrent en contact ? Nous avons donc identifié tous les cas de figure possibles afin de les traiter dans leur intégralité.

- **Cercle à cercle :** C'est le cas de figure le plus simple, si la distance entre les deux centres des cercles est inférieure à la somme des rayons des cercles, alors les cercles entrent en collision.
- **Cercle et rectangle :** Différents cas de collision sont possibles (cf illustration ci-contre) :
  - Si au moins un point du rectangle se trouve dans le cercle (distance au centre inférieure au rayon)
  - Si le centre du cercle est contenu dans le rectangle (On calcule l'aire des triangles formés avec le point et les sommets du rectangles à l'aide de la formule de Héron, si la somme de ces aires est supérieure à l'aire du rectangle alors le point est en dehors, sinon à l'intérieur du rectangle).

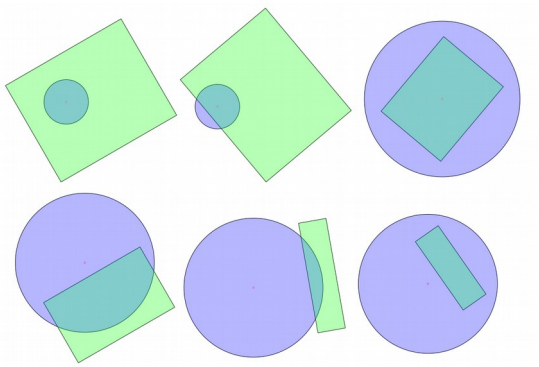


Illustration 1: Schéma des différents cas de collision Cercle/Rectangle

- Si une arête du rectangle coupe le cercle alors il y a collision. Pour ce cas on calcule la projection du centre du cercle sur l'arête AB du rectangle. On vérifie si la distance au centre du cercle est inférieure au rayon, puis que ce point projeté C appartient bien au segment AB du rectangle. Pour cela on vérifie que $\vec{AC}.\vec{AB} \geq 0$ (le point C se trouve bien après le point A dans la direction AB) et que $\vec{AC}.\vec{AB} \leq \|\vec{AB}\|$ (le point C ne dépasse pas B).

