Projet Blast

Rapport de projet

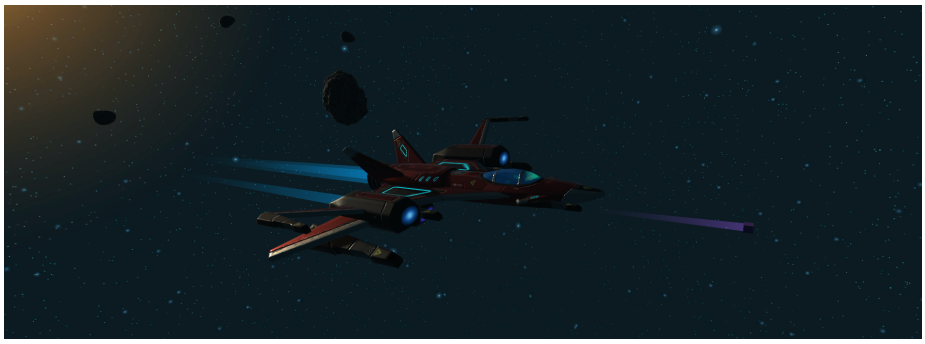
— Groupe Quadro —

Nicolas "g00pix" FROGER
Mathieu "Matiboux" GUÉRIN
Pierre "Rokka" DE LA RUFFIE

17 mai 2019

Table des matières

1 Introduction



Après un semestre entier de travail de groupe, le projet **Blast** touche enfin à sa fin.

Malgré les difficultés rencontrées et une première partie de développement compliqués, notamment au sujet de la formation du groupe, beaucoup de travail a été effectué pour pouvoir atteindre ce stade final de notre jeu vidéo.

1.1 Présentation générale du jeu

Blast est un jeu de tir à la troisième personne dans l'espace. Le joueur contrôle un vaisseau voyageant dans l'espace et doit effectuer plusieurs missions qu'il peut choisir. Il en existe deux : la livraison et la destruction. Tout ses aspects seront développés dans ce rapport.

Le projet **Blast** a été développé en *C#*[1] et en utilisant le moteur de jeu *Unity*[2] (version 2018.3.5f). N'étant initialement pas familier avec ce moteur de jeu, il nous a fallu un certain temps d'adaptation afin de comprendre ses mécaniques et ses spécificités. Concernant la centralisation des données, nous avons préféré utiliser *Git* et *Github* plutôt que *Unity Collab*, son analogue prévu pour fonctionner uniquement avec *Unity*, puisque certains d'entre nous connaissaient déjà cet outil, très populaire dans le domaine du développement, et qu'il offre plus de possibilités que son alternative.

Avec le jeu est aussi fourni un site, hébergé grâce à *GitHub*[3], et contenant tout d'abord un lien de téléchargement du jeu mais aussi quelques informations.

— Site : `https://g00pix.github.io/ProjetBlast/`
— Page des téléchargements : `https://g00pix.github.io/ProjetBlast/downloads`

1. *C#* : `https://fr.wikipedia.org/wiki/C_sharp`
2. *Unity* : `https://unity.com/`
3. *GitHub* : `https://github.com/`

2 Rappel du cahier des charges

2.1 Modifications de groupe

Le groupe a subi de nombreuses modifications depuis sa création. Ci-dessous est présenté l'historique de cette évolution (avec en gris les départs) :

- Groupe d'origine
  - Nicolas FROGER
  - Mathieu GUÉRIN
  - Maëlle FERRARIN
  - Adonis KHALIFE
- 1ère soutenance
  - Nicolas FROGER
  - Mathieu GUÉRIN
  - Mattéo DEMICHELE
- 2nde soutenance
  - Nicolas FROGER
  - Mathieu GUÉRIN
  - Pierre DE LA RUFFIE
  - Clément PERICAT

La période entre la 1ère et la 2nde soutenance a été marquée par la fusion de deux groupes avec deux projets différents :

- Blast, développé par le groupe **Quadro** :
  - Nicolas FROGER
  - Mathieu GUÉRIN
- Blitzkrieg, développé par le groupe **Overlord** :
  - Pierre DE LA RUFFIE
  - Clément PERICAT

Après concertation, le projet retenu, comme la première page du rapport le laisse savoir, a été **Blast**.

Aujourd'hui, le groupe **Quadro** est constitué des 3 personnes suivantes :
- Nicolas FROGER (chef de projet)
- Mathieu GUÉRIN
- Pierre DE LA RUFFIE

2.2 Mises à jour de la répartition des tâches

Avant la première soutenance, la répartition des tâches a été un peu plus difficile que prévue à cause des départs de membres et était donc un peu floue, mais les bases du jeu ont néanmoins pu être commencées.

Suite à la fusion des deux groupes après la première soutenance, il a fallu choisir lequel des deux projets allait être continué et lequel allait être laissé de côté. Le projet **Blast** a été retenu et c'est donc le développement de celui-ci qui est poursuivi. Il a donc été nécessaire au nouveau membre de s'approprier le jeu et se mettre au courant de ce qui avait déjà été fait pour être en mesure de commencer à ajouter des fonctionnalités.

L'arrivée de Pierre lors de cette fusion a nécessité une mise à jour de la répartition des tâches. Celle-ci a été adaptée à ses compétences et n'a pas posé de problèmes.

Chacun des membres s'est intégré au projet et s'est attribué des rôles en rapport avec le développement de notre jeu vidéo :

— Nicolas : Développement multi-joueur, missions et upgrades.
— Mathieu : Développement objets, robot ennemis et gameplay.
— Pierre : Création et développement interface, audio et map.

2.3 Mise à jour des fonctionnalités

En ce qui concerne les fonctionnalités prévues par le cahier des charges, peu de modifications ont été effectuées.
En effet, le seul changement concerne la section "Objets du jeu" qui a été renommée en "Améliorations du vaisseau" pour mieux refléter l'évolution du projet.

3 Les fonctionnalités du jeu

Chaque fonctionnalité du projet est détaillée ci-dessous dans leurs sections respectives. Ces sections correspondent aux catégories de fonctionnalités que nous avions mentionnées dans le cahier des charges. Pour chaque section, le développement de chaque fonctionnalité, ainsi que tous les aspects qui la composent, seront détaillés. Pour chaque fonctionnalité qui s'y prête, nous préciseront leur fonctionnement et leurs enjeux techniques.

3.1 Le vaisseau

Le vaisseau est l'élément qui représente le joueur. Chaque vaisseau dispose d'un certain nombre de fonctionnalités qui sont détaillées ci-dessous.

3.1.1 Contrôles

Le vaisseau est un objet dans l'espace que le joueur peut contrôler. Une caméra y est virtuellement attachée, à la troisième personne, et permet au joueur de se voir et ainsi de se déplacer dans le jeu.
Les mouvements de ce dernier ont été configurés dès le début du projet et améliorés par la suite. Le joueur est ainsi capable de contrôler :

— L'accélération et les mouvements du vaisseau (avec Z Q S D)
— Le roulis du vaisseau (avec A E)
— La direction du vaisseau (avec les mouvements de souris)

Un *boost* est également disponible au joueur et lui permet de multiplier par 10 son accélération et sa vitesse maximale. Pour activer le boost, le joueur doit maintenir Shift ⇑ pendant son déplacement.

L'accélération est d'ailleurs progressive : c'est à dire que le joueur mettra un certain temps à atteindre une vitesse rapide, et encore plus pour atteindre sa vitesse maximale. Cette vitesse maximale est de 19.6 m/s en mode normal, et de 196 m/s en mode *boost*.

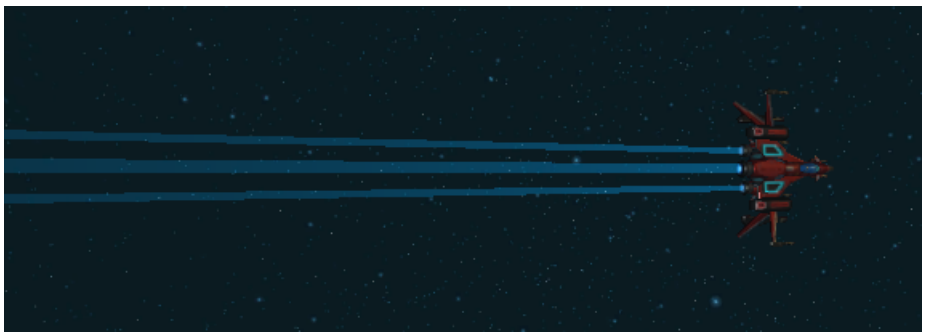
Le joueur est aussi capable de regarder ce qu'il se passe autour de lui sans modifier la direction de son vaisseau : il s'agit du mode « caméra libre ». Pour passer dans ce mode, il suffit au joueur d'appuyer sur le *clic molette* avant de déplacer sa souris.

3.1.2 Mouvements du joueur

Les mouvements ont été compliqués à bien gérer. En effet, le jeu se déroule dans l'espace, c'est-à-dire un environnement sans gravité, et donc sans notion de haut et de bas. Cela représente une vraie problématique car il ne faut pas que le joueur se perde.
Il faut aussi que les contrôles se fassent relativement au joueur et non au monde, ce qui a posé beaucoup de problèmes au début car la caméra et le vaisseau ne réagissaient pas de la manière souhaitée lorsque ce dernier était incliné et a nécessité beaucoup de recherches pour les régler.

