- 1ère Année FILIÈRE INGÉNIEUR -

Programmation avancée

Livrable - Semestre 6

Jean-Brice Le Goff
Clément Weinreich

Mai 2021

Sommaire

1. Introduction

1.1 Présentation du projet

L'objectif du projet de programmation avancée est de réaliser un jeu de type rogue like en C#. Un rogue like est un sous-genre de jeux vidéo de rôle dans lequel le joueur doit explorer des donjons infestés de monstres dans le but de gagner des trésors et de l'expérience. La plupart des jeux de type rogue like se déroulent dans un univers fantasy, mais nous avons choisi de créer un univers de science fiction. Pour développer le jeu, nous avons utilisé deux librairies C# : Roguesharp[1] ainsi que RLNET[2]. La totalité du code réalisé est en Anglais. Nous avons fait ce choix pour garder une certaine cohérence entre les méthodes de la librairie RogueSharp qui sont en anglais, et notre code. Dans le même objectif de cohérence, les commentaires et la documentation ont également été écrits en anglais.

1.2 Présentation du jeu

Notre jeu vous met dans la peau d'un vaillant patrouilleur de l'espace, en mission pour retrouver trois artefacts. Vous avez déjà localisé les planètes sur lesquelles se trouvent les artefacts : Alleo le désert gelé, Damari aux cratères pourpres, et Thaadd à la jungle verdoyante. Trois téléporteurs sont déjà réglés vers chacune d'elles. Il ne vous reste plus qu'à explorer ces terres inconnues jusqu'à trouver les précieux artefacts. Mais prudence, car ils sont chacun gardés par de terribles adversaires, parmi lesquels des monstres redoutables, et des humains zombifiés. Pour vous défendre, vous avez accès à un variété d'armes, d'armures et d'objets de soin. Deux marchands croisés sur la route sont installés dans votre vaisseau, et peuvent vous fournir en équipement en échange de votre or. Votre quête ne prendra fin qu'une fois les trois artefacts réunis. Bon courage.

1.3 Exécution du code

Notre solution fonctionne sous .NET 5.0. Les versions utilisées des librairies externes sont :

- RLNET 1.0.6
- RogueSharp 4.2.0

[1] https://github.com/FaronBracy/RogueSharp
[2] https://github.com/WildGenie/rlnet

- System.Drawing.Common 5.0.2

La solution est exécutable et a été testée sous Windows 10 et Ubuntu 20.04.2.

2. Fonctionnalités

2.1 Interface utilisateur

Voici une capture d'un écran habituel du jeu avec la délimitation de chaque parties de l'interface utilisateur :

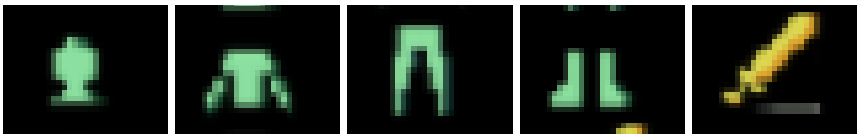
L'écran principal, qui prend toute la partie haute de l'écran, affiche la carte, les objets présents sur la map et les personnages.

Au centre sous l'écran de la carte, se trouve la console des statistiques du joueur. L'icône en forme de bras au poing serré représente la puissance d'attaque du joueur. Il s'agit donc des

dégâts qu'il est capable d'infliger à un ennemi selon son équipement. L'icône en forme de bouclier représente la défense du joueur, soit la valeur d'attaque entrant qu'il est capable d'absorber en prenant le minimum de dégâts possible."Gold" désigne la quantité d'or possédée par le joueur.

Juste sous la console des stats se trouve la console du joueur qui permet d'afficher les divers messages en réponse aux agissements du joueur.

L'écran en bas à gauche affiche les différents emplacements d'équipement et les équipements qui y sont affichés. Il y a un emplacement pour chaque partie du corps pouvant être protégée, auxquels s'ajoute l'emplacement d'arme représenté par une icône d'épée.



L'écran en bas à droite montre tous les objets consommables qui ont été ramassés par le joueur et qui sont disponibles dans l'inventaire.

La quasi-totalité des sprites utilisés dans le jeu ont été créés par nos soins, à l'exception des caractères de base destinés à l'écriture.

2.2 Gameplay

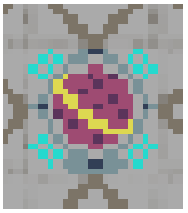
2.2.1 Déplacement

Le personnage se déplace de cellule en cellule à l'aide des touches Z (vers le haut), Q (vers la gauche), S (vers le bas) et D (vers la droite).

La plupart du temps, la caméra suit le joueur en le gardant toujours au centre de l'écran. Quand le joueur arrive à une extrémité de la carte, la caméra cesse de se déplacer dans cette direction. Cela permet de simuler le fait qu'elle bloque sur le bord de la carte.

Il est possible de se déplacer entre les différentes zones (planètes) par le biais de téléporteurs disposés à divers endroits dans le jeu. Il y en a trois dans le vaisseau qui mènent à chacune des planètes et disparaissent une fois que vous revenez avec l'artefact. Un quatrième téléporteur apparaît après le combat contre les boss de zone et permet de

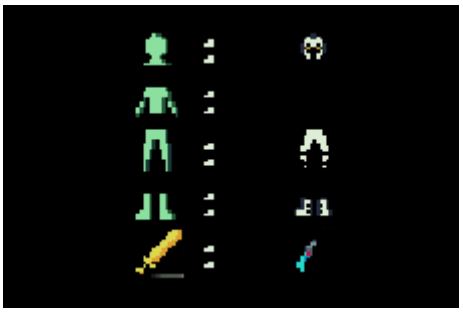
retourner au vaisseau. Pour activer un téléporteur, il faut se placer sur sa cellule et appuyer sur la touche Left Ctrl. Voici à titre d'exemple l'apparence de l'un des téléporteurs du vaisseau.



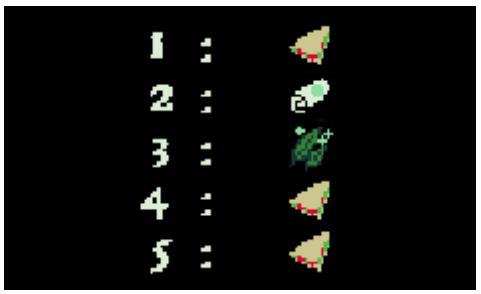
Dans une zone, on peut descendre au niveau inférieur en empruntant les escaliers cachés dans des extrémités des niveaux (le plus loin possible du joueur). Il s'utilisent comme les téléporteurs, avec la touche Left Ctrl.

2.2.2 Objets

Les objets désignent tous les éléments qui peuvent être ramassés par terre par le joueur. Ils comprennent les armes et les armures, ainsi que les items consommables. Le ramassage se fait simplement en déplaçant le personnage sur la cellule de l'objet. Le joueur n'a qu'un emplacement pour chaque partie d'équipement (arme et armure) et doit donc sélectionner laquelle conserver lorsqu'il en trouve une nouvelle. Si l'emplacement est déjà occupé par un équipement au moment du ramassage, l'ancien équipement est déposé sur une cellule autour du joueur et le nouveau est équipé.



Le joueur possède aussi un inventaire de 5 emplacements, numérotés de 1 à 5, pour les items consommables.



Lorsqu'il tente de ramasser un objet alors que son inventaire est plein, l'objet reste sur la cellule et un message indique qu'il transporte trop d'objets.

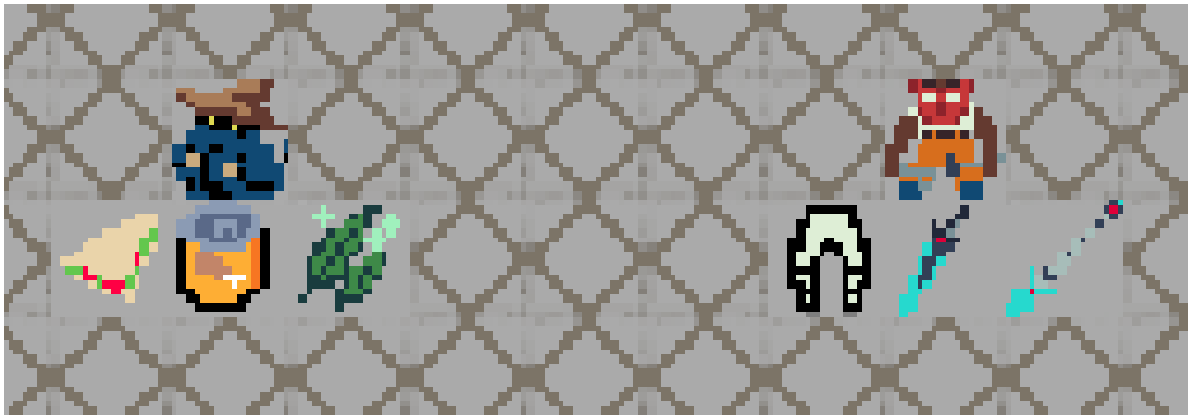
Afin de pouvoir à nouveau ramasser des consommables, le joueur doit donc utiliser l'un des objets qu'il possède déjà. Pour utiliser un item consommable dans l'inventaire, il suffit d'appuyer sur la touche du clavier qui correspond au numéro de l'emplacement contenant l'objet.