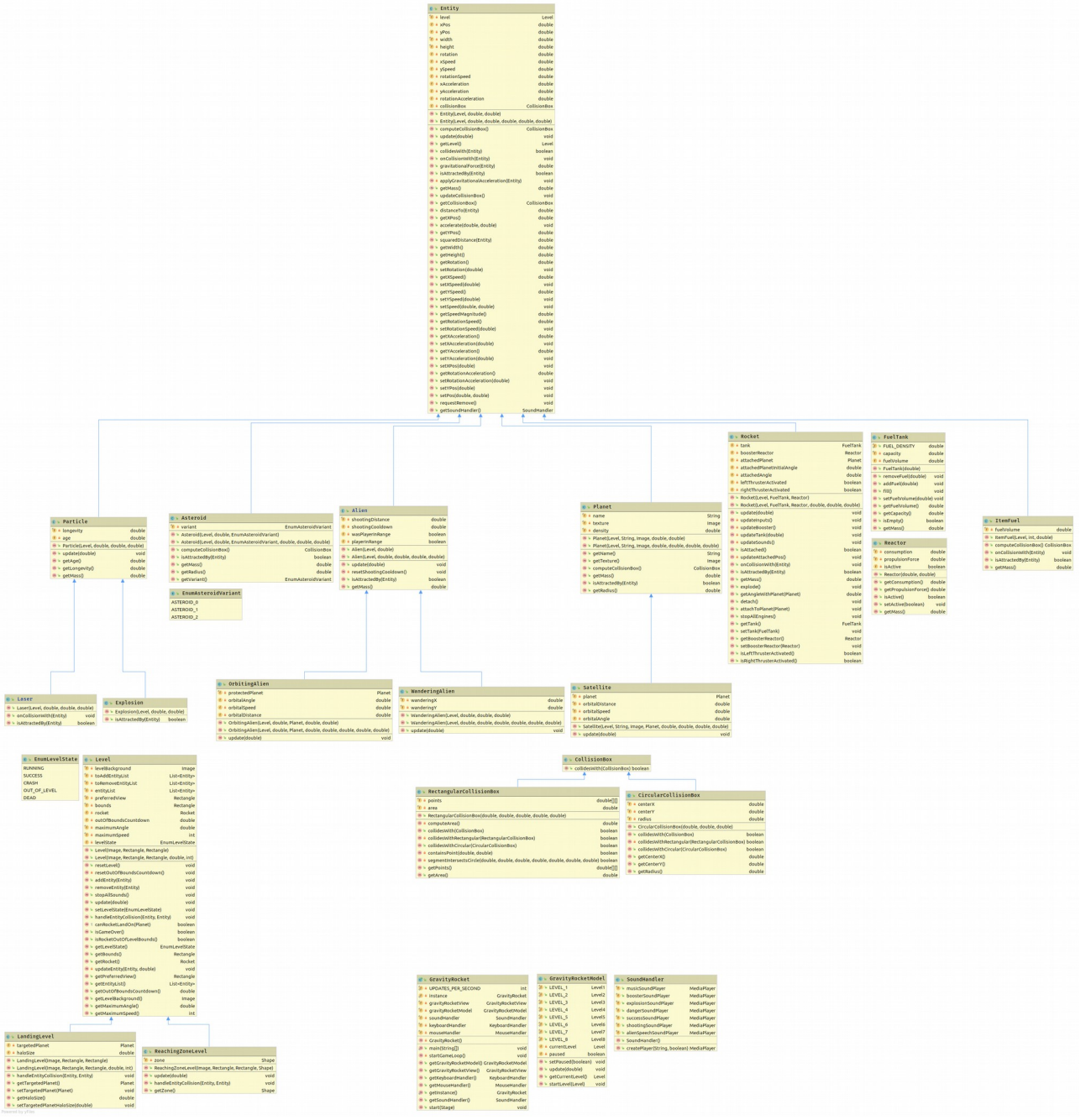


Illustration 2: Diagramme UML de la partie logique

Voir en ligne: https://github.com/Reden-Rane/GravityRocket/blob/master/uml_diagram_logic.png

- **Rectangle et rectangle :**
  - Ici il suffit de vérifier qu'au moins un point des rectangles se trouve dans l'autre rectangle (On calcule l'aire des triangles formés avec le point et les sommets du

rectangles à l'aide de la formule de Héron, si la somme de ces aires est supérieure à l'aire du rectangle alors le point est en dehors, sinon à l'intérieur du rectangle).

- *Comment gérer les particules (explosions, lasers) ?*
  - Chaque particule hérite du type entité, chaque particule possède un âge et une longévité, ainsi quand l'âge de la particule dépasse sa longévité elle est détruite du niveau, libérant de l'espace en mémoire.
- *Comment structurer efficacement le programme pour le rendre modulable ? (Ajout de nouvelles entités dans le niveau, nouveaux niveaux, etc.)*
  - Dans un premier temps nous avons découplé la partie logique et graphique du jeu (pattern Model-View-Controller). Cela permet de modifier facilement le rendu des entités sans avoir à toucher aux classes qui gèrent leur partie logique, ou bien de changer le rendu d'une entité en cours de partie par exemple, ce qui peut s'avérer très pratique pour ajouter de nouveaux effets aux entités dans certains cas par exemple.
  - Pour ajouter des entités dans le niveau on passe par des listes qui les stockent temporairement pour les rajouter après la boucle de mise à jour de toutes les entités. Idem pour les suppressions. En effet, si on supprime/ajoute des entités pendant la mise à jour d'une autre entité, on peut avoir à faire à une exception de co-modification de la liste des entités ce qui mène à un crash du jeu. On règle ainsi ce soucis.
  - Comme on stocke la liste des entités dans une liste présente dans le niveau, il suffit d'y ajouter les entités que l'on veut pour que notre système les mette à jour et les dessine automatiquement, sans ajout supplémentaire dans le code. Cela permet de créer de nouvelles entités et de les ajouter en jeu facilement.
  - On peut facilement changer le comportement des entités (ie : IA des aliens) en surchargeant la méthode update dans une classe fille d'Entity.