Les solutions ont cependant été assez simples, malgré que le tout ait été résolu assez tardivement dans le développement du jeu. Pour garder l'exemple de la caméra, nous utilisons la fonction `Transform.LookAt()` [4] pour que la caméra regarde le vaisseau du joueur. Lorsque le joueur était incliné ou était à l'envers, la caméra restait orientée de la même façon et les contrôles devenaient très confus. Nous avons appris par la suite que cette fonction dispose d'un paramètre permettant de choisir le vecteur qui représente le haut, par défaut fixé à `Vector3.up` [5], il a donc suffit d'utiliser le vecteur haut relatif au joueur (`player.transform.up`) pour régler ce problème.

De plus, des **effets de traînées** ont été ajoutés à l'arrière du vaisseau, sur ses réacteur, pour donner au joueur une réelle impression de mouvement, que le vide de l'espace seul ne permet pas.
Des effets similaires ont été ajoutés aux projectiles pour leur donner aussi un effet de vitesse, mais aussi de puissance. Ces deux traînées utilisent les composants *Trail Emitters* de Unity.



4. `Transform.LookAt()` :
`https://docs.unity3d.com/ScriptReference/Transform.LookAt.html`
5. `Vector3.up` : Correspond au vecteur `(0, 1, 0)`.
Plus d'infos sur `https://docs.unity3d.com/ScriptReference/Vector3-up.html`

3.1.3 Mécaniques de tir

Des mécaniques de base pour **le combat** ont ensuite été ajoutées par Mathieu. Un premier canon a été ajouté, puis un second, tous deux capables de tirer sur demande **des projectiles**, à l'aide du clic gauche et droit pour, respectivement, le canon gauche et droit. Ces projectiles tirés iront heurter les objets qu'ils rencontreront, avant de disparaître à l'impact. Les collisions sont détectées à l'aide de la méthode prédéfinie `OnCollisionEnter()` [6] et supprime le projectile lorsqu'elle est appelée. Si aucune collision n'est détectée, le projectile est supprimé au bout d'un certain temps (environ 5 secondes).

La mécanique de tir a ensuite été mise à jour avec la possibilité de charger les canons avant le tir, et ce de manière individuelle. Ainsi, maintenir le clic gauche ou droite démarrera, respectivement, la charge de l'arme gauche ou droite. Ce chargement est perçu par le joueur à l'aide d'une orbe translucide qui augmente progressivement de taille au bout du canon. Au bout d'un court instant, des particules apparaîtront en plus pour rendre l'effet plus visible au joueur. On peut voir l'effet au bout des deux canons du vaisseau sur l'image ci-dessous.

Pour résoudre ensuite un problème pratique de jouabilité, une aide à la visée avec un **viseur dynamique** a été ajouté à l'interface permettant au joueur d'avoir une plus haute précision dans l'utilisation de ses armes.

6. `OnCollisionEnter()` : `https://docs.unity3d.com/ScriptReference/Collider.OnCollisionEnter.html`

Le chargement des armes du vaisseau du joueur peut être utile à ce dernier puisqu'il a des effets sur les dégâts et sur la vitesse des projectiles. Le chargement devient maximal au bout de 2 secondes, c'est à dire qu'au delà de 2 secondes, les effets seront les mêmes. Le chargement maximal permet, au tir, la propulsion de projectiles avec les effets suivants qui leur sont appliqués :

— Les projectiles font **3 fois** plus de dégâts aux ennemis ; et
— Les projectiles se déplacent **2 fois** plus rapidement.

3.1.4 Physique des projectiles

Tout d'abord, les tirs n'étaient que de simples cubes sans effets. Puis, nous avons opté pour des tirs plus graphiques, avec un *trail emitter* derrière, de la même manière que pour le vaisseau. La vitesse de propulsion des projectiles était auparavant gérée indépendamment de celle du vaisseau, ce qui causait parfois le phénomène absurde que le vaisseau allait plus vite que le projectile qu'il tire devant lui. Ce problème a été corrigé en ajoutant la vitesse du vaisseau à la vitesse de propulsion du projectile.

De plus, l'ajout des tirs chargés a nécessité des améliorations allant de pairs avec celles-ci. À présent, la masse des tirs est proportionnelle à la puissance de ceux-ci. Cela peut sembler assez anodin mais toute la physique de Unity se base sur la masse. Par exemple, un tir d'une masse nulle n'aura aucune conséquence sur la trajectoire d'un astéroïde qui la percute. Par contre, un tir avec une masse importante aura tendance à repousser l'astéroïde, ou encore à influencer sur sa vitesse ou son angle de rotation.

Les tirs chargés ont aussi une vitesse plus importante.

3.1.5 Améliorations

Cette partie était anciennement dénommée "les objets" car nous avions en idée que les objets seraient utilisés pour améliorer le vaisseau. Cependant, nous avons préféré faire un système d'amélioration de vaisseaux plus simple. À l'aide des points d'expériences pouvant être acquis dans les missions, il est désormais possible "d'acheter" des améliorations. L'amélioration est appliquée au vaisseau après l'échange des points. Les améliorations implémentées concernent l'accélération du vaisseau, les points de vie du joueur et les dégâts qu'infligent les armes du vaisseau. Chacune d'entre elles dispose de trois niveau d'amélioration différents.

3.1.6 Interactions entre entités

Au début du projet, nous avions introduit les points de vie pour les vaisseaux, mais ces points ne pouvaient pas évoluer. Nous avons par la suite ajouté un nouvel aspect au vaisseau qui le rend destructible. Une collision entre un objet quelconque et le vaisseau fera baisser son niveau de vie. Cela fonctionne notamment avec les armes équipées sur les vaisseaux. Il est donc possible de tuer un autre joueur dans une partie multijoueur à l'aide de ses armes.

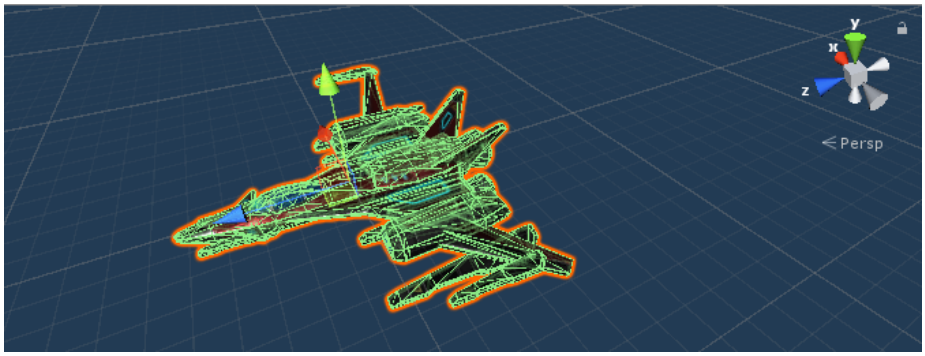
Cette fonctionnalité a été étendue à l'aide d'une *interface C#*[7]. Cette notion, abordée en travaux pratiques à l'EPITA, permet de définir des méthodes qui sont communes à différents objets. Dans ce cas précis, l'*interface C#* (nommée `ILivingEntities`) définit des méthodes communes à toutes les entités "vivantes", c'est-à-dire des objets possédant un niveau de vie qui peut baisser au cours de la partie jusqu'à zéro, ce qui engendre la mort de l'entité. Les joueurs sont considérés comme des entités vivantes et implémentent donc cette *interface*.

7. *Interface C#* :
`https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/interface`

3.1.7 Collisions

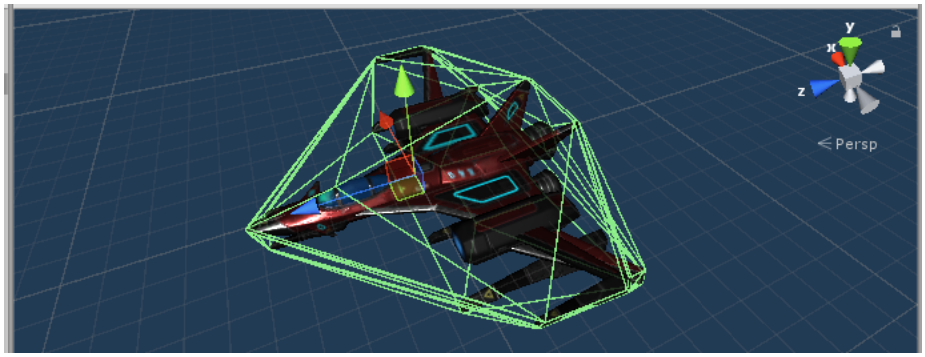
Initialement, les collision étaient gérées grâce à des formes géométriques simples : un pavé pour les vaisseaux et une sphère pour les astéroïdes. Cette technique avait ses avantages, mais aussi beaucoup d'inconvénients. Cela améliorait grandement la vitesse de calcul des collisions bien-sûr, mais diminuait aussi sa précision. On se retrouvait alors avec un vaisseau qui se cognait avec des astéroïdes à quelques mètres de lui. Sachant que les collisions retirent de la vie, il était important d'avoir une boîte de collision plus précise afin de pouvoir éviter ces astéroïdes facilement. De plus, les astéroïdes n'ont pas une forme totalement sphérique et le vaisseau pouvait rentrer dans certaines aspérités de la surface lorsque les astéroïdes étaient plus gros que la normale.