Dans son état actuel, le jeu ne comprend que des consommables de soin. Ils sont au nombre de 5 et offrent chacun une quantité de points de vie différente.

Le sandwich rend 20% de la barre de vie, l'herbe médicinale rend 30%, les bandages rendent 40%, la ration rend 50% des points de vie. Le kit de soin rend la totalité des points de vie au joueur.

2.2.3 Marchands

Les seuls personnages non jouables qui ne sont pas hostiles au joueur sont les marchands. Il sont tous deux installés dans l'arrière du vaisseau et vendent respectivement des consommables et des équipements (arme et armure).



Les marchands renouvellent leur étal à chaque nouvelle arrivée du joueur dans le vaisseau. La liste des équipements susceptibles d'être présentés par le marchand est échangée contre une liste d'objets plus puissants, mais aussi plus chers, à chaque fois que le joueur complète une planète et obtient son artefact.

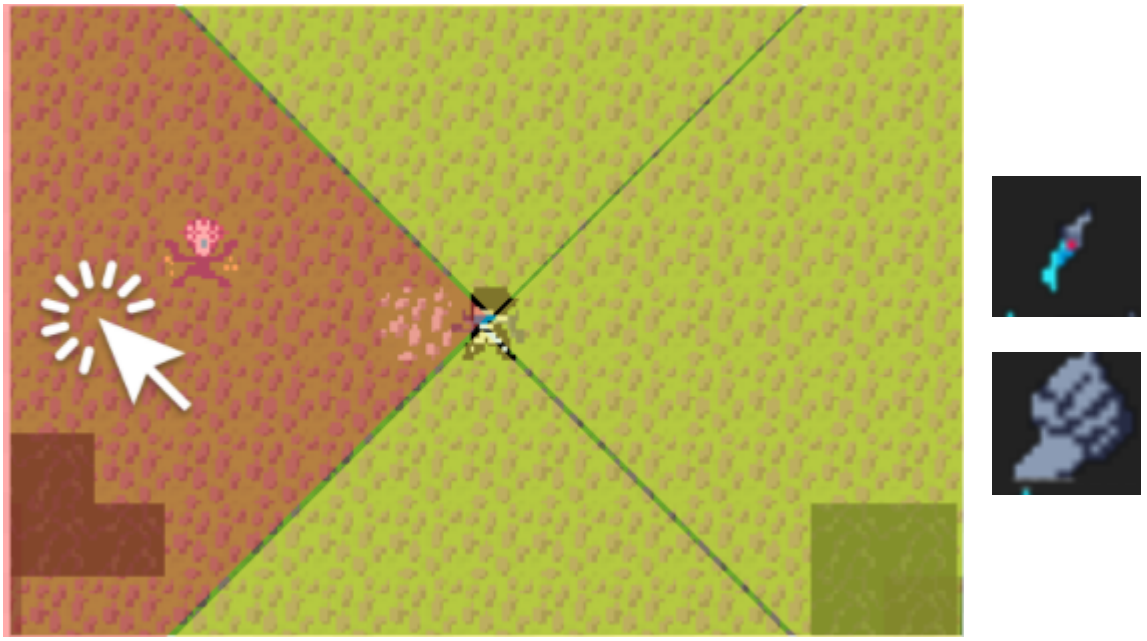
Pour acheter un objet au marchand, il suffit de se déplacer sur sa cellule comme pour ramasser un item classique. Si le joueur possède assez d'or, le prix de l'item en or est alors déduit du capital du joueur, et l'objet acheté est ramassé par le joueur. Dans le cas où le joueur n'a pas assez d'or, le joueur ne parvient pas à se déplacer sur la cellule de l'objet et un message s'affiche dans la console pour indiquer la somme qui manque au joueur pour acheter cet objet.

2.2.4 Combat

Au début de la partie le joueur ne possède aucun des équipements pour le combat. Il est donc, de base, équipé d'une arme appelée "fist" ("poing" en anglais) qui représente sa capacité de combat à main nue, avec une portée et des dégâts minimaux. Il est aussi équipé

sur toutes les parties du corps d'un équipement "none" ("aucun" en anglais) avec une défense minimale. Les équipements "fist" et "none" disparaissent dès que le joueur en trouve un meilleur sur le terrain ou chez le marchand.

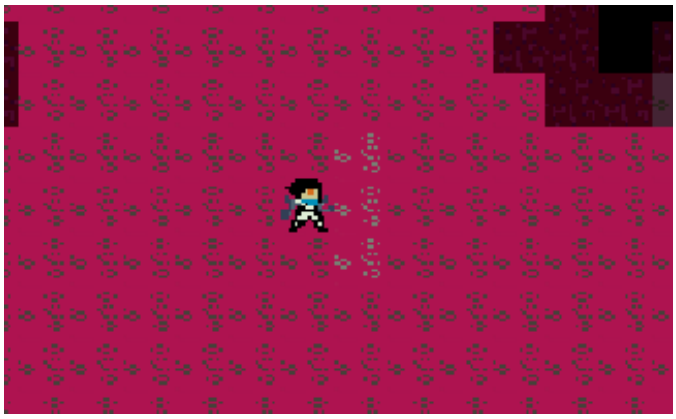
Les combats du jeu se déroulent en temps réel. Pour attaquer un ennemi, il suffit de cliquer dans la direction de cet ennemi (autour du joueur). On peut attaquer dans les directions haut, bas, droite et gauche. La zone touchée par l'attaque et les dégâts infligés varient en fonction de l'arme équipée par le joueur. La zone d'attaque de l'arme dépend du type d'arme (dague, épée, lance, ...). Les cellules attaquées apparaissent en surbrillance.

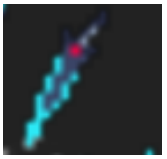


Dans cet exemple, le joueur clique dans la zone à gauche de son personnage et attaque donc sur la gauche avec une arme dont la portée ne permet de toucher qu'une cellule (il peut s'agir du poing ou d'une dague). Cette cellule apparaît donc en surbrillance.

Une lance permet d'attaquer une cases plus loin qu'une attaque de base :

Une épée permet permet quant à elle d'attaquer une ligne de trois cellules perpendiculaires à la direction de l'attaque :





Les dégâts infligés dépendent de la version de l'arme (Mk1, Mk2, ...) , où Mk désigne l'abréviation de "Mark", un terme anglais utilisé pour nommer les versions d'un objet, et notamment de certaines armes. Plus le numéro de version est haut, plus l'arme est puissante.

Les dégâts infligés et reçus dépendent aussi de la défense de l'entité attaquée, qui déduisent leur valeur de celle de l'attaque. La défense du joueur peut être modifiée par son équipement. Il existe un équipement de protection pour chaque partie du corps : casque pour la tête, plastron pour le torse, bas d'armure pour les jambes et bottes pour les pieds. De manière similaire aux armes, la valeur de défense octroyée par chaque pièce d'armure dépend du matériau dont il est constitué. Par ordre de résistance on trouve le polymère, le carbone, le platine et le titane.

Le sprite et la couleur de chaque arme et pièce d'armure varient en fonction de son type, de la version ou du matériau.

2.2.5 Ennemis de base

Le jeu comporte une variété d'ennemis qui se distinguent par leur sprite et quelques spécificités dans leur comportement.

Les comportements commun à tous les ennemis sont :

- le fait de réagir à la présence du joueur dès qu'il entre dans leur champs de vision en se mettant à le poursuivre pour l'attaquer,
- le fait d'attaquer avec son arme la case ou se trouve le joueur si ce dernier est à portée (au lieu de se déplacer),
- le fait de pouvoir être semé par le joueur s'il sort de son champ de vision pendant dix périodes de temps (cela sera détaillé par la suite).

Chaque instance d'ennemi possède un nombre de points de vie de base tiré aléatoirement entre une valeur minimale et une valeur maximale qui dépend du type d'ennemi.

Voici les différents types d'ennemis du jeu :

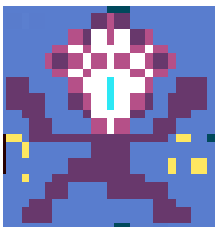
Le zombie :

Le zombie est l'ennemi le plus basique. Il se déplace, et attaque la case qui est en face de lui. Il peut avoir particulièrement peu de points de vie. La champ de vision du zombie est aussi plus faible que pour les autres ennemis.

La mecabat :

La mecabat est le plus rapide des ennemis, en termes de déplacement et d'attaque. Il a la plus faible valeur de dégât par attaque, mais sa rapidité lui permet d'effectuer plus de dégâts par seconde que les autres ennemis. Son champ de vision est identique à celle du joueur.

Le dendroide :



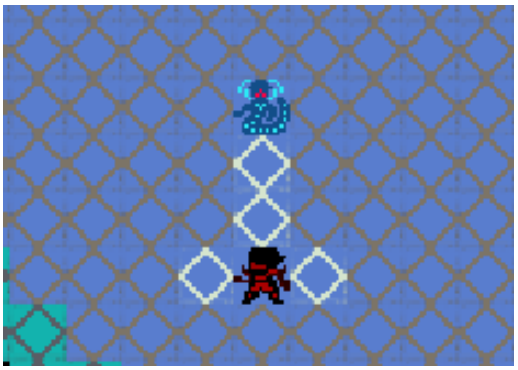
Le dendroide est le plus lent des ennemis mais ses attaques sont très puissantes et il possède le meilleur champ de vision de tous les ennemis.