Bibliographie

https://fr.wikipedia.org/wiki/Gravitation, formule de gravitation utilisées dans le jeu

https://fr.wikipedia.org/wiki/Formule_de_H%C3%A9ron, formule de Héron utilisée pour calculer l'aire de triangles, utilisé dans les collisions pour vérifier qu'un point fait partie d'un rectangle.

https://www.vectorstock.com/, site regroupant une banque d'images que nous avons utilisées pour notre jeu.

https://www.youtube.com/watch?v=WFkGjEut9U4, musique issue du jeu Faster Than Light

https://www.youtube.com/watch?v=CwTWaN5ZMgQ, musique issue du film 2001 L'Odyssée de l'espace

https://www.youtube.com/watch?v=ndQ-IFVPoAA, musique issue du film Batman The Dark Night Rises

https://www.youtube.com/watch?v=m3zvVGJrTP8, musique issue du film Interstellar

https://www.youtube.com/watch?v=GBp1Y-LoObs, musique issue du film Les Gardiens de la Galaxie

https://www.youtube.com/watch?v=gqB3rg2oWCc, musique issue du film Seul sur Mars

https://www.youtube.com/watch?v=6RO7K4W-c9g, musique issue du jeu Mass Effect

https://github.com/iluwatar/java-design-patterns/tree/master/model-view-controller, dépôt utile pour s'aider sur le pattern design Model-View-Controller.

https://docs.oracle.com/javase/8/docs/, documentation de Java 8

https://stackoverflow.com/, utilisé dans le cas de problèmes rencontrés (ie. Transparence agissant étrangement sur les boutons de l'IHM)

Architecture du projet

Comme précisé précédemment, nous avons employé une architecture Model-View-Controller, qui découple la partie logique et graphique. Nous avons donc trois packages principaux : logic, graphics et controller.