C'est pourquoi le projet a subi une refonte totale du système de collision. Celles-ci ne sont maintenant plus gérées par des *BoxCollider*[8] ou des *SphereCollider*[9], mais par des *MeshCollider*[10]. En utilisant les Mesh[11] (qui étaient fournies dans les différentes librairies d'où viennent nos modèles 3D) nous avons été capable, grâce à Unity, d'obtenir une boîte de collision complexe que voici :



8. *BoxCollider* : `https://docs.unity3d.com/ScriptReference/BoxCollider.html`
9. *SphereCollider* : `https://docs.unity3d.com/ScriptReference/SphereCollider.html`
10. *MeshCollider* : `https://docs.unity3d.com/ScriptReference/MeshCollider.html`
11. Un *mesh* ou maillage est un objet tridimensionnel constitué de sommets, d'arêtes et de faces organisés en polygones sous forme de fil de fer dans une infographie tridimensionnelle. Les faces se composent généralement de triangles, de quadrilatères ou d'autres polygones convexes simples, car cela simplifie le rendu. Les faces peuvent être combinées pour former des polygones concaves plus complexes, ou des polygones avec des trous (il s'agit, pour résumer, de l'aspect physique d'un objet). — `https://fr.wikipedia.org/wiki/Mesh_(objet)`

Cependant, ce *MeshCollider* ne sera pas capable d'intéragir avec d'autres *MeshCollider*. Cela pause un problème sachant que les astéroïdes aussi possèdent cette spécificité. Ils faut donc que les collider deviennent *Convex*[12] : les *Convex MeshCollider* possèdent une limite de 255 triangles, ce qui explique la diminution de la précision de la version finale de la boîte de collision du vaisseau (il en va de même pour les astéroïdes). Celle-ci reste tout de même suffisante pour ce projet :



12. *Convex* : `https://docs.unity3d.com/ScriptReference/MeshCollider-convex.html`

3.2 Les missions

Afin d'apporter un but au jeu, un système de mission a été implémenté. Il permet au joueur d'avoir une progression et des objectifs à accomplir. Les missions permettent d'obtenir des points d'expérience permettant entre autre l'amélioration du vaisseau.

Les missions sont implémentées à l'aide de classes abstraites [13]. Cette notion a également été vue en travaux pratiques à l'EPITA. Ce type de classe permet de définir une classe partielle, dans ce cas *Mission*, qui doit être étendue. Dans notre cas, cela nous permet de développer différent types de missions, tout en gardant des propriétés et méthodes communes, ce qui évite de gérer tous les types manuellement pour faire la même chose. Tous les types de missions partagent certaines propriétés, comme le nombre de points d'expérience qu'elles donnent au joueur quand celui-ci la finit. Les missions disposent toutes de mêmes méthodes comme la méthode *Begin()* qui correspond au démarrage de la mission.

3.2.1 Mission de livraison

Le premier type de mission implémenté est la mission de livraison. Le joueur doit se rendre dans une zone pour récupérer un colis et doit l'amener dans une autre zone. Les positions des zones sont déterminées aléatoirement tout en restant dans des distances atteignables par le joueur.

3.2.2 Mission de destruction

Le second type de mission implémenté est la mission de destruction. Un bâtiment apparaît sur la map et le joueur doit s'y rendre afin de détruire le bâtiment avant que la tourelle présente au dessus ne le détruise. La position est déterminée comme celle de la mission de livraison.

Ces deux types de missions utilisent des zones de début et de fin représentées par des bâtiments. Ces derniers seront vus plus en détail dans leur partie dédiée.

13. *Classes abstraites* :
`https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/abstract`

3.3 Interfaces

L'interface est gardée volontairement simple pour permettre au joueur de garder une bonne visibilité dans le jeu. Plusieurs éléments ont étés implémentés au fur et à mesure de la progression du jeu, c'est pourquoi nous allons les détailler dans leur ordre chronologique.

3.3.1 La boussole

Pour permettre au joueur de se repérer dans l'espace, **une boussole** a été ajoutée à l'interface. Elle se met à jour selon les mouvement et la position du joueur. La boussole est constituée d'un indicateur qui indique la valeur actuelle et la boussole en elle-même en dessous. Il s'agit d'une très grande image de près de 8000 pixels de largeur qui contient toutes les valeurs de 0 à 350. Cette image est partiellement masquée dans Unity à l'aide du composant *Mask*[14] et est déplacée en fonction de l'angle du joueur entre un vecteur arbitraire qui représente le nord (arbitraire car il n'y a pas de nord dans l'espace) et l'orientation du vaisseau. L'image fait une boucle car en arrivant à la toute droite de celle-ci on retourne au début, c'est-à-dire à gauche. Cette boussole est utile pour se rendre quelque part sur la carte et sert d'indicateur de direction.

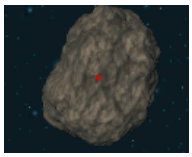
3.3.2 HUD (Head-Up Display)

Afin de pouvoir rajouter des caractéristiques au vaisseau ainsi que d'informer le joueur de son statut, nous avons ajouté l'affichage d'informations du vaisseau sur le bord inférieur gauche de l'écran. La santé et la vitesse du vaisseau du joueur y sont affichés. La vitesse est donnée en mètres par seconde, 1 mètre équivalent à une unité dans le monde en 3D.

14. *Mask* : `https://docs.unity3d.com/ScriptReference/UI.Mask.html`

3.3.3 Le viseur

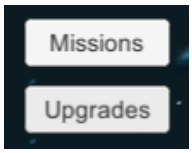
Pour résoudre un problème de difficulté à la visée, nous avons ajouté un **viseur dynamique** à l'interface. Ce dernier est affiché quand le vaisseau pointe en direction d'un objet dans l'espace. Cela est rendu possible grâce au tracé de rayons virtuels avec la méthode `Physics.Raycast()` [15]. Cette fonction retourne le point d'impact avec un objet, s'il y en a un. Le point d'impact est ensuite projeté sur l'interface en deux dimensions, ce qui nous donne la position à jour du viseur, qui se placera aux coordonnées correspondantes.



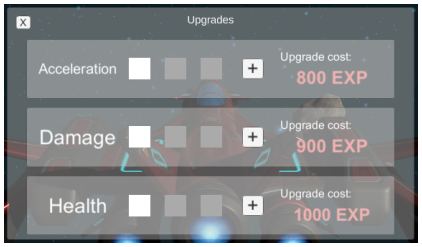
Dans le cas où le rayon virtuel ne rencontre aucun objet, la fonction retourne faux et le viseur est masqué jusqu'à ce que le joueur dirige son vaisseau vers un nouvel objet.

3.3.4 Missions et améliorations

Pour la deuxième soutenance et l'introduction des missions et des améliorations, il a fallu que l'interface soit adaptée à ces nouvelles additions.



C'est pourquoi l'interface de jeu possède maintenant deux boutons permettant d'accéder aux deux fenêtres *Upgrades* et *Missions*, montrées ci-dessous.



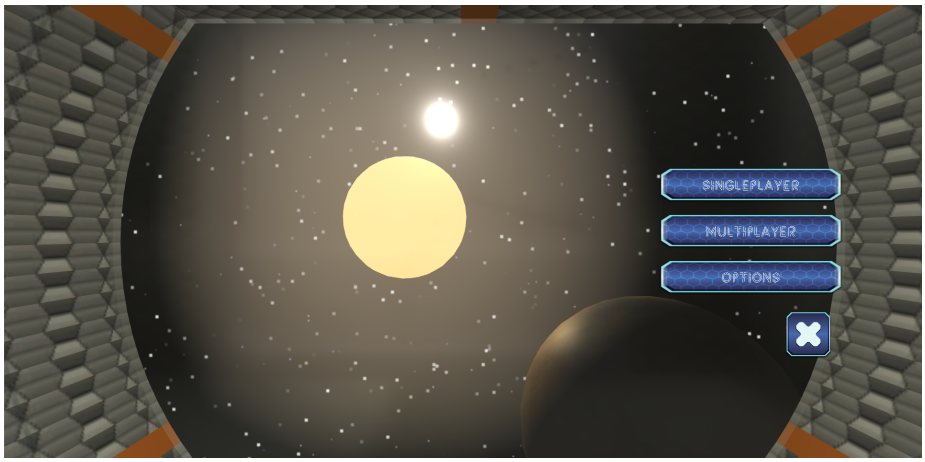
Sur l'image, on peut voir à gauche la fenêtre des *upgrades* et à droite la fenêtre des *missions*. Les caractéristiques des *missions* et des *upgrades* seront expliquées dans les sections suivantes.

Nous avons aussi ajouté un menu de pause, accessible en appuyant sur la touche Esc. Celui-ci permet de mettre le jeu en pause pour le reprendre ensuite, ou le quitter.

15. `Physics.Raycast()` : `https://docs.unity3d.com/ScriptReference/Physics.Raycast.html`

3.3.5 Menu principal

Enfin, l'évolution du projet a aussi nécessité une remise en forme du menu. Pour la première partie de développement, le menu n'était composé que d'un bouton "Play". Pour la seconde partie, nous avions créé une scène entière, se déroulant dans une station spatiale avec chaque sous-menu qui correspondaient à un angle de caméra et un endroit de la station différent. Voici à quoi ressemblait le menu principal :



Pour le menu final, la structure du menu en général est restée la même (*Singleplayer et Multiplayer*), avec quelques ajouts comme un menu d'option ainsi que la possibilité de choisir un pseudonyme avant de rejoindre une partie multijoueur. De plus, nous avons ajouté quelques améliorations visuelles. Tout d'abord, le vaisseau du joueur au premier plan, ainsi qu'une refonte des textures de la station. Ensuite, nous avons retiré le soleil, et ajoutés des astéroïdes dans le fond (les astéroïdes tournent).

3.4 Sauvegardes

Avant la deuxième soutenance, la gestion des sauvegardes avait été laissée de côté par manque de pertinence par rapport à ce que nous avions alors développé.