Nous avions prévu que chaque ennemi ai quatre sprites comme le joueur, et ils sont donc codés de cette manière. Cela dit, afin de ne pas trop perdre de temps sur la réalisation de ces sprites, nous avons placé le même sprite dans les quatres attributs sprites des ennemis.

2.2.6 Boss de zone

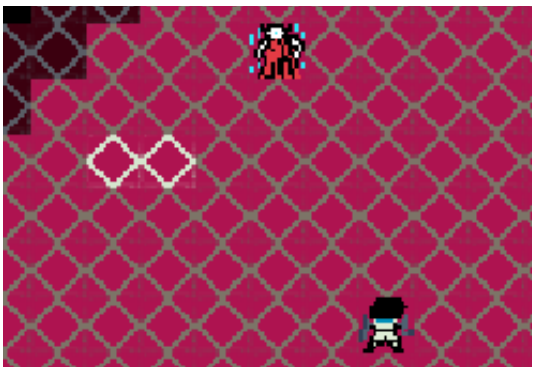
Sur la dernière salle de chaque planète se trouve un ennemi unique appelé "boss", qui est équipé d'une arme et de capacités spéciales. Tous les boss de zone ont une grande quantité de points de vie qui varie aléatoirement entre un minimum et un maximum.

Boss d'Alleo :



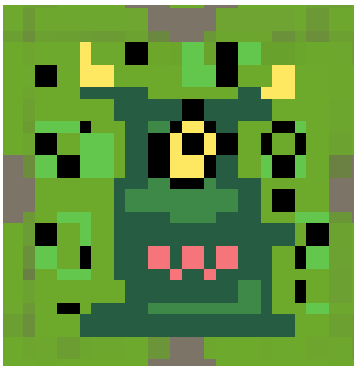
Le boss de la planète Alleo se comporte comme un ennemi classique mais a la particularité d'avoir une arme spéciale, le trident qui permet d'attaquer une zone en "T".

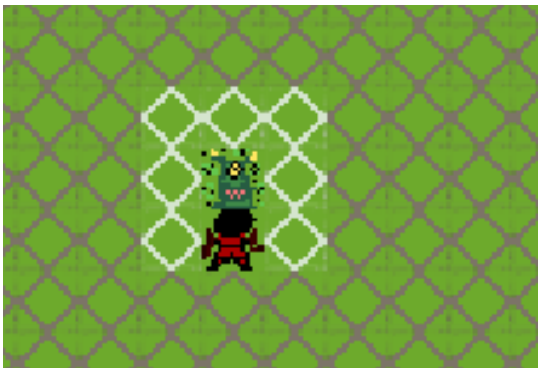
Boss de Damari :



Le boss de la planète Damari possède une arme similaire à la lance en termes de portée mais qui inflige des dégâts très importants. Ce boss possède une capacité de déplacement unique lui permettant de se téléporter à travers la pièce au lieu de bouger de case en case. Il attaque dans la direction vers laquelle il s'est téléporté. S'il touche le joueur, il ne se déplace pas. La direction de l'attaque est aléatoire, mais la quantité de dégât infligée n'est pas négligeable ce qui force le joueur à éviter de se retrouver autour du boss avant qu'il ne se téléporte.

Boss de Thaadd :





Le boss de Thaadd se comporte aussi comme un ennemi standard mais possède en arme spéciale, une "faux de la mort" permettant d'attaquer toutes les cases aux alentours de celuli-ci.

Au moment où un boss est vaincu, il laisse sur le sol l'artefact de la planète à récupérer, ainsi que l'arme dont il était équipé. Simultanément, un téléporteur vers le vaisseau apparaît. Si le joueur oublie de ramasser la relique avant d'emprunter le téléporteur pour quitter la zone, le téléporteur vers la planète ne disparaît pas et il peut recommencer l'exploration de la zone.

2.2.7 Génération des zones

La plupart des zones sont générées grâce aux méthodes RogueSharp de création de "Map", et suivent le pattern de caverne (CaveMapCreationStrategy). Le vaisseau et les salles de Boss sont des zones que nous avons pré-construites à l'aide de méthodes pour sélectionner plusieurs cellules selon une disposition particulière.

2.2.8 Système de planification des actions

Le jeu se déroule en temps réel. A chaque période de temps (définie à 100 millisecondes) on fait le tour de tous les ennemis de la zone pour vérifier si le nombre de périodes qu'ils doivent attendre entre chaque action est atteint. Il varie en fonction de l'ennemi, et définit donc sa vitesse d'action, à la fois pour le déplacement et l'attaque. Si l'ennemi peut agir, son action est choisie en fonction de la situation (état d'alerte par rapport au joueur, proximité géographique avec le joueur, ..).

Même s' il concerne seulement les ennemis dans l'état actuel du jeu, nous avons choisi de parler du système de planification dans une partie indépendante car il aurait pu nous permettre de couvrir le déroulement des actions de tous les personnages non jouables.

2.2.9 Leveling

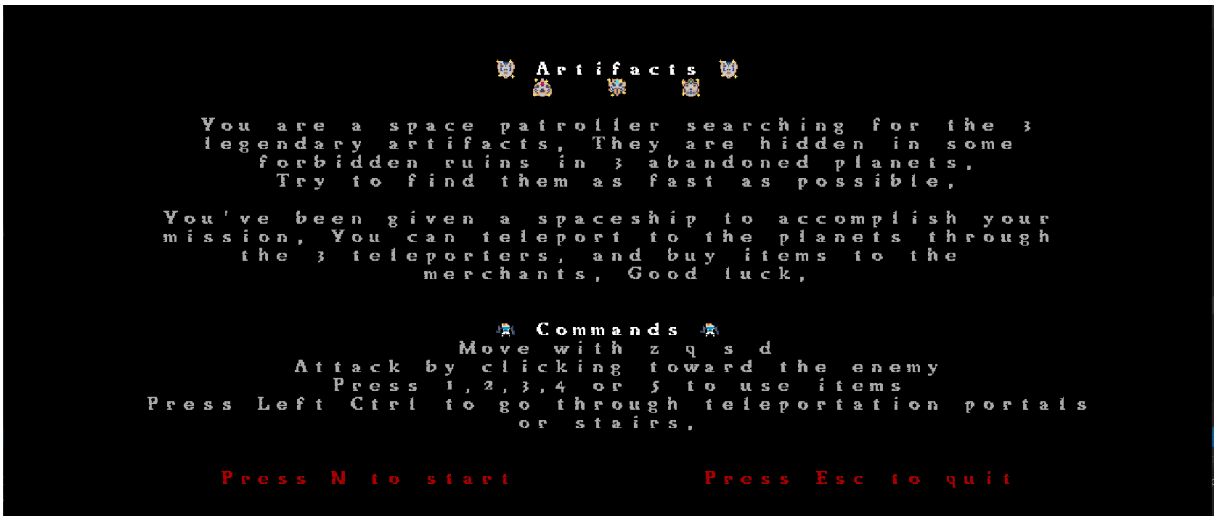
Le jeu dispose d'un système de leveling qui permet d'adapter la difficulté du jeu. Le niveau est égal à la somme du nombre d'étages parcourus dans la zone et du nombre d'artefacts détenus par le joueur. A chaque fois que le joueur emprunte un escalier, cela augmente le nombre d'étages parcourus. Quand le joueur revient dans le vaisseau après l'exploration d'une planète, cela réinitialise le niveau de difficulté. Ce dernier reste cependant incrémenté de 1 de manière permanente suite à l'acquisition de l'artefact de la planète.

Le niveau de difficulté affecte à la fois la puissance des ennemis et celle des équipements (armes et pièces armures) qu'il est possible de trouver. Chaque niveau de difficulté ajoute

des équipements plus puissants dans la liste des équipements pouvant être générés sur la carte.

2.2.10 Ecran de début et de fin de partie

Au lancement du jeu, on trouve d'abord un écran expliquant l'histoire du jeu et l'objectif du joueur, ainsi que les contrôles du personnage. Il indique aussi le bouton pour lancer la partie (la touche 'N') et celui pour quitter le jeu (la touche 'Echap').