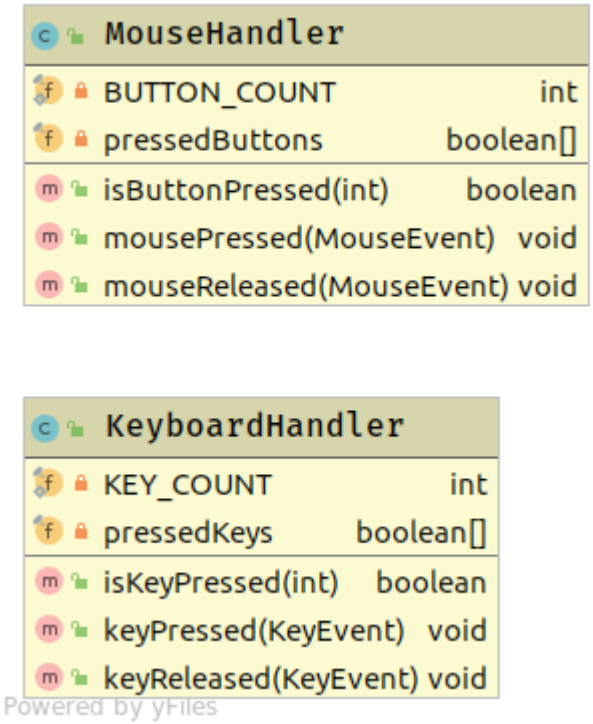


Illustration 3: Diagramme UML de la partie contrôleur

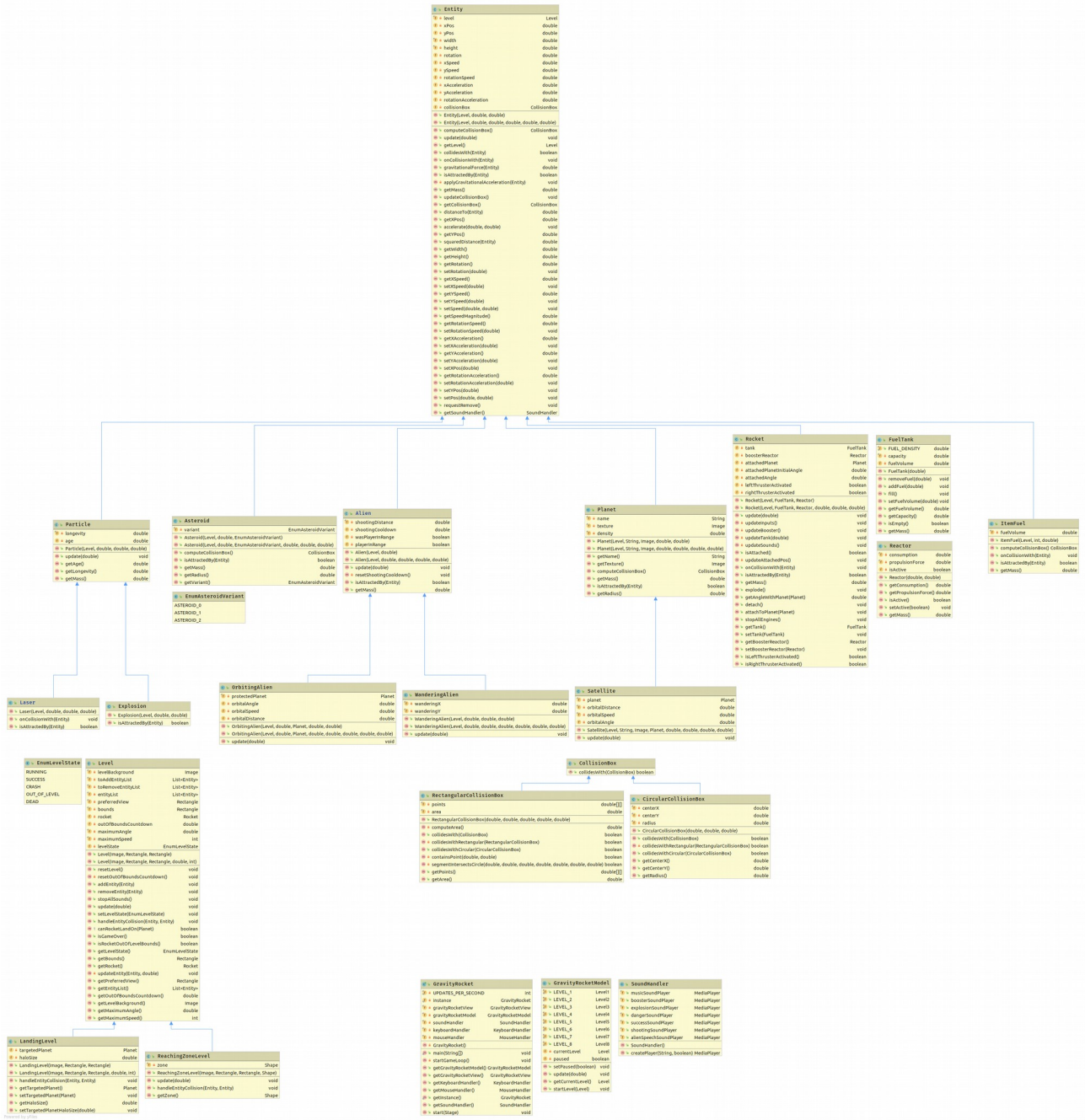


Illustration 4: Diagramme UML de la partie logique

Voir en ligne : https://github.com/Reden-Rane/GravityRocket/blob/master/uml_diagram_logic.png