Depuis, avec l'ajout des missions et des améliorations, la gestion des sauvegardes est devenu une fonctionnalité importante à implémenter dans notre jeu vidéo. Par simplicité, nous avons choisi de gérer les sauvegardes avec des fichiers *JSON*[16].

save.json | README.md

```
1  {"playerName":"Mati","playerXp":400,"playerX":0,
   "playerY":0,"playerZ":0,"playerHealth":100,
   "addonAcceleration":1,"addonDamage":1,
   "addonHealth":1}
```

Le système de sauvegarde permet donc au joueur de sauvegarder son expérience ainsi que les attributs de son vaisseau. Ces attributs sont vastes et comprennent les améliorations appliquées par le joueur (attributs préfixés par "*addon*"), ainsi que :

- le nom du joueur (playerName) ;
- sa position (playerX, playerY, playerZ) ;
- sa rotation (playerRX, playerRY, playerRZ) ;
- son niveau de vie (playerHealth) ;
- son niveau d'expérience (playerXp)

À la deuxième soutenance, le chargement de la sauvegarde, si elle existe, était fait automatiquement au lancement d'une partie multijoueur ou solo. La sauvegarde était manuelle et se faisait en appuyant sur la touche O .

Aujourd'hui, la sauvegarde est chargée au lancement du jeu lui-même, ce qui permet de charger le nom du joueur et d'éviter de le lui demander à nouveau. La sauvegarde est encore manuelle et peut se faire à partir du menu de pause, en plus de la touche attribuée O .

Il est toujours possible d'effacer la sauvegarde existante de son ordinateur en appuyant sur la touche L . Alors, au prochain lancement, le jeu se comportera comme si il était lancée pour la première fois, sauf si une autre sauvegarde a été créée.

16. *JSON* : https://www.json.org/
https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

3.5 Monde et carte

3.5.1 Génération d'astéroïdes

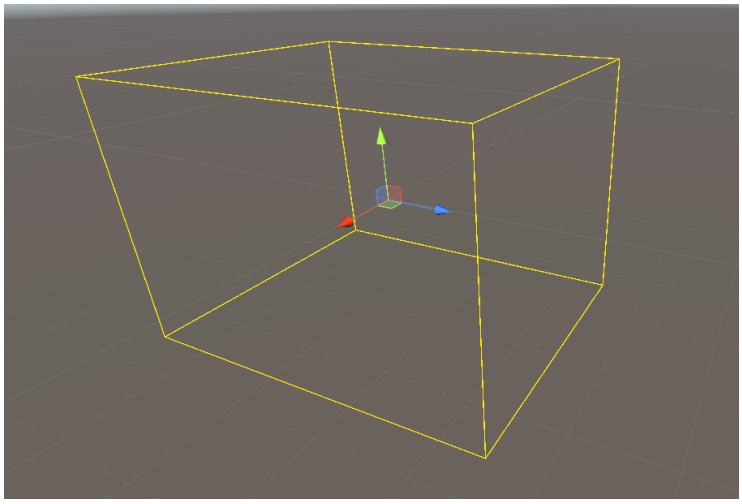
Parmi les nombreux scripts que nous avons écrit, il y a le générateur d'astéroïdes. Il constitue la majorité de la création de l'environnement de jeu. À chaque *Update*[17], ce script va compter le nombre d'astéroïdes présents autour du joueur. Si ce nombre est trop faible, il va instancier des astéroïdes de manière aléatoire autour de celui-ci (en faisant attention tout de même à ne pas instancier un astéroïde dans le joueur ou dans un autre astéroïde).

Génération d'un astéroïde :

La génération définit des caractéristiques aléatoirement, telles que sa position (tout de même comprise entre certaines bornes), sa taille (entre 0 et 3), et sa rotation. L'une des seules caractéristiques qui n'est pas aléatoire est la masse de l'astéroïde généré. En effet, celle-ci sera égale à 100 fois sa taille. Cela permet d'avoir un certain dynamisme/réalisme puisque les objets les plus gros seront moins influencés physiquement par les tirs des vaisseaux que les petits.

Prévisualisation :

La création de ce script nous a aussi permis de découvrir comment utiliser les Gizmos[18] dans les scripts. Ils se sont avérés très utile afin de visualiser la zone dans laquelle les astéroïdes sont générés, et donc quelle zone vérifiera le nombre d'astéroïde. Voici à quoi ressemble la zone :



17. La fonction *Update* est appelée à chaque image calculée du jeu. Pour plus d'informations :`https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html`
18. Les *Gizmos* sont des outils associés à un objet dans une scène. Ils permettent un débogage visuel et des outils d'aide dans la prévisualisation de la scène, comme la grille de la scène ou les boîtes de collision. `https://docs.unity3d.com/Manual/GizmosMenu.html`

3.5.2 Skybox

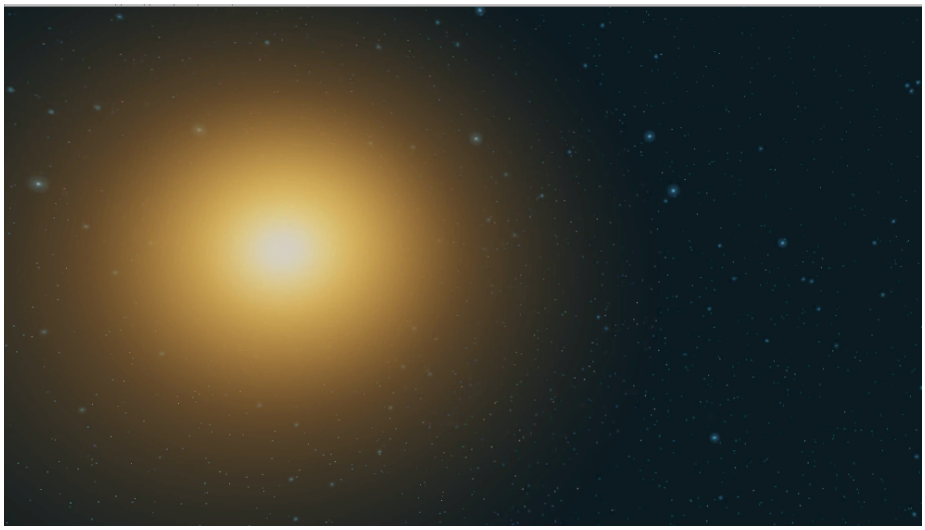
La *skybox*[19] est un élément important pour immerger le joueur dans le monde fictif de notre jeu vidéo car elle représente le décor. La première version avait été créée à l'aide du logiciel *Spacescape*[20]. Il s'agit d'un outil de création de *skybox* spatiales avec des étoiles et des nébuleuses. Ce logiciel est complet mais est difficile à prendre en main. Pour obtenir un bel univers, il faut créer différents calques, avec par exemple des points pour les étoiles. L'addition de plusieurs calques donne un résultat plus esthétique.

Lorsque la création de calques est terminée, le logiciel dispose d'une fonction d'exportation pour des moteurs de jeux vidéos dont Unity. L'importation dans Unity est ensuite très simple : 6 images sont créées et il suffit de suivre les instructions d'Unity.

Voilà la *skybox* que nous avons utilisé jusqu'alors :

19. Une *skybox*, comme son nom l'indique, représente le ciel. Elle montre à quoi ressemble le monde au delà de la distance atteignable. Elle donne l'illusion d'un monde plus grand qu'il n'est réellement. — `https://docs.unity3d.com/Manual/class-Skybox.html`
20. *Spacescape* : `https://sourceforge.net/projects/spacescape/`

Nous avons finalement décidé de nous tourner vers une autre *skybox*, qui était déjà présente dans le pack contenant les astéroïdes. Celle-ci est plus épurée, plus lumineuse, et la présence d'un soleil permet de se repérer plus facilement dans l'espace que précédemment.



Nous avons aussi, suite à se changement de *skybox*, changé la position et l'angle de la lumière directionnelle de la scène pour pouvoir faire en sorte que celle-ci provienne du soleil de la *skybox* et donc que les ombres soient dans le même sens que ce soleil.

3.5.3 Map

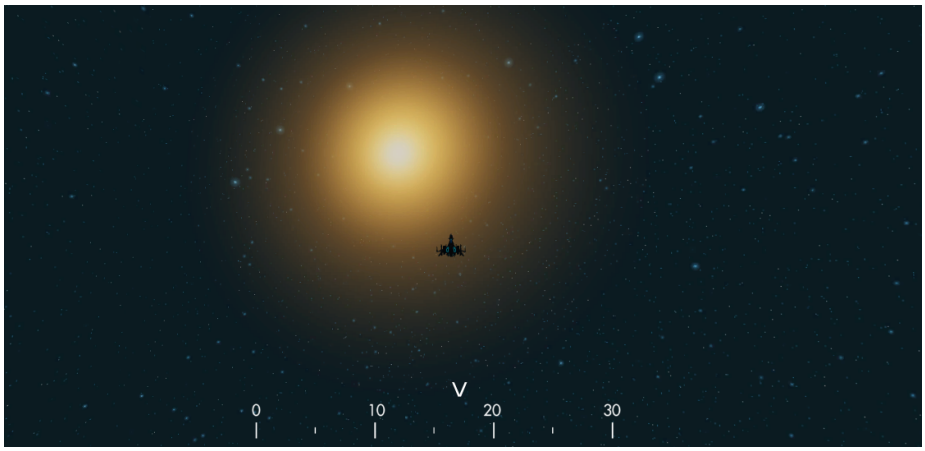
Pour permettre un meilleur repérage du joueur dans l'espace, nous avons ajouté une vue du dessus. La touche M permet de basculer entre deux modes de vue :

— La vue normale : une vue à la troisième personne, derrière le vaisseau.

— La vue "carte" : une vue de loin, au dessus du joueur.

Dans cette vue, les contrôles du vaisseau sont désactivés et le joueur peut avoir un aperçu de ce qui se trouve autour de lui. Pour permettre une plus grande flexibilité, le joueur peut, lorsque qu'il en mode de vue d'ensemble, zoomer ou dézoomer à l'aide de la molette de sa souris.