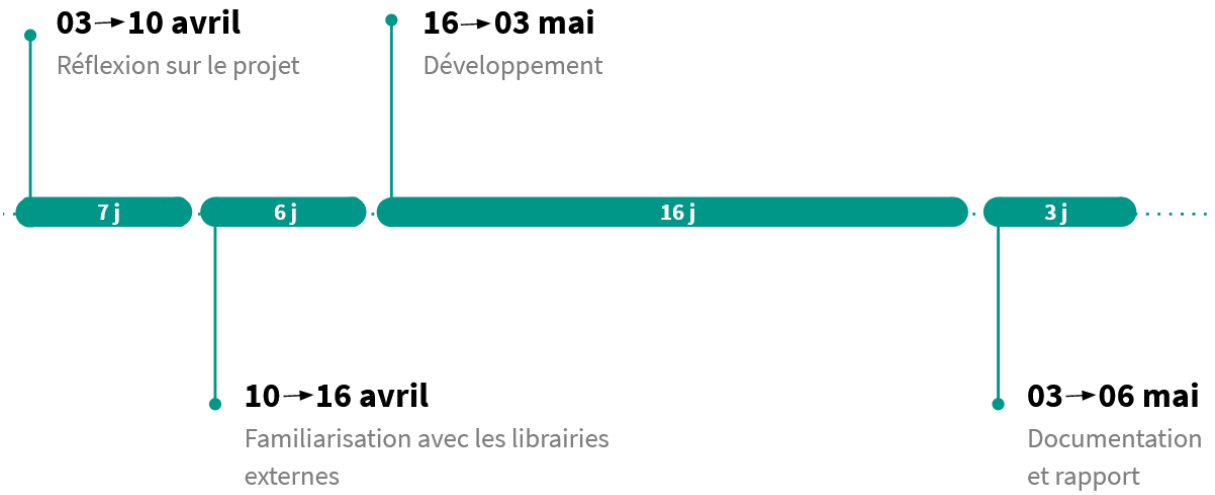
Ces deux options apparaissent également lorsque le joueur perd la partie (les points de vie sont tombés à 0) et lorsque qu'il gagne (les trois artefactes ont été collectés).

Sur l'écran de victoire, on peut aussi trouver le chrono affichant le temps mis par le joueur pour finir la partie.

3. Gestion de projet

3.1 Phases de réalisation du projet

Concernant notre organisation, nous avons découpé ce projet en 3 phases. On peut représenter ces phases sur ce graphique :



3.1.1 Phase de réflexion

Durant la phase de réflexion, nous avons réfléchi principalement au gameplay de notre jeu. L'histoire et l'univers étant facilement adaptable à n'importe quel type de jeu, le gameplay nous semblait être l'élément central qu'il fallait définir. Initialement nous avions imaginé un

jeu de type "escort", dans lequel il fallait accompagner un personnage et le défendre face à des ennemis. Nous avons ensuite adapté cette idée en imaginant un système d'accompagnement du joueur par un "familier" qui permettrait au joueur de résoudre des énigmes, et se défendre face à certains ennemis coriaces. Nous avons alors validé cette idée, puis avons imaginé un univers de science fiction futuriste. Le familier s'est alors transformé en drone. Comme vous avez pu le voir, nous n'avons pas eu le temps d'implémenter cette fonctionnalité, mais toute l'architecture que nous avons établie était adaptée à l'ajout de personnages non jouables aidant le joueur, tel qu'un drone. L'histoire du jeu s'est ensuite dessinée au fur et à mesure de l'avancement du projet.

C'est lors de cette phase que nous nous sommes renseignés sur les différentes librairies externes que nous aurions pu utiliser. Nous avions notamment retenu SadConsole, Monogame, Roguesharp et RLNET. C'est lors de la phase suivante que nous avons choisi laquelle d'entre elles allait être utilisée.

3.1.2 Phase de familiarisation avec les librairies externes

Une fois que nous avions une idée du gameplay et de l'histoire du jeu, nous nous sommes lancés dans une phase de test des librairies externes. Initialement, nous comptions utiliser Monogame. Cependant, nous avons trouvé que la partie graphique prenait une trop grande place dans le code, ce qui n'était pas l'objectif principal du projet. Certes, Monogame offrait des possibilités que les autres librairies graphiques ne proposaient pas, mais un roguelike ne nécessitait pas tant de complexité et de fonctionnalités. Ensuite, lors des tests de RLNET, nous avons trouvé la librairie simple d'utilisation et très intuitive. C'est pourquoi nous l'avons choisi.

Cette phase consistait donc à se familiariser avec les librairies RLNET et Roguesharp, afin de les maîtriser pour pouvoir les utiliser à bon escient dans notre projet. Il nous semblait impensable de se lancer dans le projet sans avoir déjà utilisé ces librairies. C'est pourquoi nous avons consacré 1 semaine à la réalisation d'un premier projet suivant le tutoriel Roguesharp (cité dans la partie 3.1.1). Ce tutoriel très complet permet de faire le tour des fonctionnalités de rogue sharp, et propose une structure type de fichiers pouvant être utilisée. Après avoir appris à utiliser Roguesharp et RLNET, nous avons pu commencer à développer notre projet.

3.1.3 Phase de développement du projet

La phase de développement s'est organisée sous forme de plusieurs séances de travail intensives. Pour versionner le code, nous avons utilisé le repo Github créé par Github classroom dans l'organisation ensc-progav. Nous avons commencé par développer les fonctionnalités de base que nous maîtrisions grâce au tutoriel. Cela comprend donc les déplacements, les ennemis, la génération et la gestion de la map. Nous avons cependant implémenté ces éléments d'une manière différente que dans le tutoriel, notamment pour toute la partie écran de jeu. Nous avons ensuite alimenté ce code pour se rapprocher de notre idée initiale : un personnage qui voyage à travers son vaisseau dans l'espace.

Lors de cette phase, nous avons également consacré un temps important à la réalisation des sprites du jeu et de leur intégration dans le "font file" lu par la console RLNET. Cette manière d'utiliser des sprites est quelque peu détournée de l'objectif du "font file", mais cela nous a permis d'ajouter des sprites assez facilement dans le jeu.

Pour la répartition des tâches, Jean-Brice s'est davantage occupé de la création et gestion des graphismes, des interactions du joueur avec la souris, et plus généralement du gameplay. Clément s'est davantage occupé de la partie système, objets, équipements et implémentations des personnages ainsi que leur comportement.

3.1.4 Phase de documentation

Une fois un maximum de fonctionnalités implémenté en fonction du temps que nous avions, nous avons décidé de rédiger une documentation complète du code en respectant une norme xml. Cette documentation se rajoute aux commentaires qui étaient en partie déjà présents dans le code. Ensuite, grâce à l'outil Doxygen, nous avons pu exporter cette documentation au format html. La page html de documentation générée ne contient pas seulement la documentation que nous avons écrite, elle contient également des diagrammes de classes ce qui permet de mieux comprendre certaines interactions entre les classes.

Lors de cette phase, nous nous sommes également occupés de la rédaction de ce rapport.

3.2 Outils utilisés

Pour travailler à 2 sur le même projet, nous avons utilisé principalement github. Nous étions déjà très familier avec cet outil, et avons déjà travaillé ensemble avec Gitlab pour le projet de programmation du premier semestre. Cela nous a donc permis d'être efficaces pour mettre en commun le code.

Nous avons également utilisé l'extension live share de visual studio et visual studio code. Cela nous permettait de développer en même temps sur les mêmes fichiers lorsque cela était nécessaire.

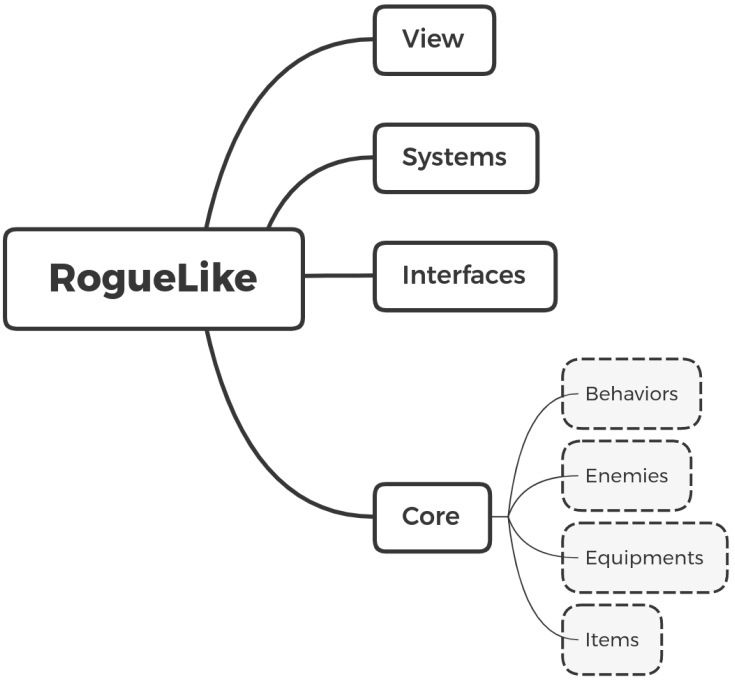
4. Conception

Nous allons maintenant expliquer comment le projet a été structuré.

4.1 Architecture du projet

4.1.1 Les espaces de noms

Premièrement, nous avons basé l'architecture de notre projet sur celle du tutoriel RogueSharp[3]. On retrouve donc certains espaces de noms comme Systems, Core ou interfaces. Cependant, nous avons modifié et adapté cette architecture pour correspondre à nos besoins. Voici l'architecture finale de nos espaces de noms :



[3] https://roguesharp.wordpress.com/2016/02/20/roguesharp-v3-tutorial-introduction-and-goals/

Nous avons fait le choix de séparer toute la partie console et vue dans le namespace View afin de s'y retrouver plus facilement parmi les nombreuses classes. Les classes dont l'objectif était de gérer les interactions entre le jeu et ses éléments, gérer la caméra, générer des maps ou des items ont été placées dans le namespace Systems. Ce namespace contient donc toutes les classes qui permettent de faire fonctionner le jeu, mais qui ne sont pas des éléments à part entière de celui-ci. Ensuite, nous avons regroupé les interfaces dans le namespace Interfaces. De plus, chaque nom d'interface commence par un 'I' (i majuscule) pour permettre de les reconnaître facilement. Pour finir, l'espace de nom Core contient tous les éléments du jeu. On y retrouve par exemple des classes liées à la map, au joueur, etc. Cet espace de nom contient lui même d'autres espaces de noms :

- Behaviors : les classes liées au comportement des ennemis
- Enemies : les classes liées aux ennemis, on y retrouve les ennemis mais également les boss.
- Équipements : toutes les classes qui décrivent des équipements. Cela peut être des équipements d'attaque, ou de défense.
- Items : toutes les classes qui décrivent des items utilisables. Pour l'instant il n'y a que des items permettant de soigner le joueur.