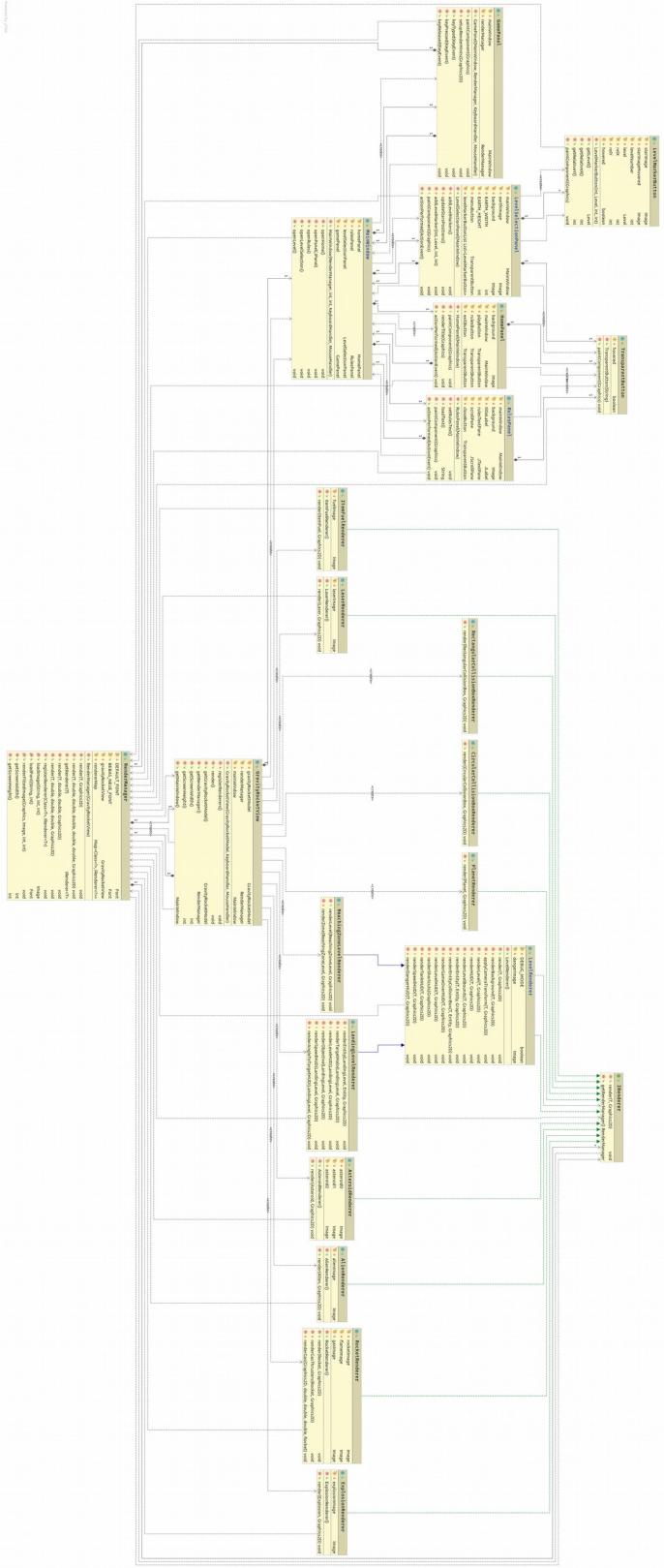


Illustration 5: Diagramme UML de la partie graphique

Voir en ligne :
https://github.com/Reden-Rane/GravityRocket/blob/master/uml_diagram_graphics.png

Suggestions d’améliorations

Nous n’avons pas de bugs recensés. Si nous avions eu plus de temps nous aurions pu rajouter : un système de score pour se comparer à d’autres joueurs, un mode multijoueur où il faudrait faire une course contre la montre avec un autre joueur, ajouter des astéroïdes en mouvement et finalement, ajouter d’autres niveaux.

Organisation des séances

N° de le séance	Travail réalisé
1	• Interface Homme-Machine du panel d’accueil • Mise en place du squelette du programme
2	• Interface Homme-Machine du panel d’accueil et sélection des niveaux • Ajout de la fusée, du booster, du réservoir
3	• Interface Homme-Machine du panel de sélection des niveaux • Ajout du système de niveaux et des planètes
4	• Interface Homme-Machine du panel des règles du jeu • Rédaction des règles • Ajout des boites de collisions
5	• Design des niveaux