Vue d'ensemble autour du joueur :

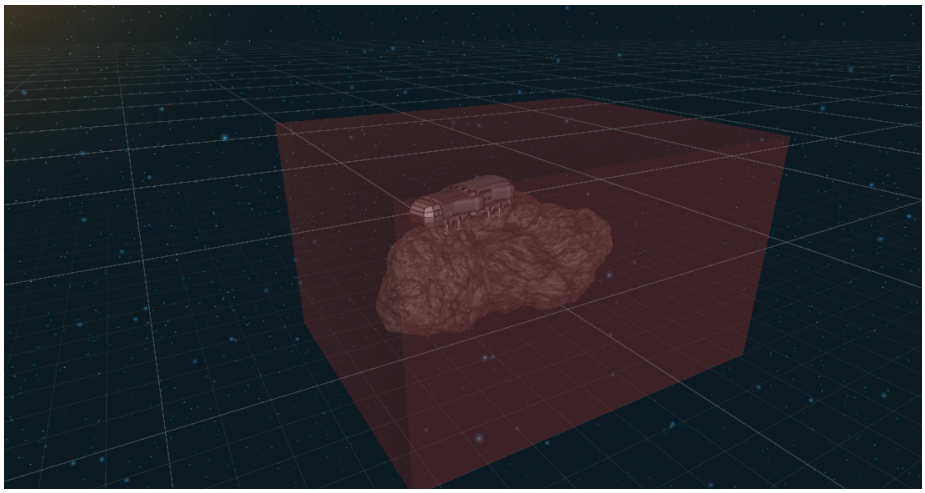


Pour le mode de vue normal, le joueur a la possibilité de faire pivoter la caméra autour de son vaisseau en conservant sa trajectoire en maintenant la molette de la souris appuyée, permettant de voir sur les côtés ou à l'arrière du vaisseau.

3.5.4 Les bâtiments

Mission : Livraison

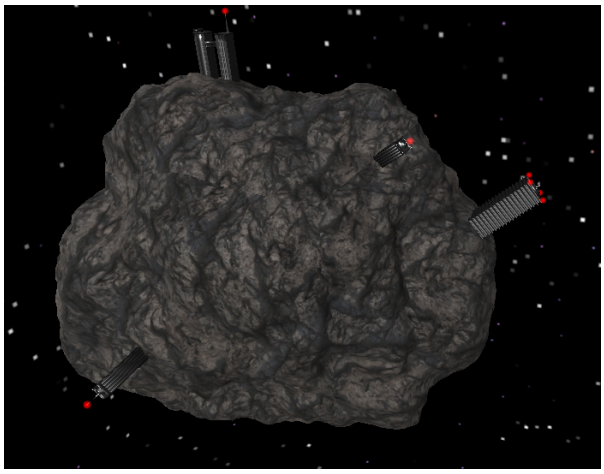
Ce bâtiment apparaît dès qu'une mission du type "livraison" est acceptée. Pour commencer la mission, il faut se rapprocher de celui-ci. Une fois fait, un bâtiment identique apparaît autre part dans l'espace et il faut rapporter le paquet à sa position. Les bâtiments disparaissent alors lorsque le joueur s'éloigne de celui-ci après avoir effectué sa mission.



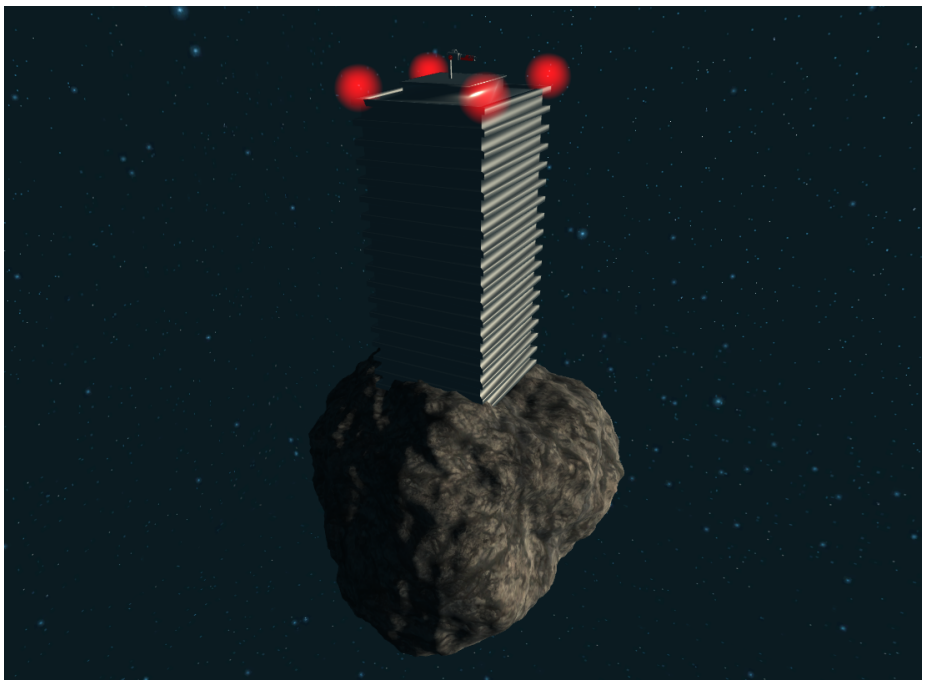
La zone rouge est celle à toucher pour récupérer le paquet

Mission : Destruction

Le bâtiment des missions de destruction est une tour, placée sur un astéroïde avec une tourelle au dessus. Cette tourelle tire automatiquement sur le joueur dès que celui-ci est à portée de vue. Cette tourelle prend le premier joueur dans son champ de vision et le verrouille. Dès que celui-ci sort de sa zone de tir, la tourelle arrêtere de tirer (sauf si un autre joueur est dans la zone, au quel cas elle le verrouillera).



À gauche, la première version du bâtiment de destruction. L'astéroïde était beaucoup plus gros, et possédait plusieurs bâtiments. En dessous, la version finale, avec un astéroïde beaucoup plus petit, un seul bâtiment mais toujours avec la tourelle sur son toit.



Mission : Destruction : La tourelle

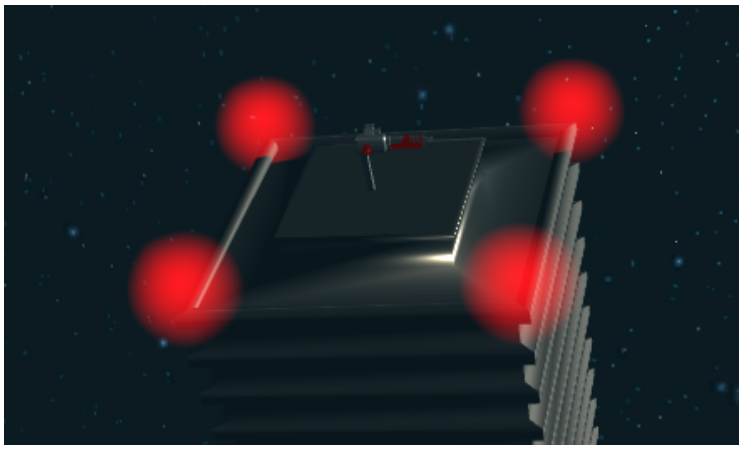
Le fonctionnement de la tourelle, survolé précédemment, va être un peu plus explicité ici.

À son instanciation, la fonction `GameObject.FindGameObjectsWithTag()` [21] va remplir un tableau à une dimension avec tout les objets possédant le tag "Player", c'est-à-dire tous les joueurs. Ensuite, à chaque Update, le script calculera grâce à la fonction `Vector3.Distance` la distance entre la tourelle et les joueurs contenus dans le tableau. Si l'un de ces joueurs est à une distance inférieure à la distance de verrouillage, ce joueur deviendra la "cible" de la tourelle. Si la distance qui sépare le joueur et la tourelle finit par être supérieure, la tourelle ne ciblera plus personne.

De plus, la tourelle possède sa propre vitesse de tir. En effet, à chaque Update, un *Timer* s'incrémentera et lorsqu'il aura dépassé un certain seuil fixé à l'avance, la tourelle tirera (ce qui réinitialisera le Timer par la même occasion).

Lors du développement de cette tourelle, les premiers soucis ont étés de la "maintenir" sur son socle, car elle vise le joueur en utilisant la fonction `Transform.LookAt()` et que celle-ci se sert du centre de rotation de l'objet. Comme la tourelle est plus maintenue d'un coté que de l'autre, celle-ci "flottait" dans les airs. Un autre problème a été de gérer la rotation de l'astéroïde. Comme la tourelle est posée sur le bâtiment, lui même posé sur l'astéroïde et que celui-ci a une rotation, il fallait que la tourelle calcule la différence entre sa propre rotation et la rotation de son point d'ancrage.

Ce qui est intéressant avec ce script est qu'il est capable de gérer plusieurs joueurs en même temps, grâce à ce système de "file d'attente" où le premier joueur à rentrer dans la zone sera le premier attaqué, puis le second, et ainsi de suite.



21. `GameObject.FindGameObjectsWithTag()` : `https://docs.unity3d.com/ScriptReference/GameObject.FindGameObjectsWithTag.html`

3.6 Multijoueur

3.6.1 Le multijoueur en jeu

Le multijoueur permet à plusieurs joueurs de se rejoindre dans une même partie. L'implémentation de cette fonctionnalité a été effectuée par Nicolas. Elle utilise l'outil **Photon**[22] pour permettre la communication entre les joueurs et faciliter le développement. Photon admet plusieurs avantages, comme sa facilité d'utilisation, mais également l'aspect *cloud*. En effet, l'éditeur de ce module s'occupe lui-même d'héberger les utilisateurs. Au lieu de fonctionner par partage d'IP comme dans de nombreux jeux, il suffit ici d'entrer le nom d'une salle pour rejoindre une partie. Il est possible de créer une salle mais celle-ci sera hébergée directement chez Photon au lieu d'être hébergée directement chez le joueur, ce qui permet une plus grande simplicité pour que plusieurs joueurs se rejoignent : il n'est par exemple pas nécessaire d'ouvrir des ports ou d'échanger des adresses IP, ce qui est un avantage pour les utilisateurs non initiés.