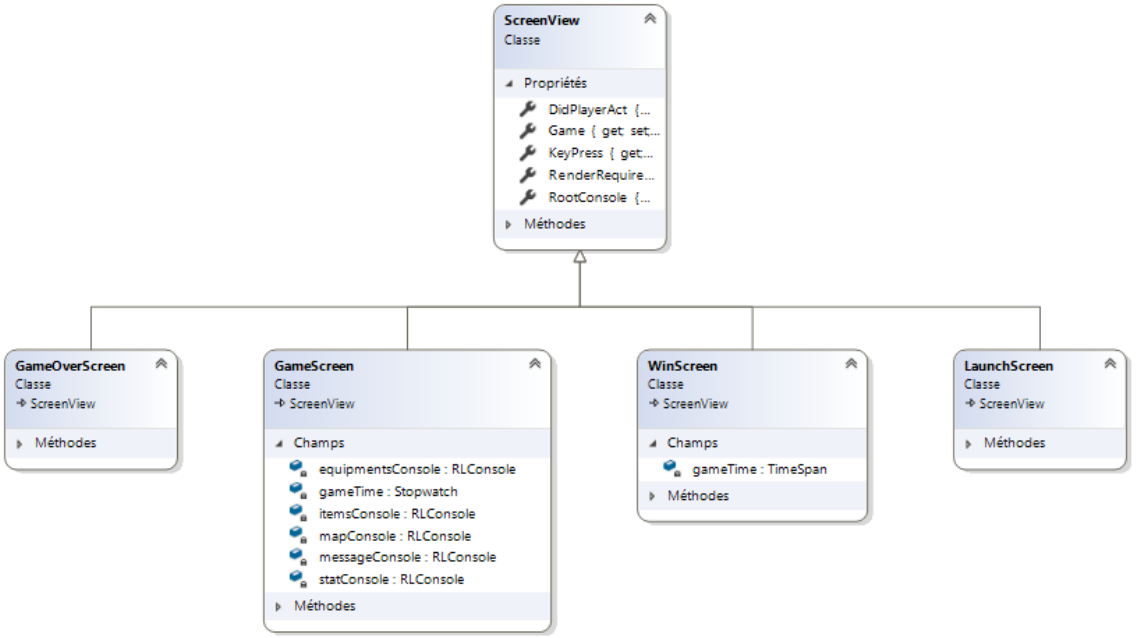
4.1.2 Organisation des principales classes/interfaces

Dans cette partie, nous allons détailler les principales composantes de notre code. Certaines classes "mineures" seront omises, car possèdent peu d'interactions avec les autres classes. Cependant, pour plus de détails, vous pouvez vous référer à la documentation du code en ouvrant *documentation.html.* Cette documentation a été générée grâce à l'outil Doxygen à partir de la documentation XML que nous avons écrite. Cette documentation rentre en détail dans le fonctionnement de chaque méthode, et dans le rôle de chaque classe, attribut ou propriété.

De plus, nous avons inclus un diagramme de classe complet permettant de montrer les interactions entre chacune des classes. Du fait de sa taille et de sa mauvaise lisibilité, nous avons décidé de ne pas l'inclure dans ce rapport, mais il est compris dans le répertoire du rendu.

Classes liées à la vue (console d'affichage)

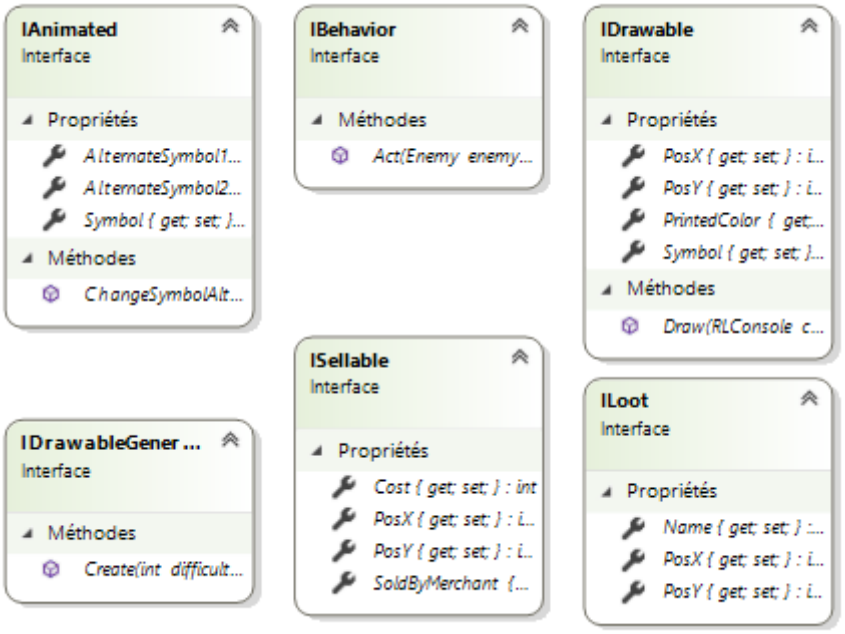
Pour gérer l'affichage, nous avons utilisé la librairie RLNET. Cette librairie permet de créer facilement des interfaces de jeux fonctionnant avec des cellules. Cette partie est donc la seule qui s'occupe de la gestion de l'affichage. Voici sa structure :



Il y a une classe mère ScreenView qui contient la console principale sous forme d'une propriété statique. Cela permet aux différents écrans (écran de victoire, de lancement, de défaite ou de jeu) qui héritent de la classe ScreenView, d'utiliser une et une seule console. Chaque classe qui hérite de ScreenView est donc un type d'écran. Pour l'écran de jeu, plusieurs consoles viennent s'inclure dans l'écran principal. Cela permet de gérer plusieurs éléments de l'écran différemment comme par exemple l'inventaire, les statistiques du joueur, etc. La classe ScreenView contient également une propriété statique Game. La classe Game contient tous les éléments du jeu tels que la map, le joueur ou le système de planification (Scheduling).

Interfaces

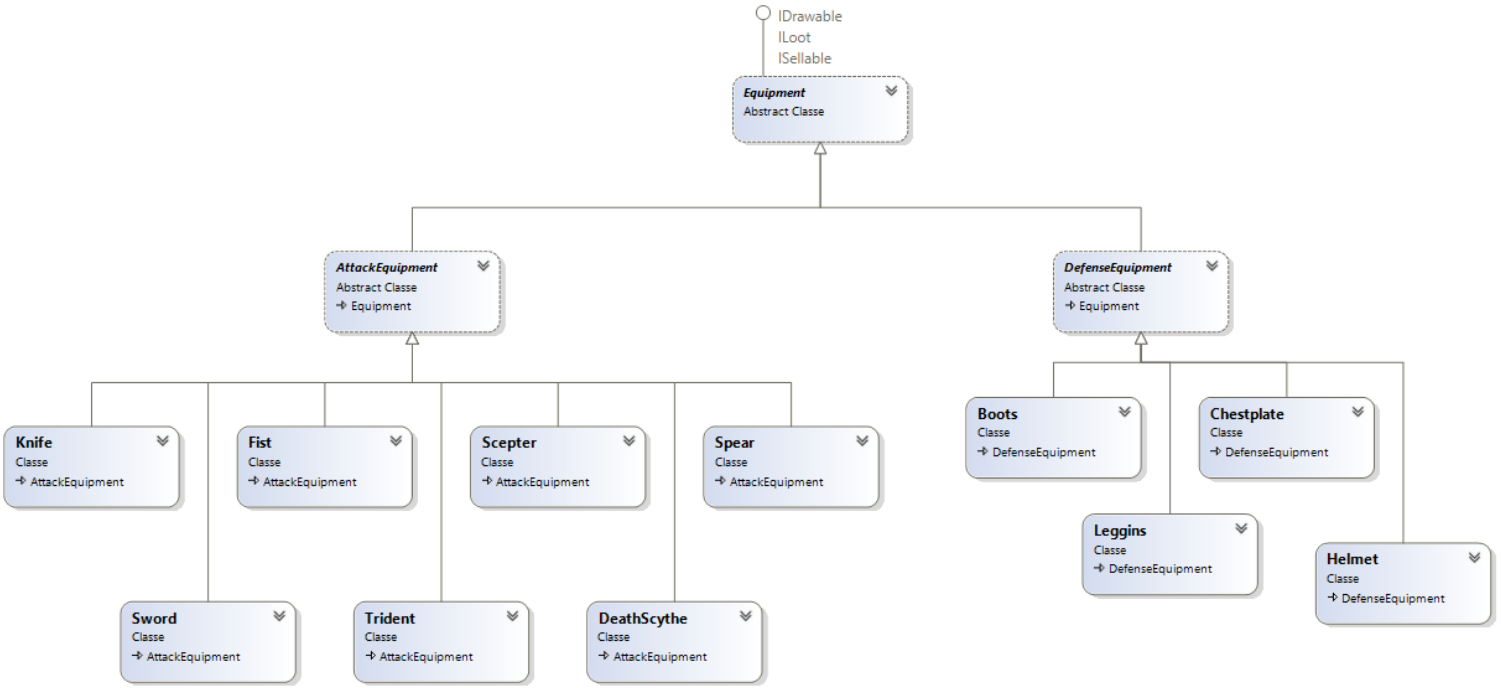
Nous allons maintenant décrire les interfaces que nous avons utilisées.



Ces interfaces nous ont permis de décrire des comportements généraux ce qui fut très utile par la suite lors du développement. Par exemple, tous les éléments pouvant être affichés sur la console proviennent d'une classe qui implémente l'interface IDrawable. Les objets qui peuvent être posés sur le sol sur la map sont représentés par des classes qui implémentent ILoot. De plus, le C# autorise l'implémentation de plusieurs interfaces ce qui permet de combiner plusieurs comportements. Nous n'allons pas décrire chacune de ces interfaces, car cela a déjà été effectué dans la documentation du code. Nous reviendrons cependant sur certaines d'entre elles au fur et à mesure du détail de la conception.

Les équipements

Pour illustrer l'utilisation de ces interfaces, nous allons maintenant décrire la partie équipements du jeu.



Une classe abstraite mère nommée Équipement permet de décrire le comportement de base d'un équipement. C'est pourquoi, elle implémente 3 interfaces :

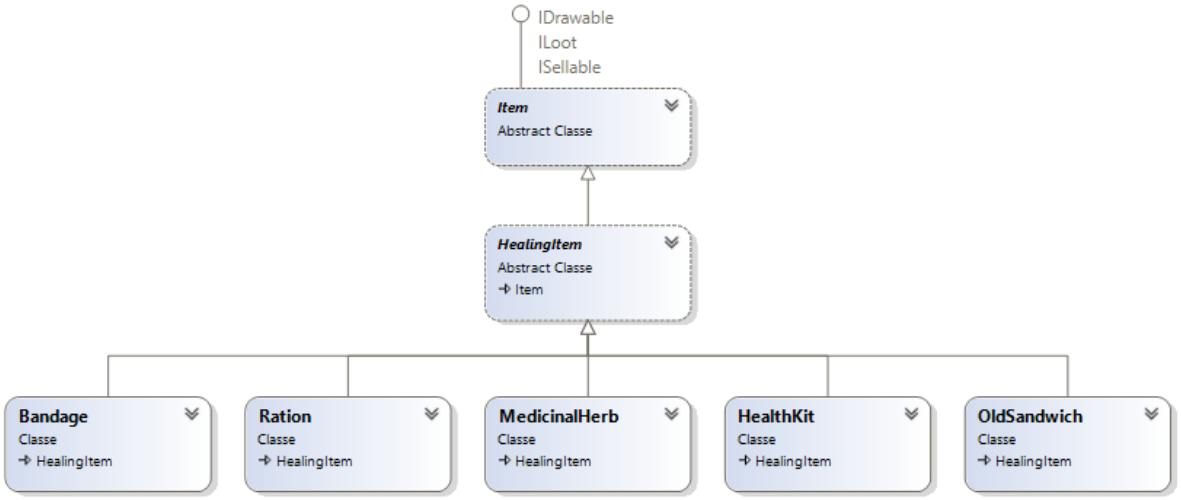
- IDrawable : un équipement est affichable sur la console
- ILoot : un équipement peut être déposé sur le sol dans la map
- ISellable : un équipement peut être vendu par un marchand

Nous avons ensuite créé 2 types d'équipement, les équipements d'attaque (les armes) et les équipements de défense (les armures). Les classes AttackEquipment et DefenseEquipment sont donc également abstraites. Chacune de ces deux classes sont héritées par plusieurs armes ou plusieurs parties d'armures.

Pour les armes, la classe AttackEquipment possède une méthode de base permettant de calculer les cellules de la map concernées par l'attaque en fonction de la portée de l'arme. Chaque arme a donc des statistiques différentes ainsi qu'une portée qui lui est propre. Lorsqu'un personnage attaque avec une arme, c'est alors la méthode générique d'attaque de l'arme qui est appelée grâce à la classe AttackEquipment. Pour les armes très spécifiques comme la DeathScythe qui est une arme de boss qui décrit un comportement très spécifique, la méthode d'attaque est redéfinie.

Les items

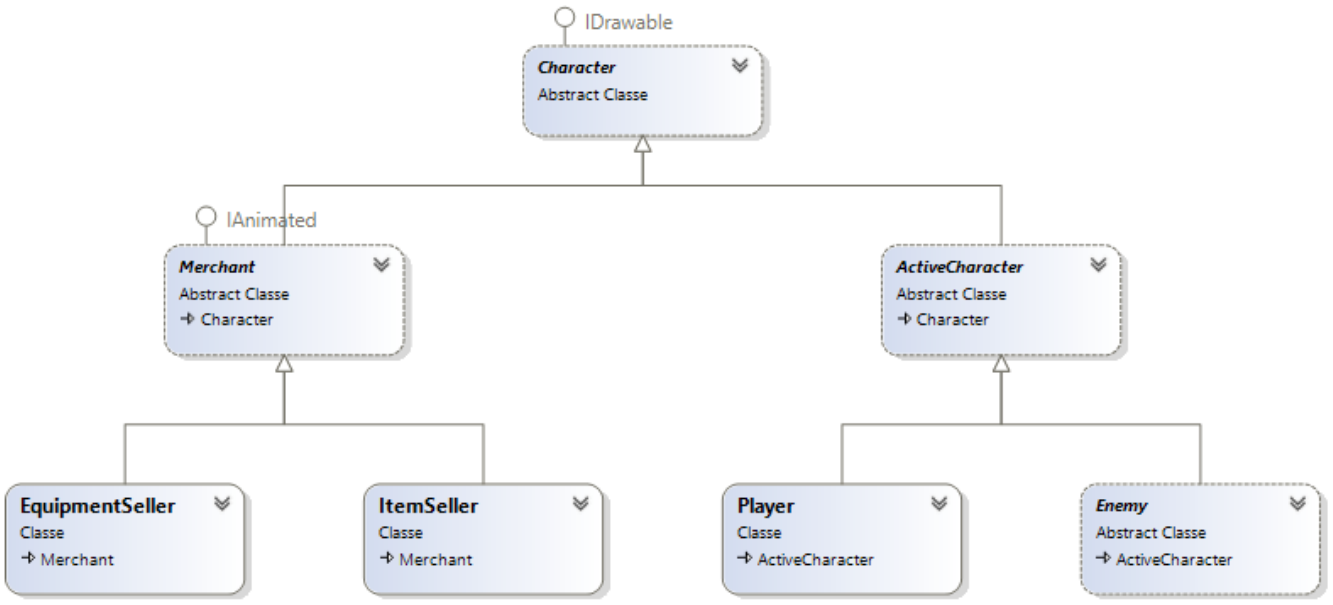
Les items sont des objets utilisables par le joueur.



Une classe Item décrit de façon générale un objet. Tout comme les équipements, les objets peuvent être affichés sur la console, sur le sol de la map et peuvent être vendus par un marchand. Pour l'instant, il n'y a qu'un seul type d'objet : les HealingItem qui permettent au joueur de se soigner. Ceux-ci ont été décrits précédemment dans la présentation des fonctionnalités, nous n'allons donc pas revenir dessus.

Les personnages

Nous allons maintenant montrer comment ces objets et ces équipements peuvent être utilisés.



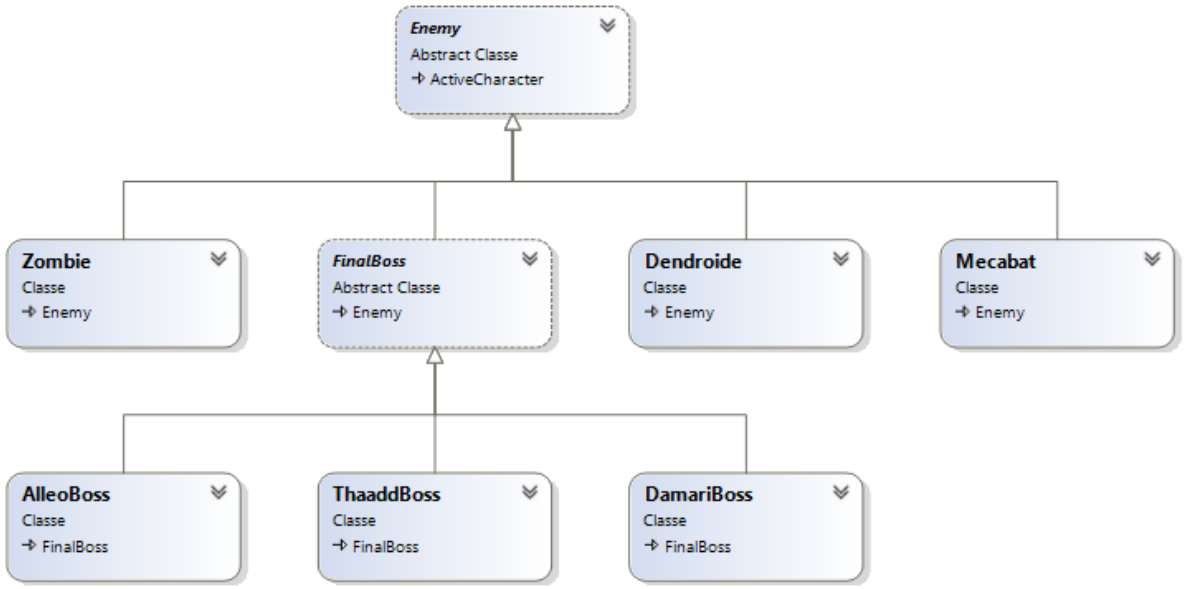
Les personnages sont représentés par une classe abstraite Character qui contient les attributs et propriétés de base d'un personnage (nom, position, couleur, symbole de représentation, etc.). Pour l'instant, il n'y a que deux types de personnages, les ActiveCharacter qui correspondent aux personnages qui peuvent attaquer et se déplacer sur la map, et les marchands qui vendent des objets au joueur. Les personnages actifs sont donc dotés de statistiques supplémentaires tels que des points de vie, de l'attaque, de la défense, etc. Il n'y a pour l'instant que deux types de personnages actifs, le joueur et les ennemis. Nous reviendrons plus précisément sur les ennemis dans la partie suivante.