Le multijoueur est requis dans le cadre de ce projet et est une partie de travail non négligeable. Il nous a été recommandé à plusieurs reprises de faire de cette fonctionnalité une priorité et de tout implémenter autour d'elle pour ne pas avoir à tout recommencer par la suite. Ces différentes raisons expliquent pourquoi cette partie a été travaillée dès le début du projet. Le premier objectif était que deux joueurs puissent se rejoindre dans une même partie et puissent évoluer dans le même monde. Il a donc été nécessaire de faire des différents éléments qui composent un joueur un *prefab*, avec à l'intérieur le modèle 3D du joueur, sa caméra, ses scripts et ses effets. Il s'agit, comme son nom l'indique, d'un élément « préfabriqué ». Cet élément est ajouté à la scène à

22. *Photon* : `https://www.photonengine.com/`
Photon PUN : `https://www.photonengine.com/en-US/PUN`

chaque fois qu'un joueur rejoint une partie. On dit qu'il est « instancié ». Photon adapte des fonctions déjà existantes de Unity pour qu'elles fonctionnent dans l'environnement multijoueur. C'est par exemple le cas pour créer une instance de joueur : la fonction de base est `Instantiate()` [23] et celle de Photon est `PhotonNetwork.Instantiate()` [24]. Ces fonctions prennent en général les mêmes paramètres que celles de base, avec parfois quelques paramètres en plus liés à la mise en réseau.

Une partie importante dans le développement de cette fonctionnalité est la résolution de conflits. En effet, il faut s'assurer qu'un joueur ne voit et ne contrôle que son vaisseau et pas celui d'un autre joueur, tout en assurant que le monde dans lequel ils évoluent est le même pour chacun d'entre eux. Il faut donc dans de nombreux scripts, notamment celui qui gère les déplacements du vaisseau, vérifier que celui-ci est associé au bon joueur avant d'effectuer son code. Dans le cas contraire, la vérification se faisant en chaque début de script, la suite du code n'est en général pas effectuée. La caméra a posé le plus de problèmes malgré que la solution soit simple, il fallait seulement la laisser désactivée dans tous les cas et l'activer uniquement lorsqu'elle appartenait au joueur. La propriété `PhotonView.IsMine` s'est avérée très utile pour les différentes vérification.

Pour que des objets soient transmis à travers le réseau, il faut ajouter des composants du module Photon appelés *Photon View*. Il faut ensuite relier à ce composants les autres composants à synchroniser qui composent l'objet, tels que les *Rigidbody* qui permettent de leur appliquer de la physique. Des *Photon View* sont par exemple présents sur le joueur et les projectiles.

Nous avons ensuite ajouté la possibilité de choisir un pseudonyme qui sera affiché en jeu au dessus de leur vaisseau, comme l'image présentée plus tôt l'illustre.

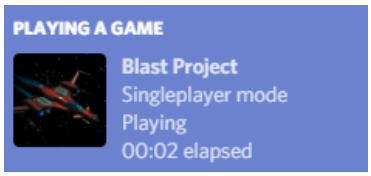
À chaque ajout de fonctionnalité, il a été nécessaire d'assurer que chacune d'entre elles étaient compatibles avec le multijoueur. Si ce n'était pas le cas, il fallait les retravailler. Cela a permis de garder un multijoueur fonctionnel tout au long du développement.

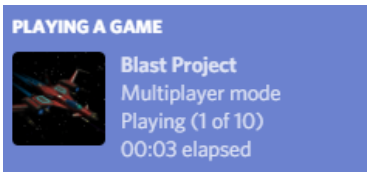
23. `Instantiate()` :
https://docs.unity3d.com/ScriptReference/Object.Instantiate.html
24. `PhotonNetwork.Instantiate()` :
https://doc.photonengine.com/en-us/pun/current/gameplay/instantiation

3.6.2 Intégration avec Discord Rich Presence

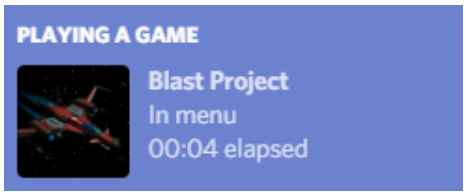
Comme de nombreux joueurs de jeux vidéos, et comme beaucoup de nos camarades à l'EPITA, nous utilisons beaucoup *Discord*[25] et les services que l'application rassemble. Discord est un logiciel de discussion par Internet. Ce service est développé à l'origine pour les joueurs et se connecte bien aux jeux vidéos. Nous avons alors pensé intégrer le service *Rich Presence*[26] de Discord à notre jeu vidéo. Cela a pour effet d'afficher sur le profil Discord de l'utilisateur la notification qu'il est en train de jouer à *Blast* après qu'il ait lancé le jeu.

L'information contient un certain nombre d'informations, que nous appelons le "*contexte*". Sur les deux images ci-dessous, on peut voir que l'affichage diffère selon si la partie dans le jeu est lancé en mode solo ou multijoueur.





Avant, cette notification n'apparaissait qu'une fois que le joueur ait lancé une partie. Maintenant, cette information apparaît dès le lancement du jeu, avec ce nouveau contexte utilisé pour l'affichage :



25. *Discord* : `https://discordapp.com/`
26. *Discord Rich Presence* : `https://discordapp.com/rich-presence`

3.7 Les Sons

N'ayant pas vraiment de connaissances en matière de conception sonore, nous avons pu récupérer des sons sur le site ZapSplat [27], une bibliothèque gratuite d'effets sonores libres de droits.

Un pack entier de plusieurs Mégaoctets contenait des sons de moteurs, de tirs de laser, mais aussi des voix d'une intelligence artificielle prononçant quelques mots. C'est avec tout cela que nous avons pu rajouter des sons à notre jeu vidéo.

Parmi ces sons, nous en trouvons deux prononcés par une IA :
— "Welcome", prononcé à l'ouverture du jeu.
— "Scan Complete", prononcé quand le joueur réussit à se connecter à Photon après avoir choisi le mode multijoueur et avoir entré son nom (si besoin).

Nous avons aussi ajouté des effet sonores, par exemple :
— Un bruit de tir laser pour chaque tir effectué par le joueur.
— Un bruit ambiant des moteurs.

Une intégration avec les scripts existant a été faite, permettant par exemple de faire varier l'intensité du bruit ambiant des moteurs en fonction de la vitesse du vaisseau du joueur. Cette intégration permet une meilleure immersion du joueur lors du contrôle de son vaisseau. Plus le joueur accélère, et donc plus le vaisseau va vite, plus le son des moteurs sera audible et aigu.

27. *ZapSplat* : `https://www.zapsplat.com/`

4 Commentaire sur l'avancement

Tâche	Soutenance 1	Soutenance 2	Soutenance 3
Sauvegardes	20%	50%	100%
Contrôles vaisseau	100%	100%	100%
Les objets	33%	75%	100%
Gestion du vaisseau	75%	100%	100%
Gestion des missions	10%	40%	90%
Monde/Carte	40%	90%	100%
Interface	33%	100%	100%
Multijoueur	40%	50%	70%
Site web	50%	100%	100%

Voici le planning des différents avancements prévus des parties du projet en fonction des soutenances.

Nous sommes plutôt satisfaits de notre travail sur ce projet. Nous avons bien avancé sur toutes les fonctionnalités sur lesquelles nous souhaitions travailler, et nous avons même pu développer quelques fonctionnalités supplémentaires.

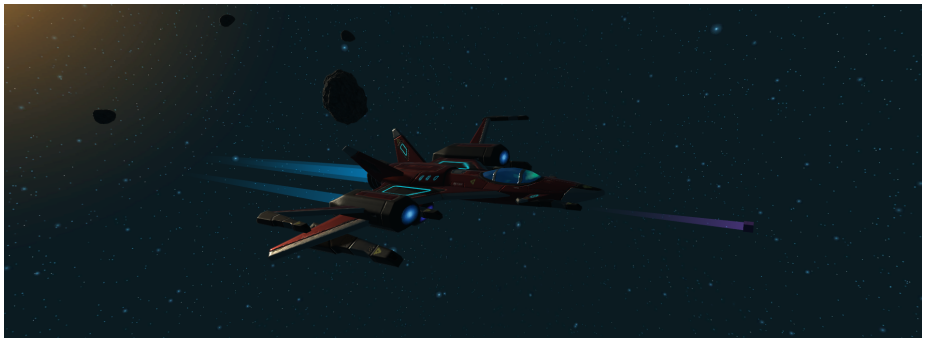
5 Le site web

Adresse web : `https://g00pix.github.io/ProjetBlast`

Le **site web** a été conçu simplement pour offrir une vitrine à notre projet. Il regroupe la présentation du projet, les liens de téléchargement du jeu et des rapports, ainsi que la présentation des membres du groupes.

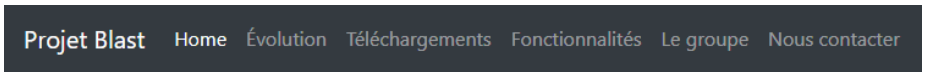
5.1 Présentation du site

Dès la page d'accueil, une capture d'écran du jeu permet de le présenter visuellement. L'image mise en avant (qui illustre aussi la section "Introduction" de ce rapport) présente très efficacement les graphismes et l'univers de notre jeu.