Le joueur a la possibilité de porter une armure contrairement aux ennemis, il peut récupérer des items ou de l'équipement. En revanche les ennemis n'ont pas d'armure mais peuvent avoir une arme.

L'autre type de personnage correspond aux marchands. Il y a deux types de marchands, les vendeurs d'équipements et les vendeurs d'items. Comme nous l'avons vu précédemment, ces vendeurs sont situés dans le vaisseau du joueur.

Les ennemis

Nous allons maintenant détailler l'implémentation des ennemis dans le jeu.



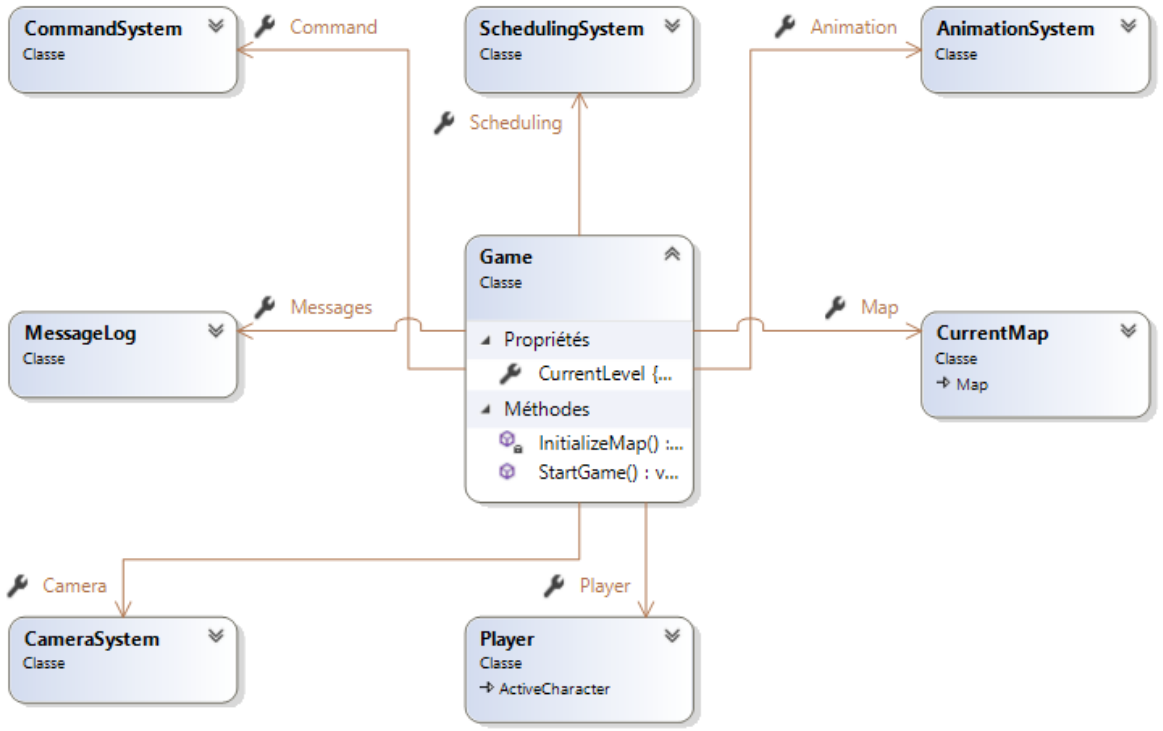
Nous retrouvons la classe abstraite Enemy qui a été évoquée précédemment. Les enemis sont donc des personnages actifs dotés de statistiques spécifiques. Par exemple, la Mecabat sera plus rapide qu'un Zombie. Il y a cependant un type particulier d'ennemi : les FinalBoss. Ce sont les boss finaux des planètes. Il y a donc 3 boss qui ont 3 armes différentes. La classe FinalBoss permet notamment de gérer la mort des boss. Quel que soit le boss, sa mort est équivalente mais diffère des autres ennemis. Afin d'éviter la duplication de code, nous avons donc créé cette classe FinalBoss.

4.2 Interactions entre les différentes composantes

Maintenant que nous avons décrit la structure des principales composantes, nous allons nous concentrer sur les interactions entre ces classes.

4.2.1 La classe Game

La classe Game contient tous les éléments qui constituent le jeu, mis à part la partie View. En effet, le jeu est une propriété de la classe ScreenView comme nous l'avons vu précédemment.



La classe Game contient plusieurs éléments essentiels au jeu :

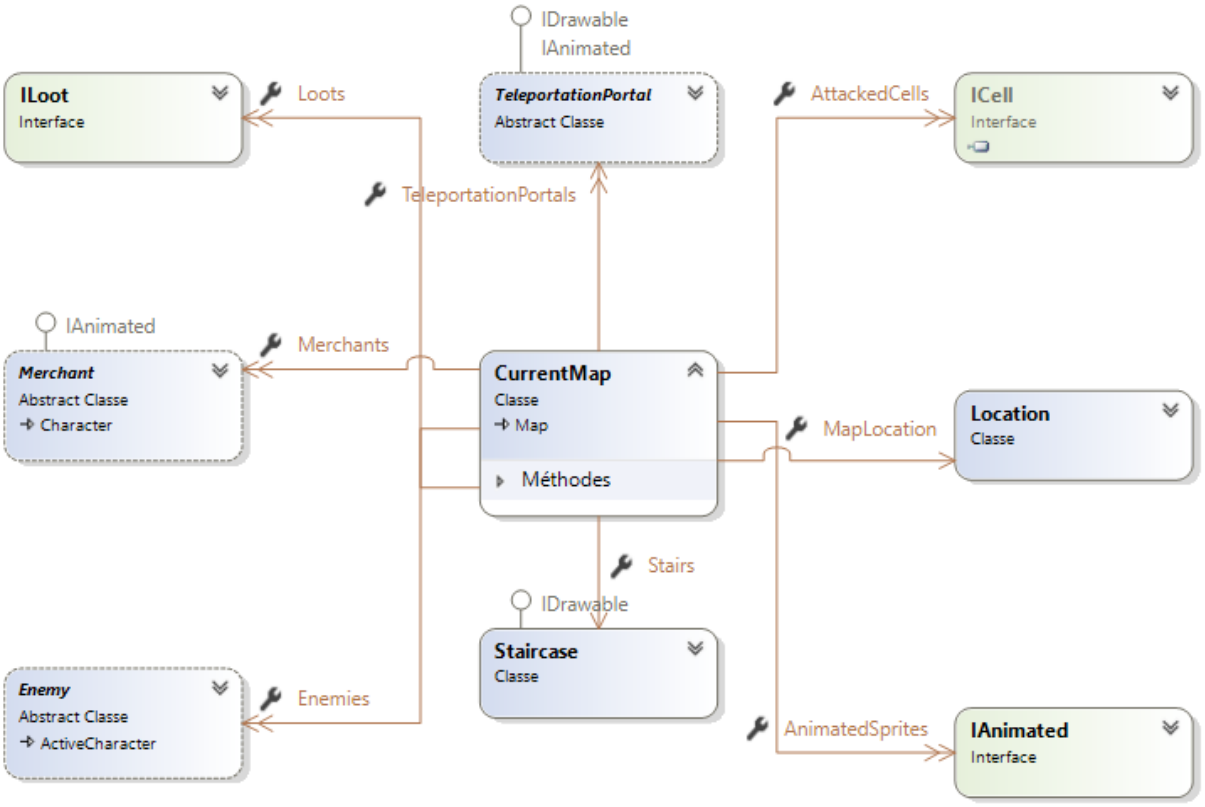
- Player : Le joueur
- CurrentMap : La map sur laquelle le joueur est situé actuellement
- AnimationSystem : Le mécanisme qui permet d'animer des éléments de la map périodiquement
- SchedulingSystem : Le mécanisme qui permet de faire faire des actions aux personnages non jouables tels que les ennemis
- CommandSystem : L'interface entre les personnages actifs et le jeu, cette classe permet de faire réaliser des actions aux personnages actifs sur le jeu.
- MessageLog : Permet d'informer le joueur sur l'état actuel du jeu, et de son avancement
- CamerSystem : Le mécanisme qui permet de faire suivre la caméra sur le joueur pour faire une "scrolling map"

Comme on peut le voir dans les méthodes de la classe Game, cette classe n'est utilisée que lors de la création d'une partie. En effet, cette classe sert principalement d'agrégateur afin de regrouper tous les éléments essentiels au jeu. La boucle de jeu étant dans la partie View,

il est très utile d'avoir une classe qui regroupe le joueur, la map et les mécanismes indispensables au bon déroulement d'une partie.