Cette image marque aussi fortement l'évolution qu'il y a eu depuis la dernière soutenance, puisqu'elle remplace l'image que nous utilisions précédemment pour ce rôle :

5.2 Plan du site



Voilà le plan de notre site internet, tel que le menu le présente :

— La page d'accueil
— La page de l'évolution du projet
— La page des téléchargements
— La page des fonctionnalités du jeu
— La page de présentation du groupe
— La page de contact

La page d'accueil est la page qui intègre l'image que nous avons présenté dans la précédente sous-section, ainsi qu'un présentation rapide du projet.

La page qui retrace les évolution majeures du projet permet de mettre en évidence les difficultés et les moments importants que notre projet ou notre groupe ont traversés.

La page des téléchargements contient les liens pour télécharger le jeu, ainsi que les fichiers en rapport avec le projet : notamment le cahier des charges et les rapports de soutenance.

La page qui présente les fonctionnalités majeures du jeu permet d'offrir au visiteur un aperçu de ce qui est possible de faire dans notre jeu vidéo.

La page de présentation du groupe présente ce dernier, ainsi que chacun de ses membres. Des liens ont été ajoutés pour contacter les membres individuellement par mail, ou pour accéder à leur profil sur GitHub.

Finalement, la page de contact permet de rédiger puis d'envoyer un mail collectif à tous les membres du groupe.

5.3 Informations techniques

Le site internet est hébergé sur *GitHub* grâce à son service ***GitHub Pages***[28] et est accessible publiquement à l'adresse web indiquée au début de cette section. Cette méthode est très pratique car le code source du site internet est lié à celui du projet dans le même repository. Le code source du site est conservé dans une branche appelée « *gh-pages* » alors que le code source du jeu vidéo est conservé dans une autre branche, appelée « *master* ». Toutes les modifications faites dans la branche du site internet seront automatiquement répercutées sur le site lui-même. Ce déploiement est rapide et facilite grandement la collaboration en groupe.

GitHub Pages utilise *Jekyll*[29] pour construire les site web que le service héberge. *Jekyll* est un outil de génération de site internet statique. Il offre quelques fonctionnalités pratique au développeur, et nous en avons donc tiré profit.

Pour le moment, le site utilise le framework *Bootstrap*[30] pour faciliter le développement et l'ajout d'éléments esthétiques, sans avoir à perdre beaucoup de temps sur du code CSS qui n'est pas l'objectif de ce projet. Son utilisation est simple et bien documentée, il suffit d'ajouter des éléments HTML contenant les noms de classes correspondant pour que le site prenne forme rapidement.

Grâce à *Jekyll*, le développement du site se fait à l'aide de templates HTML qui seront remplies par *Github Pages* à chaque déploiement. Le contenu s'écrit majoritairement en *Markdown*[31], un langage de balisage léger, ce qui rend l'écriture plus simple. Il est possible d'ajouter des petites parties de code HTML dans les fichiers de contenu pour différencier certaines pages et y ajouter des éléments plus complexes, qui interagissent avec le framework *Bootstrap*.

28. *GitHub Pages* : `https://pages.github.com/`
29. *Jekyll* : `https://jekyllrb.com/`
30. *Bootstrap* : `https://getbootstrap.com/`
31. *Markdown* : `https://fr.wikipedia.org/wiki/Markdown`

6 Téléchargement et Installation

6.1 Téléchargement

Le jeu est disponible au téléchargement sur le site web du Projet Blast, dans la section **Téléchargements**.

6.2 Installation

Lorsque l'utilisateur clique sur le bouton *Télécharger*, un fichier du format *.msi*[32] est reçu. Ce type est le format d'installateur historique du système d'exploitation *Microsoft Windows*. Sa création est compliquée et nécessite un ensemble d'outils appelés *WiX Toolset*[33]. La programmation de cet installateur se fait avec le langage de balisage XML[34]. Le grand inconvénient de ce type d'installateur est la complexité en ce qui concerne sa programmation. En effet, il est nécessaire de lister manuellement chaque fichier ainsi chaque dossier et sous-dossier. Le choix a été porté sur cette méthode dans le but d'apprendre, car elle est très utilisée dans le monde de l'informatique. En effet, même *Microsoft* l'utilise pour installer ses différents produits comme le pack de bureautique *Office*.

Pour installer le jeu, l'utilisateur a uniquement besoin de suivre les étapes qui lui sont proposées. L'installation démarre alors. Un raccourci est créé sur le bureau et l'installateur se ferme. L'utilisateur peut alors directement lancer le jeu.

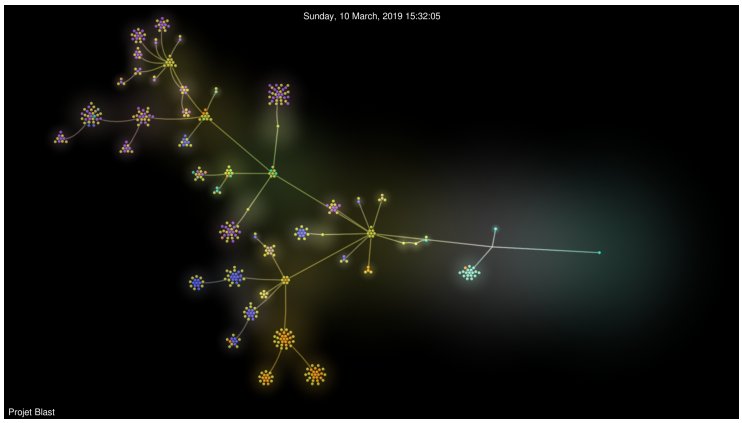
32. *.msi* : `https://fr.wikipedia.org/wiki/Windows_Installer`
33. *WiX Toolset* : `https://wixtoolset.org/`
34. *XML* : `https://fr.wikipedia.org/wiki/Extensible_Markup_Language`

7 Structure du repository

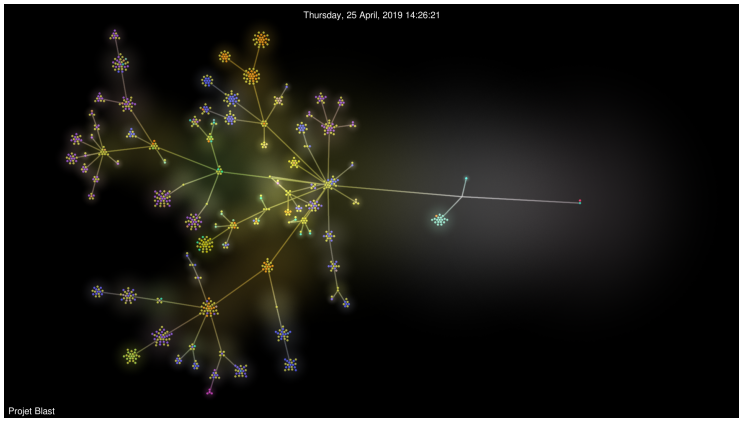
7.1 Structure physique

Comme dans nos rapports de soutenance, voilà la représentation schématisée de la structure du *repository Git* de notre projet. Cette visualisation a été générée à l'aide de l'outil Gource [35] à partir de l'historique des modification de notre projet. Chaque branche représente un dossier, chaque élément représente un fichier et chaque couleur représente un type de fichier.

À la 1ère, la structure ressemblait à cela :

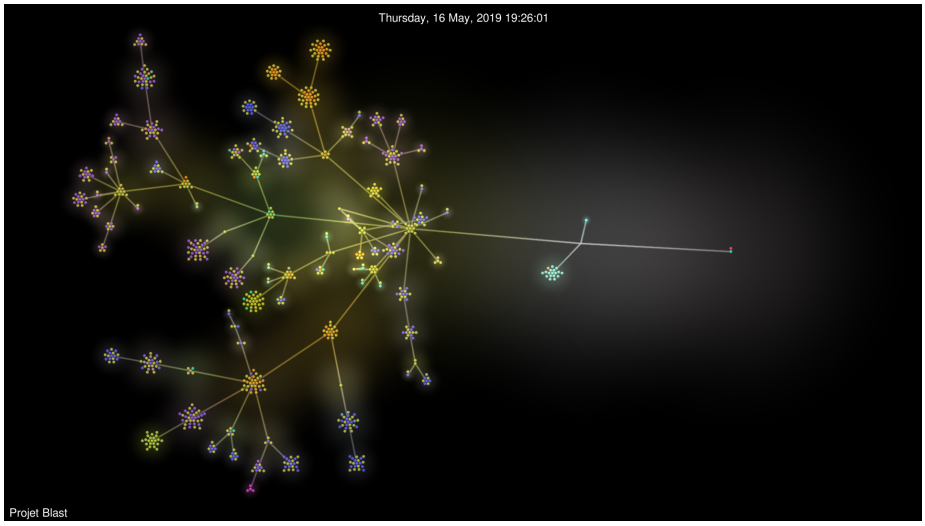


À la 2nde soutenance, elle ressemblait à cela :



35. *Gource* : `https://gource.io/`

Aujourd'hui, voilà à quoi ressemble la structure finale de notre projet :



Au bout droit, on trouve la racine du projet. De droite à gauche, on trouve sur l'arbre :

- Les fichiers de configuration Unity, dont la majorité sont en bleu ciel.
- Au centre, les fichiers du jeu que nous avons développé, y compris scripts et éléments du jeu.
- À gauche, les grandes branches de la partie haute correspondant aux trois librairies principales que nous avons utilisé :
  - StarSparrow, une bibliothèque de vaisseaux que nous utilisons dans le jeu.
  - Photon, un framework réseau qui nous aide pour l'implémentation du multijoueur.
  - TextMeshPro (une plus petite branche), une librairie qui nous permet d'améliorer les textes de l'interface.
- Enfin, toujours à gauche, dans la partie basse du schéma, la dernière grande branche correspond à tous les *assets* qui ont été nécessaires à la conception du nouveau menu de notre jeu.

7.2 Notre projet sur Github

Le repository, et en particulier GitHub, nous permet aussi de travailler plus efficacement en groupe sur le projet. Nous avons mis à profit les outils de collaboration qui nous sont à notre disposition : à savoir la création d'*issues* et de *pull requests*.

Les *issues* nous permettent de signaler et de discuter des bugs rencontrés, ainsi que des futurs fonctionnalités que nous prévoyons d'ajouter à notre jeu. Elles permettront par la suite lorsque le projet sera disponible au public de signaler des bugs détectés par les joueurs que nous pourront alors corriger.

Les *pull requests* sont créées à chaque fois que le développement d'une fonctionnalité majeure est terminé. Cela permet à l'équipe de se mettre au courant du développement fait par d'autres, ainsi que de donner leur avis sur le travail qui a été fait.

En quelques chiffres, notre projet sur Github représente :
— Plus de **250 commits** et 3 collaborateurs.
— Un repository d'un taille de plus de **150 Mo**.
— Plus de **30 pull requests** résolues.
— Plus de **10 issues** créées.

8 Documentation et Informations

8.1 Documentation

Ce projet aura nécessité un gros travail de documentation. Nos sources principales ont bien sûr été la documentation *Unity*[36] ainsi que celle de *PhotonEngine*[37] (concernant le multijoueur), mais aussi *MSDN*[38] (pour le langage *C#* en lui même) et enfin le forum d'entraide *StackOverflow*[39].

8.2 Librairies et Assets

Nous avons utilisé plusieurs librairies dans notre projet. En voici la liste :

— *StarSparrow*, une librairie gratuite sur le *Unity Asset Store*. Elle contient les modèles 3D des vaisseaux, avec des variations de couleurs, les matériaux et les textures de ceux-ci ainsi que les Mesh (très utiles, surtout pour les mesh-colliders).

— *Photon PUN* : Il s'agit d'un moteur réseau qui permet de gérer entièrement le multi-joueur.

— *TextMeshPro* permet d'écrire des textes plus graphiques et plus colorés que l'éditeur de base de Unity, plutôt pauvre de ce coté-ci.

36. *Unity* : `https://docs.unity3d.com/Manual/index.html`
37. *Photon PUN* : `https://doc.photonengine.com/en-us/pun`
38. *MSDN* : `https://msdn.microsoft.com/`
39. *StackOverflow* : `https://stackoverflow.com/`

9 Expériences personnelles

9.1 Nicolas FROGER (chef du projet)

J'appréhendais beaucoup la création de ce jeu vidéo. En effet, je voyais cela comme un projet de très grande envergure, ce qui était une grande première pour moi. Ce projet s'est avéré difficile mais je le termine avec un sentiment positif. En effet, cela a été l'occasion pour moi d'en apprendre beaucoup sur le travail d'équipe car il a fallu s'adapter à de nombreuses reprises. J'ai du apprendre à travailler avec des personnes que je ne connaissais pas auparavant, et cela ne m'a pas dérangé. Je suis fier que notre groupe, malgré qu'il ait subit beaucoup de changements au cours du temps, ait réussi à produire un jeu fonctionnel qui nous plaît. J'ai dû assumer le rôle de chef de groupe, ce qui était également une première fois pour moi. Cela a été un très bon exercice car ce n'était pas toujours évident.

J'ai également pu mettre en œuvre mes connaissances en programmation que j'avais pu acquérir par le passé mais aussi avec l'école dans un projet concret de plus grande envergure que les travaux pratiques hebdomadaires. J'ai également pu améliorer ces connaissances car ce projet a nécessité bon nombre de recherches. Ce projet m'a permis de m'entraîner sur un aspect qui me semble fondamental dans le développement qui est le passage de l'idée au code. En effet, durant cette année, les travaux pratiques étaient tous très dirigés et s'apparentaient plus à du remplissage qu'à de la réflexion profonde, ce qui est normal pour débuter. J'ai donc pu m'entraîner à conceptualiser une idée en quelque chose de concret, faisable en code.

9.2 Mathieu GUÉRIN

Dans le rapport de 1ère soutenance, j'avais exprimé mon intérêt pour la conception d'un jeu vidéo, et c'est toujours vrai. Le projet de S2 était un projet concret et une opportunité de découvrir les outils lié au développement d'un jeu vidéo et des méthodes de travail qui en découlent, et il l'a été.

J'ai beaucoup appris grâce à la conférence Unity donnée par *GConfs* en février 2019, et depuis grâce à des tutoriels et de la documentation sur Internet. Mes connaissances personnelles avec *Git* et *GitHub*, celles acquises en *C#* grâce à l'EPITA m'auront beaucoup aidé. Quelques expériences personnelles avec d'autres outils m'auront aidé pour les travaux annexes lié au projet :
L'utilisation et le développement d'*API*[40] en tout genre pour des implémentation sur certains de mes projets personnels m'ont permis d'acquérir les compétences nécessaires pour intégrer *Discord Rich Presence* au projet.
L'utilisation, depuis longtemps, de *Bootstrap*, *GitHub Pages* et de *Jekyll* pour la conception du site web de notre projet.

De plus, mon expérience a par exemple pu aider par exemple Pierre dans son apprentissage pour l'utilisation de *Git* et *GitHub*.

Ma méthode de développement se base toujours sur mon expérience de jeu en tant que joueur. Quasiment tout mon travail peut se résumer à ça : améliorer le jeu d'après nos objectifs, tout en améliorant l'expérience du joueur, que ce soit au niveau du *gameplay*, ou du ressenti. J'ai ajouté beaucoup de fonctionnalités sur le ressenti, puisque c'est ce qui m'amuse particulièrement : voir concrètement le résultat de mes ajouts au projet.

Cette dernière période est arrivée très vite. Entre les autres projets lié à l'EPITA, les différentes démarches administratives pour notre scolarité, et bientôt les partiels, il a été plutôt difficile de tenir le rythme.

Cependant, nous avons tout de même pu nous accrocher. Nous avons réussi à finir ce qui avait été démarré et ce qui était toujours en développement lors de la 2nde soutenance. Aujourd'hui, le résultat de notre projet nous satisfait et est terminé.

40. *API* : `https://fr.wikipedia.org/wiki/Interface_de_programmation`

9.3 Pierre DE LA RUFFIE

Alors, beaucoup de choses à dire sur ce projet. Déjà je suis très fier du travail que nous avons fourni. Ensuite, je sors de cette dernière partie de développement avec le sentiment d'avoir accumulé beaucoup de connaissances sur Unity et sur le C#. Voulant travailler plus tard sur ce moteur de jeu, j'ai tout de suite pris ce projet très au sérieux, le voyant aussi comme un moyen d'avoir un aperçu sur ce qui m'attendrais par la suite.

J'ai d'abord eu beaucoup de mal à me lancer dans ce projet de second semestre, que ce soit dans le groupe **Overlord** ou dans le groupe **Quadro**. Pour le second, il m'a fallut un très très long temps d'adaptation. Déjà pour me mettre à niveaux sur *Git* (aujourd'hui encore je ne maîtrise pas l'outil à 100%), puis au niveau des scripts que mes camarades produisaient et qui m'étaient difficile de comprendre. Cependant, à travers la distribution des rôles que nous avions effectués, j'ai pu apprendre l'existence de certaines spécificités de *Unity* que je ne connaissais pas, comme les MeshCollider ou les Gizmos par exemple.

Je me suis beaucoup amusé à faire le menu, les différentes scènes avec les mouvements de caméra, et les animations. J'ai aussi découvert ce qu'étais vraiment la génération procédurale sur *Unity* et en C#. Même si je n'ai qu'effleuré la surface de ce sujet avec le générateur d'astéroïdes, j'ai regardé beaucoup de tutoriels (notamment celui de *Brackeys*[41]) et j'ai déjà beaucoup d'idées pour d'autres projets sur *Unity* utilisant ce genre de mécanique.

Je garde tout de même certains regrets sur ce projet. Par exemple, j'aurais aimé pouvoir faire de la musique mais je me suis très vite rendu compte que cela nécessitais des connaissances plus complexes que ce que j'imaginais. De plus le logiciel que j'avais prévu pour faire cela s'est avéré incapable d'exporter la musique dans sa version gratuite. N'ayant pas envie de dépenser d'argent pour ce projet, j'ai finalement décidé d'abandonner et de me focaliser sur des sons qui augmenteraient l'immersion.

Cependant, je reste globalement satisfait du travail que nous avons accompli. Ce projet a vraiment été pour moi une expérience enrichissante et un amusement certain. J'espère pouvoir à nouveaux un jour refaire ce genre de projets.

41. `https://www.youtube.com/watch?v=64NblGkAabk`

10 Conclusion

Le Projet Blast a connu des débuts difficiles suite au départ de deux membre du groupe dès le commencement ainsi qu'un début du développement un peu tardif. Cependant, beaucoup de travail a été effectué et nous sommes aujourd'hui très fiers du résultat. Ce projet aura été pour nous l'occasion d'apprendre à travailler efficacement en équipe, mais aussi d'améliorer nos connaissances en programmation et de les utiliser dans la conception d'un projet de grande ampleur. Nous avons eu l'occasion d'expérimenter dans le domaine de la conception de jeu vidéo, un domaine intéressant et ludique, que nous risquons de ne pas revoir dans la suite de nos études. Ce tout premier projet à EPITA a été une expérience très intéressante et enrichissante.