4.2.2 La classe CurrentMap

Cette classe représente la map sur laquelle le joueur est actuellement situé.



Nous considérons la map comme la partie la plus importante du code, car toutes les fonctionnalités développées se retrouvent dans les associations de cette classe.
On retrouve notamment :

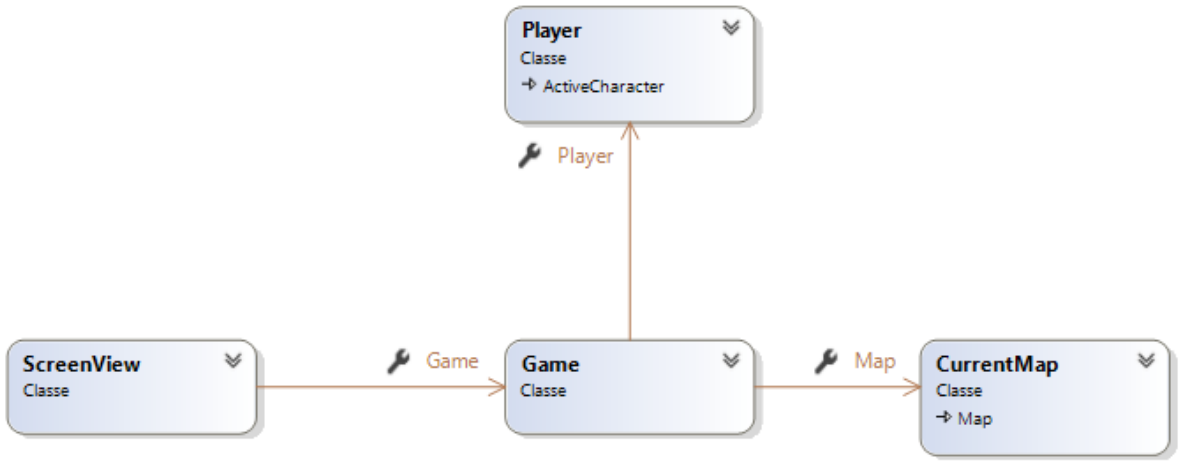
- AnimatedSprites : La liste de tous les éléments animés sur la map, cela peut être des ennemis ou des portails de téléportations
- MapLocation : Les informations sur la localisation de la map, notamment le type de map et le type de planète
- AttackedCells : La liste des cellules de la map qui sont attaquées par un personnage
- TeleportationPortals : La liste des portails de téléportations présents sur la map
- Loots : La liste de tous les objets présents sur le sol de la map
- Merchants : La liste des marchands présents dans la map

- Enemies : La liste des ennemis présents dans la map

Cette classe est très générale, elle peut être une salle de boss, le vaisseau spatial ou n'importe quel étage d'une des 3 planètes. C'est pourquoi son nom est CurrentMap, elle correspond réellement à la map sur laquelle le joueur est située à un moment précis. Lorsque le joueur change de map à l'aide d'un portail de téléportation, un appel est fait la classe MapCreation depuis la partie View, afin de modifier la propriété Map de la classe Game. Cela permet donc de changer la map. Pour plus d'informations sur la classe MapCreation, vous pouvez vous référer à la documentation du code.

4.2.3 Synthèse des interactions entre les classes

Maintenant que nous avons détaillé les principales composantes du code ainsi que les interactions entre certaines classes de ces composantes, nous pouvons synthétiser tout cela de la sorte :



Le squelette de notre projet réside dans l'interaction entre ces 4 classes. Le ScreenView possède une instance de Game, et Game possède une instance du joueur (Player), et une instance de la map (CurrentMap).
Les boucles de jeu étant présentes dans ScreenView et ses sous classes, l'instance de Game permet d'avoir accès à tous les éléments du jeu, dont notamment le joueur et la map.

4.3 Justification de l'utilisation de librairies externes

Ayant tous les deux de l'expérience dans le développement (DUT informatique) nous avons fait le choix de ne pas partir de zéro. Roguesharp nous permet d'effectuer de la génération procédurale, ainsi que de gérer une map avec un système de champ de vision. Des méthodes sont également offertes pour réaliser du pathfinding, notamment avec l'algorithme A*. De plus, Roguesharp a été développé pour s'intégrer parfaitement avec la librairie RLNET. RLNET permet d'avoir une console composée de cellules sur lesquelles ont peut écrire des caractères grâce à un "font file" qui est une image contenant tous les caractères pouvant être affichés sur la console.

Utiliser ces librairies nous a permis de gagner beaucoup de temps sur la gestion de la map, et de ne pas ré-écrire des algorithmes que nous connaissions déjà. Nous nous sommes cependant renseignés sur la génération procédurale, algorithme que nous avons tous deux découverts lors de ce projet. Tout cela nous a permis d'implémenter des fonctionnalités plus avancées, comme le système d'achat d'objets, de combat, ou de voyage entre différentes planètes.

5. Tests réalisés

Pour tester le code, dès qu'une fonction était écrite, son bon fonctionnement était évalué à l'aide d'affichage d'informations dans la console. Nous avons cependant rédigé un plan de test pour évaluer le bon fonctionnement de chacune de nos fonctionnalités. Les résultats de ces tests sont présents en annexes, ainsi qu'en .xlsx dans le répertoire de rendu.

6. Bilan

Pour conclure, nous sommes très satisfaits du travail que nous avons pu réaliser au cours de ce projet de programmation. Cependant, nous n'avons pas eu le temps de développer le gameplay comme nous le souhaitions initialement avec des aides venant de personnages non jouables comme un drone. Les énigmes que nous avions imaginées utilisaient les interactions avec ces personnages non jouables, c'est pourquoi nous n'avons pas pu en inclure dans l'état actuel du jeu.

On peut également dégager plusieurs pistes d'améliorations. Pour améliorer la durée de vie du jeu, nous aurions pu avec plus de temps ajouter un tableau des scores à l'aide d'un fichier

de sauvegarde des temps. Cette fonctionnalité était prévue, c'est pourquoi le temps de jeu est compté, et affiché sur l'écran de victoire.

Concernant l'interface, nous aurions également aimé avoir des aides de jeux ou des boîtes de dialogue contextuelles, c'est-à-dire qu'elles ne s'afficheraient que lorsque le joueur se trouve proche de certains éléments du jeu comme par exemple un portail de téléportation ou un marchand. Nous aurions également aimé avoir une minimap sur l'interface pour informer le joueur de sa position, mais après plusieurs tentatives, ce ne fut pas concluant et le temps passé à réaliser cette minimap commençait à empiéter sur le bon avancement du projet.

Concernant le gameplay, nous aurions aimé faire en sorte que le joueur puisse posséder plusieurs armes, et échanger entre elles durant les combats, pour s'adapter à certaines situations difficiles. Comme évoqué précédemment, nous aurions également aimé ajouter des personnages non jouables permettant d'aider le personnage ce qui aurait pu alimenter le gameplay.

7. Annexes

7.1 Tests des fonctionnalités

Test des différentes fonctionr

Situation	Se déplacer	Acheter un objet si possible	Se téléporter	Acheter un obje	Tuer un boss	Récupere un artifact	Changer d'étage dans une map
Le joueur ne possède pas d'équipement et pas d'objet	OK	OK	OK	OK	OK	OK	OK
Le joueur possède des équipements ou des objets	OK	OK	OK	OK	OK	OK	OK
Le jouer a récolté 0 artefacts	OK	OK	OK	OK	OK	OK	OK
Le joueur a récolté 1 ou 2 artefacts	OK	OK	OK	OK	OK	OK	OK
Le joueur a récolté 3 artefacts	OK	OK	OK	OK	OK	OK	OK
Le joueur est dans le vaisseau spatial	OK	OK	OK	OK	Pas possible	Pas possible	Pas possible
Le joueur est dans une salle de boss	OK	Pas possible	OK	Pas possible	OK	OK	Pas possible

nalités

Fonctionnalités					
Remplacement de pièce d'équipement	Attaquer avec les 5 armes différentes	Gagner	Perdre	Récuperer l'argent des enemis	Récuperer l'arme d'un boss
OK	OK	OK (difficile)	OK	OK	OK
OK	OK	OK	OK	OK	OK
OK	OK	Pas possible	OK	OK	OK
OK	OK	Pas possible	OK	OK	OK
OK	OK	OK	OK	OK	OK
OK	OK	Pas possible	Pas possible	Pas possible	Pas possible
OK	OK	OK	OK	OK	OK

7.2 Diagramme de classe

Un diagramme de classe complet généré par Visual Studio est disponible au format png dans ce répertoire.